Saxon User Documentation

Version 9.4

Saxon User Documentation: Version 9.4

Table of Contents

List of Tables

Chapter 1. About Saxon

Introduction

To help you find your way around this documentation, there is a full table of contents [../contents.html] and also an alphabetical index.

The most significant new feature in Saxon 9.4 is the introduction of bytecode generation (in Saxon-EE only). Other new developments are comprehensively listed in Changes in Saxon 9.4.

Saxon implements the latest working drafts of XSLT 3.0, XQuery 3.0. XPath 3.0, and XSD 1.1.

In addition Saxon continues to offer highly conformant implementations of XSLT 2.0. XPath 2.0, XQuery 1.0, and XSD 1.0. To avoid compromising this conformance, features from the newer versions of the standards are enabled only if explicitly requested. The conformance to these specifications has been upgraded to implement the latest published errata, as consolidated in the proposed Second Editions of the specifications.

A significant change in the 9.2 release was a repackaging of the functionality, and this is retained in 9.3 and 9.4: instead of the two products Saxon-B (open source) and Saxon-SA (commercial), there are now three editions: home, professional, and enterprise (HE, PE, and EE). The Home Edition remains open source, under the same very liberal licensing rules, but some of the more advanced features that were present in Saxon-B have not been retained in Saxon-HE, and are instead available only in the commercial packages Saxon-PE and Saxon-EE.

Much of the new functionality in this release is available only in the commercial offerings.

Like previous releases, this version is shipped simultaneously for the Java and .NET platforms.

This documentation covers all versions of the product, with differences noted where applicable. For a summary of the differences, see Product Packages.

> This documentation (including full API documentation for both the Java and .NET platforms) is provided on the Saxonica [http://www.saxonica.com/] web site and is also available for download either from SourceForge [http://sourceforge.net/project/showfiles.php?group_id=29872] or from Saxonica [http://www.saxonica.com/download/download_page_fs.html]. The download file is named `saxon-resources9-n.zip`, and is separate from the software download. It includes documentation for all editions (Saxon-HE, -PE, and -EE), on both platforms. The file also includes sample applications. Saxon-HE source code for the latest release is also available for download from the SourceForge project page.

JavaDoc API specifications [../javadoc/index.html] and .NET API specifications [../dotnetdoc/index.html] are also available.

A full change log is provided.

If you are looking for an XSLT 1.0 or XPath 1.0 processor, Saxon 6.5.5 [http://saxon.sourceforge.net/saxon6.5.5/] remains available. However, Saxon 9.4 is capable of executing XSLT 1.0 stylesheets with identical results in the vast majority of cases, often with much better performance, so you may prefer to use the latest release.

Getting Started

If you are new to Saxon, or to Java, or to XML, XSLT, and XQuery, this section provides a quick checklist of the things you need to do before you can run your first application.

First decide which platform you want to use. If you already use either the Java or .NET platform, it's simplest to use the platform you are familiar with. If you don't use either of these platforms, we recommend starting with Java, as Saxon has been available on Java for longer and this version is therefore better integrated and better documented. If you currently use both, then the choice largely depends on what language you want to use to write your own applications.

- Getting started with Saxon on the Java platform
- Getting started with Saxon on the .NET platform

Getting started with Saxon on the Java platform

Saxon doesn't come with a graphical user interface: it's designed to be integrated into other tools and applications. You will therefore start by using it from the operating system command line. On Windows, you may want to install a text editor such as `jEdit` that offers a more friendly command line than the standard DOS console provided by Microsoft. However, if you're not comfortable running applications from the command line, you might like to try the open-source Kernow [http://kernowforsaxon.sourceforge.net/] product from Andrew Welch. This installs Saxon for you.

1. Ensure that Java is installed. This must be Java JDK 1.5 (also known as J2SE 5.0) or later. Since version 9.2, Saxon will no longer run with JDK 1.4 or earlier releases. To check that Java is installed, try typing `java -version` at the command prompt. If it is not installed, or if you have an older version, you can get the software from Oracle [http://www.oracle.com/technetwork/java/javase/downloads/index.html].

2. Download the Saxon software (you will typically start with the open-source version) from SourceForge [http://sourceforge.net/project/showfiles.php?group_id=29872].

3. The software comes as a zip file. Unzip it into a suitable directory, for example `c:\saxon`.

4. To check that the software is working, try running a simple query from the command line, as follows. The filename after the "-cp" flag should match the location where you installed the software. The file will be called `saxon9he.jar` for Saxon Home Edition, `saxon9pe.jar` for the Professional Edition, or `saxon9ee.jar` for the Enterprise Edition:

   Here tells the operating system to run the Java virtual machine; the filename after is known as the classpath, and tells Java where to find the Saxon application software; is the Saxon entry point for the XQuery processor; the option tells Saxon to report what it is doing on the console; and the option tells Saxon to run the simple query `current-date()`, which returns the current date and displays the result on the console.

   If this doesn't work, there are three possible reasons: the operating system can't find the Java software; Java can't find the Saxon software; or Saxon has been invoked, but has failed to run your query. The error message should give you a clue which of these is the case.

5. As a first-time user, you will probably want to install the sample applications. These are packaged together with a copy of this documentation and Saxon-B source code in the file `saxon-resources9-n.zip`. This can be downloaded from either the SourceForge or Saxonica sites. (It is the same file in both cases). Unzip the contents of this file into the same directory as the Saxon software. When you run a program that uses Saxon, this program as well as Saxon itself will need to be on the classpath. For more details of the classpath, see Installing (Java).

6. If you are using features specific to Saxon-PE or Saxon-EE, the commercial editions of Saxon, you will need to obtain a file containing a licence key. You can apply for a free 30-day license key by going to www.saxonica.com [http://www.saxonica.com/] and clicking on "Evaluation Copy". The license key file will be sent by email within 24 hours. This file, called `saxon-license.lic`, should be placed in the same directory where the file `saxon9pe.jar` or `saxon9ee.jar` is found.

You can now run one of the sample stylesheets or queries that comes with in the `saxon-resources` download. Assuming you installed Saxon-HE into `c:\saxon`, make this your current directory, and type:

```
java -cp saxon9he.jar net.sf.saxon.Transform -t -s:samples\data\books.xml
     -xsl:samples\styles\books.xsl -o:c:\temp.html
```

```
java -cp saxon9he.jar net.sf.saxon.Query -t
   -s:samples\data\books.xml -q:samples\query\books-to-html.xq >c:\temp.html
```

Now open `c:\temp.html` in your browser to check that it worked.

For more details on installing Saxon on the Java platform, see Installing (Java)

Getting started with Saxon on the .NET platform

1. Download the software (you will typically start with the open-source version) from SourceForge [http://sourceforge.net/project/showfiles.php?group_id=29872]. It is designed to be used with .NET version 3.5 from Microsoft and has also been used successfully with other versions. It has not been run successfully with the Mono platform, though it has not been thoroughly tested on other platforms, and there may be undocumented restrictions.

2. The software comes as a .exe installer `SaxonHEn-n-n-nNsetup.exe`. Double-click it to run the installer. It installs the executable files into a user-selected directory, sets a registry entry to point to this location, and registers the relevant DLLs in the Global Assembly Cache.

3. As a first-time user, you will probably want to install the sample applications. These are packaged together with a copy of this documentation and Saxon-B source code in the file `saxon-resources9-n.zip`. This can be downloaded from either the SourceForge or Saxonica sites. (It is the same file in both cases). Unzip the contents of this file into the same directory.

4. Make sure that .NET version 2.0 or higher is installed on your machine. This is available as a free download from Microsoft.

5. If you are using a commercial edition of Saxon (that is, Saxon-PE or Saxon-EE), you will need to obtain a file containing a licence key.

   a. Apply for a free 30-day license key by going to www.saxonica.com [http://www.saxonica.com/] and clicking on "Evaluation Copy". Be sure to specify that you want a license key for the .NET platform.

   b. The license key file will be sent by email within 24 hours. This file, called `saxon-license.lic`, should be placed in the directory containing the Saxon .dll files: for example `c:/Program Files/Saxonica/SaxonHE9.3N/bin`

6. Saxon doesn't come with a graphical user interface: it's designed to be integrated into other tools and applications. You will therefore start by using it from the operating system command line. On Windows, you may want to install a text editor such as UltraEdit [http://www.ultraedit.com/] that offers a more friendly command line than the standard DOS console provided by Microsoft.

You can now run one of the sample stylesheets or queries that comes with the `saxon-resources` download. Assuming you installed into `c:\saxon`, make this your current directory, and type:

```
bin\Transform -t -s:samples\data\books.xml
     -xsl:samples\styles\books.xsl -o:c:\temp.html
```

```
bin\Query -t -s:samples\data\books.xml
```

```
-q:samples\query\books-to-html.xq -o:c:\temp.html
```

Now open `c:\temp.html` in your browser to check that it worked.

For more complete details on installing Saxon on the .NET platform, see Installing (.NET)

What is Saxon?

The Saxon package is a collection of tools for processing XML documents. The main components are:

- An XSLT 2.0 processor, which can be used from the command line, or invoked from an application, using a supplied API. This can also be used to run XSLT 1.0 stylesheets.
- An XPath 2.0 processor accessible to applications via a supplied API.
- An XQuery 1.0 processor that can be used from the command line, or invoked from an application by use of a supplied API. This includes support for XQuery Updates 1.0 which is now a W3C Recommendation.
- An XML Schema 1.0 processor. This can be used on its own to validate a schema for correctness, or to validate a source document against the definitions in a schema. It is also used to support the schema-aware functionality of the XSLT and XQuery processors. Like the other tools, it can be run from the command line, or invoked from an application.
- As well as providing full implementations of the stable specifications listed above, Saxon also has a track record of early implementation of forthcoming standards. Saxon 9.4 offers a complete implementation of XML Schema 1.1, which is now in the final stages of standardization. It also implements many features from the forthcoming XSLT 3.0, XQuery 3.0, and XPath 3.0 working drafts which are still at an earlier stage of development.
- On the Java platform, when using XSLT, XPath, or XML schema validation, Saxon offers a choice of APIs. If you need portability across different vendor's tools, you can use the JAXP API for XSLT, XPath, and XML Schema processing, and the XQJ interface for XQuery. On the other hand, if you want a more integrated and complete API offering access to all Saxon's facilities, the s9api interface is recommended. You can also dive down deeper into the Saxon internals if you need to: there has been no particular attempt to make interfaces private, and all public interfaces are documented in the JavaDoc. Clearly, the deeper you go, the greater the risk of interfaces changing in future releases.
- On the .NET platform, Saxon offers an API that enables close integration with other services available from .NET, notably the XML-related classes in the `System.Xml` namespace. It isn't possible to use Saxon as a transparent plug-in replacement for the `System.Xml.Xsl` processor, because the API for the Microsoft engine using concrete classes rather than abstract interfaces. However, it is possible to use it as a functional replacement with minor changes to your application code.

Full details of Saxon's conformance to the specifications are provided in the Conformance section.

In addition, Saxon provides an extensive library of extensions, all implemented in conformance with the XSLT and XQuery Recommendations to ensure that portable stylesheets and queries can be written. These include the EXSLT [http://www.exslt.org/] extension libraries , , , and . Many of these extensions were pioneered in Saxon and have since become available in other products.

These extension functions are in general accessible from XQuery and XPath as well as XSLT, except where they depend on stylesheet information. Many extensions are available in Saxon-PE only, and some only in Saxon-EE.

Choosing a software package

Saxon is distributed in three packages: Saxon-HE, Saxon-PE, and Saxon-EE, each of which is available for both the Java and .NET platforms. Saxon-HE is available under an open-source license (specifically, the Mozilla Public License), and all its features are available to all users. Saxon-PE and Saxon-EE are commercial products and require activation by a license key.

There are also restricted licenses that work with Saxon-EE but restrict its capability:

- The EET license provides schema validation and schema-aware XSLT transformation only (no XQuery)
- The EEQ license provides schema validation and schema-aware XQuery only (no XSLT)
- The EEV license provides schema validation only (no XQuery or XSLT)

For a full table showing the features available in each edition and licensing variant, see the feature matrix [http://www.saxonica.com/feature-matrix.xml]

For commercial information including prices, terms and conditions, maintenance and upgrade offerings, site licensing, and redistribution licensing, see the online store [http://www.saxonica.com/shop/index.html].

Installation: Java platform

This section explains in more detail how to install Saxon on the Java platform.

- Prerequisites
- Obtaining a license key
- Troubleshooting license key problems
- Installing the software
- JAR files included in the product

Prerequisites

The following software must be installed separately, it is not included with the Saxon download.

- To run Saxon you need at least a Java VM, and preferably a Java development environment. Saxon 9.3 requires JDK 1.5 (properly the Java 2 Platform, Standard Edition 5.0) or later (it also runs under JDK 1.6). If for some reason you need to run under JDK 1.4, you will need to stick with Saxon 9.1 or an earlier release.
- If you use the XQJ XQuery API in Saxon, then you will need the StAX parser interfaces to be present on your classpath. These are available as standard in JDK 1.6. With JDK 1.5, however, you will need to install them separately. They can be obtained in the file `jsr173_1.0_api.jar` obtainable from https://sjsxp.dev.java.net/.
- Saxon will also accept input from a `StAXSource`. This is a new class introduced in JAXP 1.4. It is available as standard in JDK 1.6, but to use it with JDK 1.5, you will need to install it separately: it can be found in the JAR file `jaxp-api.jar` which can be downloaded from https://jaxp.dev.java.net/.
- Saxon has options to work with source trees constructed using DOM, JDOM, JDOM2, XOM, or DOM4J:
  - To use Saxon with DOM, you do not need any extra code on your classpath. (The relevant code has been integrated into the main JAR file; this has become possible because JDK 1.4 is no longer supported.) The DOM implementation should support DOM Level-3 interfaces. Saxon retains some legacy code designed to handle level-2 DOM implementations, which

can be activated by calling the method `configuration.setDOMLevel(2)` on class `net.sf.saxon.Configuration`, but this code is now untested and unsupported. Saxon is validated using the DOM implementation by the Apache Xerces parser, and the DOM that is packaged with the Sun JDK. It should work with other DOM implementations, but this can never be 100% guaranteed without testing. Many DOM implementations, especially non-productized implementations, deviate in minor but significant ways from the specifications.

- For the other object models, JDOM, JDOM2, XOM, and DOM4J, the supporting Saxon adapter code is integrated into the JAR files for Saxon-PE and Saxon-EE, but is available only as source code for Saxon-HE. To exploit this code with the open source Home Edition, you will need to compile the source code from the relevant subpackage of `net.sf.saxon.option`. Whichever edition you are using, the external object model is available for use only if you register it using the method `registerExternalObjectModel [Javadoc: net.sf.saxon.Configuration#registerExternalObjectModel]` on the `Configuration [Javadoc: net.sf.saxon.Configuration]` object, or via the configuration file. Saxon no longer searches the classpath to see which object models are present, because classpath searches are expensive and make the application over-sensitive to details of the way it is run.
-
- Saxon has been validated with JDOM 1.0, JDOM 1.1.1, XOM 1.1, XOM 1.2.1, XOM 1.2.6, and DOM4J 1.6.1. Saxon has been tested with the alpha release of DOM4J 2.0.0 (which at the time of writing has been at alpha status for 18 months). Preliminary testing has been carried out with JDOM 2, but this interface is not released with Saxon 9.4 because JDOM 2 is not yet sufficiently stable.

By default Saxon uses an XML parser that supports the SAX2 interface. Saxon has been tested successfully in the past with a wide variety of such parsers including Ælfred, Xerces, Lark, SUN Project X, Crimson, Piccolo, Oracle XML, xerces, xml4j, and xp. By default, however, it uses the parser that comes with the Java platform (a version of Xerces in the case of JDK 1.5). The parser must be SAX2-compliant. All the relevant JAR files must be installed on your Java CLASSPATH.

Saxon will also work with a StAX parser. Generally speaking, StAX parsers are currently less mature than SAX parsers, and any performance advantage is likely to be very minor. However, the support for StAX provides the ability to supply input via a customized pull pipeline. Saxon is tested with Woodstox 3.0.0. Saxon's schema validation is available only with a SAX2 parser.

Obtaining a license key

The open-source Saxon-HE product does not require any license key. For Saxon-PE and Saxon-EE, however, you need to obtain a license key from Saxonica. You can order a free license key for a 30-day evaluation, or purchase an unrestricted license key, from the Saxonica [http://www.saxonica.com/] web site.

Since Saxon 9.2, newly issued license keys are compatible across the Java and .NET platforms. The older .NET license keys (named `saxon-license.xml`) are no longer accepted.

The license key will arrive in the form of a file named `saxon-license.lic` attached to an email message. Saxon will search for the license key in the following locations:

1. The location specified using the configuration property `FeatureKeys.LICENSE_FILE_LOCATION [Javadoc: net.sf.saxon.lib.FeatureKeys]`, as described below.
2. The directory containing the executable Saxon software, that is `saxon9ee.jar` or `saxon9pe.jar` on the Java platform, and `saxon9pe.dll` or `saxon9ee.dll` on the .NET platform. (On .NET, remember that the software may have been moved to the Global Assembly Cache.).

3. On the Java platform only, the directories identified by the environment variable `SAXON_HOME` and `SAXON_HOME/bin`.

4. On the .NET platform only, the installation directory - that is, the directory chosen for installing the software when the installation wizard was run. This directory is identified in the Windows registry.

5. All directories on the Java class path. When running from the command line, you can set the classpath using the `-cp` option on the `java` command, or via the environment variable `CLASSPATH`. More complex environments might have other ways of building the classpath. In Tomcat, for example, the license file should be treated in the same way as free-standing (unjarred) classes, and placed in the `WEB-INF/classes` directory.

It is also possible to force Saxon to read the license file from a specific location by setting the Configuration property `FeatureKeys.LICENSE_FILE_LOCATION` (a constant representing the string `"http://saxon.sf.net/feature/licenseFileLocation"`). When this property is set, the `Configuration` will immediately attempt to load the specified file, avoiding any subsequent attempt to search for it on the classpath and elsewhere. This mechanism is useful in environments that are otherwise difficult to control: for example configuration properties can be specified as attributes to the `factory` element of Ant's `<xslt>` task.

On the Java platform it is generally simplest to copy the file to the directory in which `saxon9ee.jar` is installed. On .NET it is generally simplest to copy the file to the directory in which `saxon9ee.dll` is installed, and then set the environment variable `SAXON_HOME` to point to this location.

If you acquire Saxon-PE or Saxon-EE as a component of an application with which it has been integrated, the application may activate the software automatically using an OEM license key. In this case you do not need to acquire an individual license key of your own.

Troubleshooting license key problems

If you are having trouble running Saxon-PE or Saxon-EE for the first time, here is a list of the most common problems:

- within 24 hours, it may be because it has been intercepted by your company's spam/virus checker. This is surprisingly common. Send us an email and we will try to find a way through your company's defences.

  Another possible cause is that you provided an email address that doesn't work. Again, it's surprising how many people do this.

And some less common problems:

- . Some mail systems, notably Yahoo, have been known to change the hyphen to an underscore in transit.

- , please read the instructions above for where to install the license key once again.

- , this may mean that you are running Saxon-HE software, or that you have invoked Saxon using a Saxon-HE entry point, or that Saxon has reverted to Saxon-HE functionality because it could not find a valid license file (in the latter case it will have produced a message to this effect: but depending on your application, you might not see the message).

- then it probably is. Very occasionally a license file gets corrupted in transit. Please ask for another copy (or we might issue you a new license). Sometimes the corruptions can be traced to the use of non-ASCII characters in license fields such as your name or Company (mail systems can be unkind to such files). In the event of problems, try applying again restricting yourself to ASCII characters.

- this generally means you are using a 30-day evaluation license and the expiry date has been reached. (Perhaps you have obtained a new license, key, and Saxon is still picking up the old one.) If it's taking you longer to evaluate the software than you planned, please send us an email explaining the circumstances before you apply for another license, otherwise the request may be refused.

- , this usually means that you purchased a standard license key which covers free upgrades for one year, and that year has elapsed. You will need to purchase an upgrade, which can be done online. Most paid-for licenses are permanent, but only in relation to software issued within one year of the purchase.

Installing the software

An index of all currently-available open-source versions of Saxon is on the download page at SourceForge [http://sourceforge.net/project/showfiles.php?group_id=29872]

For the commercial versions of Saxon, please follow the download links from http://www.saxonica.com/

Please see the change log for details of all changes in this release.

Installation of Saxon on the Java platform simply involves unzipping the supplied download file into a suitable directory. The procedure is the same for all editions (HE, PE, and EE). The JAR files are differently named in each edition.

One of the files that will be created in this directory is the principal JAR file. This is , , or depending on which Saxon edition you are using.

There are some additional JAR files to support optional features such as the SQL interface. When running Saxon, the principal JAR file should be on the class path. The class path is normally represented by an environment variable named CLASSPATH: see your Java documentation for details. Note that the JAR file itself (not the directory that contains it) should be named on the class path.

Some environments (for example Java IDEs such as Eclipse, and servlet containers such as tomcat) build the classpath themselves: they might ask you to register the location of JAR files via a graphical user interface or a text configuration file, or they might ask you to put the JAR files in a well known place (for example `WEB-INF/lib` in the case of tomcat). Follow the instructions for the environment you are using.

JAR files included in the product

The full list of JAR files in the Saxon distribution is as follows:

Table 1.1.

saxon9he.jar	Saxon Home Edition. Contains all the software in Saxon-HE.
saxon9pe.jar	Saxon Professional Edition. Contains all the software in Saxon-PE, with the exception of the SQL extension.
saxon9ee.jar	Saxon Enterprise Edition. Contains all the software in Saxon-EE, with the exception of the SQL extension and the code for compiling XQuery to Java.
saxon9-sql.jar	Supports XSLT extensions for accessing and updating a relational database from within a

	stylesheet. Provided with Saxon-PE and Saxon-EE.
saxon9-unpack.jar	Allows a "packaged stylesheet" to be executed. The JAR file is provided with Saxon-HE, Saxon-PE and Saxon-EE. Packaged stylesheets can only be created using Saxon-PE or Saxon-EE; for details see Packaged Stylesheets.

When running any Java application, Saxon included, all Java classes that are needed must be present on the CLASSPATH. The classpath can be set in the form of an environment variable, or it can be included in the `java` command that invokes the application.

The classpath is written as a list of filenames. These will either be the names of directories (folders) that contain relevant classes, or the names of JAR files containing the classes. On Windows, the names in the list are separated by semicolons; on Linux, a colon is used.

The table above lists the JAR files provided with Saxon that you may need to include on your classpath. In addition, you may need to include some of the following resources:

Table 1.2.

saxon-license.lic	License file. This is needed only for running Saxon-EE/Saxon-PE. The license file is obtained when you purchase the product or when you apply for an evaluation license.
jaxp-api.jar	This file implements parts of JAXP 1.4 that Saxon uses. It is not needed when you run Saxon under Java 6, as the required classes are then included in the standard Java run-time. It is needed, however, if you run Saxon under Java 5. This JAR file is not included in the Saxon distribution because of licensing restrictions. Instead, it must be downloaded from https://jaxp.dev.java.net/. The `jaxp-api.jar` file is needed when you use the `StAXSource` class.

Source code and documentation are not included in the same download file as the executable code and sample files. Instead, they are in a resources file that can be downloaded separately. The documentation is also available online.

The source code for Saxon-HE (on both the Java and .NET platforms) is available in the package `saxon-resources-9.x` from either the SourceForge or the Saxonica sites. Note that it is not necessary to download or recompile the source code unless you need to make changes (and it is rarely necessary to make changes, because Saxon provides extensive system programming hooks to enable customization). The exception to this is that in Saxon-HE, adapters for external object models, and localization code for non-English languages, are provided in source form only.

The modules included in the source code of Saxon-HE are also used, without modification, in Saxon-PE and Saxon-EE. Source code for the additional components of the commercial editions is not provided. The source code is the same between the Java and .NET platforms: both products are generated from the same source, though there are some modules that apply only to one platform or the other. There are no build scripts provided for rebuilding the product from source code, because most users find it convenient to use their Java IDE for this purpose.

User documentation (this documentation, together with API specifications for the Java and .NET products) is also available for download from the SourceForge site. The download package contains exactly the same information as the documentation section of the Saxonica website. It also contains this documentation in its original XML form, together with the schemas and stylesheets used to publish it. Note that online documentation is available only for a few recent Saxon versions; older versions can be downloaded from SourceForge.

Installation: .NET platform

This section explains in more detail how to install Saxon on the .NET platform.

The source code of Saxon is written in Java. The version of the product that runs on the .NET platform has been produced by cross-compiling the Java bytecode into CIL code, which is then packaged as a .NET assembly. The cross-compiler is the open-source IKVMC [http://www.ikvm.net/] product. The code of course makes many calls on classes provided by the Java run-time library. On the .NET platform, most of these are provided by the OpenJDK product: Saxon comes with a version of the OpenJDK library, itself cross-compiled as a .NET assembly. Other run-time services are obtained from the .NET platform itself. These include XML parsing, handling of URI resolution, and internationalized collation support.

For more information about Saxon on the .NET platform, see Saxon on .NET.

Prerequisites: .NET platform

The following software must be installed separately, it is not included with the Saxon download.

- To run Saxon you need to install the .NET platform, version 2.0 or later.

Because Saxon is run from the command line, you might find it useful (under Microsoft Windows) to have a text editor with better command-line support than the standard DOS window. I use UltraEdit [http://www.ultraedit.com].

Installing the software

The software is issued in the form of a .exe file containing an installation wizard: all you need to do is to run it, answering simple prompts such as the directory in which you want it installed.

The main components are as follows:

Table 1.3.

saxon9he.dll, saxon9pe.dll or saxon9ee.dll	The Saxon library for Home Edition, Professional Edition, or Enterprise Edition
saxon9he-api.dll, saxon9pe-api.dll, saxon9ee-api.dll	The classes implementing Saxon's .NET API. The three DLLs are identical except that they contain references to the appropriate version of the main Saxon DLL.
Transform.exe	Command line entry-point for XSLT transformation
Query.exe	Command line entry-point for XQuery evaluation
Validate.exe	(Saxon-EE only) Command line entry-point for XML Schema validation
IKVM.Runtime.dll	Run-time library for the IKVM cross-compiler
IKVM.OpenJDK.XXXX.dll	A number of DLL files containing parts of the Java OpenJDK class library cross-compiled to .NET

These files will typically be found in a folder with a name such as `c:\Program Files\Saxonica\SaxonHE9.3N\bin`

You may find it useful to add this directory to the `PATH` environment variable. This enables you to use the commands `Transform`, `Query`, and `Validate` without identifying their location explicitly.

Obtaining a license key

The open-source Saxon-HE product does not require any license key. For Saxon-PE and Saxon-EE, however, you need to obtain a license key from Saxonica. You can order a free license key for a 30-day evaluation, or purchase an unrestricted license key from the Saxonica [http://www.saxonica.com/] web site.

Since Saxon 9.2 license keys have been compatible between the Java and .NET platforms.

The license key will arrive in the form of a file named `saxon-license.lic` attached to an email message. Copy the file into the `/bin` directory containing the Saxon DLLs, for example `c:\Program Files\Saxonica\SaxonEE9.3N\bin`

If you acquire Saxon-PE or Saxon-EE as a component of an application with which it has been integrated, the application may activate the software automatically using an OEM license key. In this case you do not need to acquire an individual license key of your own.

Sample applications

Saxon on .NET is distributed with a number of sample applications. These are issued (together with Saxon-HE source code and documentation) in a separate download file `saxon-resources9-n.zip`, available from both the SourceForge and Saxonica web sites. Once this is unzipped, the sample applications can be found in the directory `/samples`. They are described here.

Many of the samples are equally applicable to the Java and .NET platforms. However, there are several programs in the `/samples/cs` directory that are specifically designed to illustrate ways of using the `Saxon.Api` interface. This interface is exclusive to the .NET product, and is described in more detail in Saxon API for .NET. These sample applications are described in Example applications for .NET.

Historical Note

Saxon has been under development since 1998. Most of the code was written by one person, Michael Kay, which has resulted in a high level of design integrity. More recently O'Neil Delpratt has joined Saxonica's development team and has contributed extensively to the development of Saxon 9.4.

Saxon was originally written to support an internal project in ICL (now part of Fujitsu [http://www.fujitsu.com]), and ICL continued to sponsor development of Saxon until Michael Kay left the company in January 2001. ICL chose not to market it as a commercial product, but to make the code available to the public under the Mozilla public license. From 2001 through 2003 Michael Kay worked for Software AG [http://www.softwareag.com], who continued to sponsor the development of Saxon as an open source product.

In March 2004 Michael Kay founded Saxonica Limited [http://www.saxonica.com/] to provide ongoing development and support of Saxon as a commercial venture. Saxonica continues to develop the basic (non-schema-aware) version of Saxon as an open source product, while at the same time delivering professional services and additional software (Saxon-PE and Saxon-EE) as commercial offerings. The commercial product incorporates the code of the open-source product in its entirety, with the addition of schema-processing technology, and is produced in accordance with the provisions defined by the Mozilla Public License.

The port of Saxon to the .NET platform was pioneered by Pieter Siegers Kort and M. David Peterson, without any involvement from Saxonica. Their work was absorbed into the Saxonica product line from

Saxon 8.7 onwards. The Saxonica product used the same approach as the previous Saxon.NET product for cross-compiling the code into CIL assemblies. In addition, however, it provided a new .NET API for use by C# and other .NET applications, and made much greater use of .NET services such as collations and regular expression processing. This integration was done by Saxonica with generous advice from M. David Peterson. The project would not have been possible without the IKVMC cross-compilation technology developed by Jeroen Frijters, as well as the GNU Classpath developed by a large team of individual enthusiasts. The use of GNU Classpath was subsequently discontinued and replaced with OpenJDK.

The name Saxon was chosen because originally it was a layer on top of SAX. Also, it originally used the Ælfred parser (among others); Ælfred of course was a Saxon king...

Technical Support

Please read the Conditions of Use.

The open-source Saxon-HE product comes with no warranty and no formal technical support service.

- Lists and forums for getting help
- Bugs and patches

Lists and forums for getting help

If you have a general question about XSLT or XPath programming, that is not specifically related to Saxon, we recommend the xsl-list [http://www.mulberrytech.com/xsl/xsl-list/]: check first that the query isn't already covered in the FAQ. The archives of the list can be searched at MarkMail [http://xsl-list.markmail.org/]. Other useful sites for XSLT information are www.xslt.com [http://www.xslt.com/] and www.jenitennison.com [http://www.jenitennison.com/].

Similarly, there are a number of help forums for XQuery: for details see the XQuery home page at http://www.w3.org/XML/Query. The most active is talk at xquery.com [http://www.xquery.com/].

The StackOverflow [http://stackoverflow.com/] is another useful resource for asking about all manner of XML-related (and other) topics.

If you have questions specific to Saxon you can usually get an answer by raising them on the Saxon help list at http://lists.sourceforge.net/lists/listinfo/saxon-help [https://lists.sourceforge.net/lists/listinfo/saxon-help]. You will need to register.

Alternatively, you can also use the saxon-help forum on the SourceForge project site. Saxonica monitors both; however, if you want to get input from other Saxon users, the mailing list is usually more effective. You can also use the support-requests tracker on SourceForge; unlike the forum, this allows attachments to be uploaded.

> Please do not ask for help via the bug tracker on SourceForge. We prefer to keep this for confirmed bugs only. It's much easier to search it for known bugs if it isn't cluttered with support requests that are not bugs at all.

Bugs and patches

If you hit something that looks like a bug, please check the known errors on the Saxon project pages at SourceForge [http://sourceforge.net/projects/saxon]. Also check the list archives.

Please don't enter the problem into the bug register until it is confirmed as a bug: it's easier for everyone to search for real bugs if the register isn't cluttered with problems that turned out not to be bugs at all.

If you need to submit attachments, this is best done through the tracker on the SourceForge site.

Saxon-EE and Saxon-PE users are provided with an email address that allows bug reports to be sent to Saxonica privately if preferred.

The open source code of Saxon-HE is maintained in a Subversion repository on the SourceForge site. This exists solely to provide early access to source patches, it is not (currently) used to deliver incremental releases of new functionality. Building the Jar files is reasonably easy for an experienced Java programmer; an Ant script is provided to assist with this, but it is likely to require customization to local requirements (for example, deleting the parts concerned only with Saxon-EE or Saxon on .NET). Building the .NET product is considerably more complicated, and although scripts are provided, you should only attempt it if you have a good knowledge of both platforms.

If you wish to contribute modifications to the open source code, you will need to be prepared to sign a contributor agreement defining the terms under which the code is made available. This may need the written agreement of your employer. You should also check with Saxonica in advance to discuss the format of test material to be submitted along with code changes; proposed contributions have often been rejected because they came without adequate tests.

Related Products

This section lists some Saxon add-ons and extensions produced by third parties. Saxonica Limited takes no responsibility for the quality of these products. More information about third party products is available on the Saxon wiki pages [https://sourceforge.net/apps/mediawiki/saxon/index.php?title=Main_Page]

Open Source tools

Kernow [http://sourceforge.net/projects/kernowforsaxon/] is a graphical front-end for Saxon. This is an open-source product developed by Andrew Welch. It provides an effective alternative to the Saxon command line interface for users who prefer a GUI for running ad-hoc transformations.

On MAC OS/X, Todd Ditchendorf has produced XSLPalette [http://www.ditchnet.org/xslpalette/], which is designed to integrate with your own choice of text editor to provide XSLT development and debugging support.

Commercial Editors and Debuggers

A number of commercial XML IDEs provide support for XML, XSLT, and/or XQuery editing and debugging, with the ability to configure Saxon as the chosen XSLT/XQuery processor. These include:

- Stylus Studio [http://www.stylusstudio.com/]: offers XSLT, XQuery, and XML Schema development and debugging, all using Saxon. This is currently the only IDE that offers Saxon-EE built-in.
- oXygen [http://www.oxygenxml.com/] offers XSLT and XML Schema development and debugging for Saxon. Integrates Saxon-B, and works with Saxon-EE which you must purchase separately.
- A budget XML Editor and Debugger for Saxon is EditiX [http://www.editix.com/], with prices starting at $22.

- XML Spy from Altova [http://www.altova.com/] allows you to configure Saxon as your XSLT or XQuery processor, but does not provide Saxon debugging. There is also a third-party plug-in [http://members.chello.at/spiffbase/spycomponents/ValidatorBuddy.htm] that allows Saxon-EE to be used for schema validation.

XQuery Documentation

is a Javadoc-like documentation tool for XQuery. It works with Saxon, and is available either as a free-standing tool from http://xqdoc.org/ or as part of Stylus Studio [http://www.stylusstudio.com/].

Chapter 2. Changes in this Release

Version 9.4 (2011-12-09)

Details of changes in Saxon 9.4 are detailed on the following pages:

- Bytecode generation
- Reading source documents
- XPath 3.0 changes
- XSLT changes
- XSLT Packaged Stylesheets
- XQuery 3.0 changes
- Changes to XSD support
- Changes to Functions and Operators
- Changes to Saxon extensions and extensibility mechanisms
- Changes to application programming interfaces
- Changes to system programming interfaces

Bytecode generation

Saxon-EE 9.4 selectively compiles stylesheets and queries into Java bytecode before execution.

This change should be largely invisible to users, apart from the performance gain, which in typical cases is around 25%.

Where Saxon decides that generating bytecode would be advantageous, the bytecode is generated as the final stage in the query or stylesheet compilation process. The bytecode is held in memory (never written out to disk) as part of the expression tree. Interpreted expressions can call compiled expressions, and vice versa.

Bytecode generation is done using the ASM library, which is included as an integral part of the Saxon-EE JAR file.

Bytecode generation occurs even in the .NET version of the product. The bytecode is automatically and dynamically converted to .NET IL code prior to execution, by the IKVM runtime.

There are several configuration options associated with bytecode generation. The option `GENERATE_BYTE_CODE` can be set to `false` to disable byte code generation (which might be useful, for example, when doing low-level debugging). The option `DEBUG_BYTE_CODE` can be set to cause the generated byte code to contain debugging information. And the option `DISPLAY_BYTE_CODE` can be set to cause the generated code to be displayed (on the standard error output).

The facility to generate byte code replaces the facility in Saxon 9.3 to generate Java source code (which was available for XQuery only). It also replaces the XSLT facility for "compiled stylesheets".

Reading source documents

During 2011, W3C have taken steps to reduce the burden of meeting requests for commonly-referenced documents such as the DTD for XHTML. The W3C web server is routinely rejecting such requests, causing parsing failures. In response to this, Saxon now includes copies of these documents within the issued JAR file, and recognizes requests for these documents, satisfying the request using the local copy. For details see References to W3C DTDs.

In addition, Saxon 9.4 command line interfaces have been enhanced with a new option, `-catalog:filename` that causes URIs and public identifiers to be resolved by reference to an OASIS catalog. For details see Using XML Catalogs.

The PTree now retains the is-id and is-idref properties of attributes. This has been done by using previously unused bits; since the old `PTreeReader [Javadoc: com.saxonica.ptree.PTreeReader]` can read the new format, and the new `PTreeReader [Javadoc: com.saxonica.ptree.PTreeReader]` can read the old format, no new version number has been introduced.

XPath 3.0 changes

Support for the primitive type `xs:precisionDecimal` has been dropped, since it has been removed from XSD 1.1.

The XPath 3.0 string concatenation operator ("||", borrowed from SQL) is implemented.

The XPath 3.0 simple mapping operator ("!") is implemented.

Casting from strings (and `xs:untypedAtomic`) to union and list types is now supported. The effect is the same as casting to an attribute with the given type, and then atomizing the attribute. For example casting to a type `list-of-integer` defined as a list type with an item type of `xs:integer` returns a sequence of `xs:integer` values.

Certain union types can now appear in a `SequenceType` (e.g as the declared type of a function argument). The union types accepted are those that are not derived by restriction from another union types, and whose membership includes only atomic types and other union types that meet the same criteria.

The XQuery/XPath 3.0 parser has been extended to support partial function application ("?" as a function argument) in dynamic function calls. Previously this feature was supported only in direct function calls to a named function.

The implementation of maps has been updated to match the draft XSLT 3.0 spec. The extensions for maps are available in XPath 3.0 and XQuery 3.0, rather than being restricted to XSLT. This may change as the W3C specifications evolve.

XSLT changes

See also XPath 3.0 changes.

XSLT 2.0 implementation

The call `system-property('xsl:vendor')` now returns the string "Saxonica". Previously it returned a more complex string that also identified the product version; this information should now be obtained using the XSLT 2.0 system properties such as `system-property('xsl:product-version')`. Information on the values returned for each system property is at XSLT 2.0 Conformance.

The handling of type errors in an `<xsl:if>`, or an `<xsl:choose>` with no `<xsl:otherwise>` branch, has been made less draconian. If the construct appears in a context where it is not allowed to return an empty sequence, the type error is no longer reported statically, but is only reported if the implicit else/otherwise branch is actually taken at run-time. The immediate motivation for this change is that it enables stylesheets generated using Altova's products to be executed; it also appears to be a more reasonable interpretation of the intent of the specification.

Command line

The implementation of the `-TP` option on the command line, which produces a timing profile, has been rewritten. Rather than producing a trace file containing all the events with timings, which could become very voluminous, it now aggregates the timing data in memory, and outputs the results directly in HTML rather than requiring a separate post-processing step.

On the `net.sf.saxon.Transform` command line, there is a new option `-threads:N` controlling how many threads are to be used. This only has effect when the `-s` option specifies a directory. It does not cause individual transformations to be multi-threaded, it only causes the transformations of different files to run in parallel with each other.

The new `-catalog:filename` option requests use of OASIS catalogs for resolving DTD references, external entity references, URIs appearing in `xsl:include` and `xsl:import` declarations, and calls to the `doc()` and `document()` functions.

In Saxon-EE, Java bytecode is generated automatically unless suppressed using the option `--generateByteCode:off`

XSLT 3.0 Features

XSLT 3.0 features are available only if XSLT 3.0 support is explicitly requested, for example by specifying `-xsltversion:3.0` on the command line.

Maps, as defined in the draft XSLT 3.0 specification, are implemented as an extension to XPath 3.0. For details see Maps in XPath 3.0.

The `xsl:merge` instruction is implemented.

Pattern syntax in the form `~ItemType` is supported, for example `match="~xs:integer"` matches an integer. Predicates are allowed on such patterns, for example `~xs:integer[. gt 0]`.

Associated with this change, `xsl:apply-templates` (as well as `xsl:next-match` and `xsl:apply-imports` can be used to process any kind of item, not only nodes (for example, atomic values or maps)

Similarly, `xsl:for-each-group` with the `group-starting-with` or `group-ending-with` patterns can now process a sequence of atomic values.

In the `xsl:mode` declaration, the values supported for the `on-no-match` attribute have changed, in line with changes in the XSLT 3.0 working draft. The option `copy` is renamed `shallow-copy`, `stringify` is renamed `text-only-copy`, and `discard` is renamed `deep-skip`. Two new options are added: `deep-copy` and `shallow-skip`.

When `xsl:copy` is used with a `select` attribute (new feature in XSLT 3.0), the context item for evaluation of the contained sequence constructor is now the item selected by the `select` attribute.

The `intersect` and `except` operators can now be used in match patterns; multiple operators and parentheses are allowed. Parentheses are also allowed around an expression that is then filtered by a predicate, for example `match="(foo|bar)[*]"` or `(//para)[1]`.

XSLT Packaged Stylesheets

The `net.sf.saxon.CompileStylesheet` facility, which serialized the internal representation of a stylesheet, is withdrawn in this release. Because the feature offered few performance benefits, it was used mainly to enable XSLT stylesheets to be distributed without revealing the source code to users, thus preserving the intellectual property of the author. The feature is replaced by a new facility designed to achieve the same effect in a cleaner way, with fewer restrictions.

Saxon-PE/EE now provide a command to create a packaged stylesheet in the form of a ZIP file. The contents are in obfuscated form so the source code is not visible. A JAR file, `saxon9-unpack.jar`, is available with all Saxon editions allowing a stylesheet that comes in this form to be executed. This JAR file contains Saxonica proprietary (non-open-source) code, but is available at no cost and does not require a license key to run.

For full details of the facility see Packaged Stylesheets.

XQuery 3.0 changes

See also XPath 3.0 changes.

The function annotations `%public` and `%private` are implemented (they were available in 9.3 without the "%" sign).

The variable annotations `%public` and `%private` are implemented (they were not available in 9.3).

A case clause in a `typeswitch` expression can list multiple alternative types separated by the "|" operator.

In FLWOR expressions, the `count` clause is implemented; implementation of `group-by` has been completed, and sliding and tumbling windows are implemented. Implementation of XQuery 3.0 FLWOR expressions is thus functionally complete. To achieve this, a new internal design has been adopted, using tuple streams in a manner very close to that described in the specification.

The implementation of the `-TP` option on the command line, which produces a timing profile, has been rewritten. Rather than producing a trace file containing all the events with timings, which could become very voluminous, it now aggregates the timing data in memory, and outputs the results directly in HTML rather than requiring a separate post-processing step.

Changes to XSD support

Various minor changes have been made to bring Saxon 9.4 into full conformance with the XSD 1.1 specification. Most of these are edge cases and bug fixes, mostly prompted by resolution of issues in the specification.

Assertions in XSD 1.1 now use the correct rules for typing (that is, for type annotation of the tree made visible to the XPath expression defining the assertion).

Following its removal from XSD 1.1, support for the `xs:precisionDecimal` data type has been dropped.

In regular expressions, the block names that are recognized (whether running XSD 1.0 or 1.1) in constructs of the form `\p{IsBlockName}` now include blocks defined in all Unicode versions from version 3.1 to version 6.0 inclusive. Where the characters included in a block differ from one version to another, Saxon recognizes the union of the various definitions. When XSD 1.1 is enabled Saxon treats an unrecognized block name as a warning condition rather than an error; in this case both `\p{IsBlock}` and `\P{IsBlock}` will match any character. For example, Saxon now recognizes both `\p{IsGreek}` and `\p{IsGreekAndCoptic}`. Internally, the recognition of block names no longer depends on what is recognized by the underlying Java regex library.

In regular expressions, character category matches such as `\P{Nd}` now use the categorization of characters defined in Unicode 6.0.0. This not only affects characters added to Unicode since Unicode version 3.1, it also changes the categorization of some existing characters.

In regular expressions, the metacharacters `\i` and `\c` now use the definitions of name characters from XML 1.0 fifth edition and XML 1.1 second edition (which are identical), regardless whether XML 1.1 support is enabled or not.

All testing of names in Saxon now uses the rules defined in XML 1.1 and in XML 1.0 fifth edition (which are identical), whether or not XML 1.1 or XSD 1.1 is enabled. This includes `\i` and `\c` in regular expressions, the rules for data types such as `xs:NCName` and `xs:QName`, parsing of names in XPath expressions and XQuery element constructors, etc. The only thing not covered is where names are checked by non-Saxon software, for example the XML parser.

A configuration option `MULTIPLE_SCHEMA_IMPORTS [Javadoc: net.sf.saxon.lib.FeatureKeys]` has been added to force `xs:import` to fetch the referenced schema document. By default the `xs:import` fetches the document only if no schema document for the given namespace has already been loaded. With this option in effect, the referenced schema document is loaded unless the absolute URI is the same as a schema document already loaded.

A configuration option `ASSERTIONS_CAN_SEE_COMMENTS [Javadoc: net.sf.saxon.lib.FeatureKeys]` has been added to control whether comments and processing instructions are visible to the XPath expression used to define an assertion. By default (unlike Saxon 9.3), they are not made visible.

On the `com.saxonica.Validate` command line, a new option `-stats:filename` is available: it causes statistics about the validation episode to be captured in the specified XML file.

Changes to Functions and Operators

Implemented fn:parse-json() and fn:serialize-json()

Implemented fn:path()

Changes to Saxon extensions and extensibility mechanisms

An additional flag is provided for `saxon:deep-equal()`: the `I` flag tests whether the `is-ID` and `is-IDREF` properties of two nodes match.

To allow XSLT extension instructions to be called from generated bytecode, the interface has been changed: evaluation must now be done using the standard `call` method, which is aligned with the interface for integrated extension functions. For examples, see the source code of the SQL extension classes.

Changes to application programming interfaces

In s9api, an `XQueryEvaluator [Javadoc: net.sf.saxon.s9api.XQueryEvaluator]` is now a `Destination [Javadoc: net.sf.saxon.s9api.Destination]`, so queries can participate in s9api pipelines in the same way as transformations.

In s9api, the base output URI of an `XsltTransformer [Javadoc: net.sf.saxon.s9api.XsltTransformer]` is now taken from the `Destination [Javadoc: net.sf.saxon.s9api.Destination]` if (a) no base output URI has been explicitly set, and (b) the destination is a Serializer writing to a supplied File.

A class `XdmFunctionItem` has been added to both s9api and the Saxon.NET API to represent items that are functions (as distinct from atomic values and nodes).

It is now possible to set a default collection and a collection URI resolver at the level of the `Controller [Javadoc: net.sf.saxon.Controller]` (that is, an individual transformation or query); previously it could only be set globally, in the `Configuration [Javadoc: net.sf.saxon.Configuration]`. There is now a defined constant URI (`Collection.EMPTY_COLLECTION`) which always represents an empty collection, without needing to be resolved by the URI resolver.

It is now possible to set an extension function library at the level of an individual query (via the `StaticQueryContext [Javadoc: net.sf.saxon.query.StaticQueryContext]` object), or an individudual stylesheet (via the `CompilerInfo [Javadoc: net.sf.saxon.trans.CompilerInfo]` object. Previously an extension function library could only be defined at the `Configuration [Javadoc: net.sf.saxon.Configuration]` level.

In the JAXP `XPathFactory` implementation, Saxon configuration properties (as listed in `FeatureKeys [Javadoc: net.sf.saxon.lib.FeatureKeys]`) can now be supplied to the `getFeature()` and `setFeature()` methods, provided the properties are of type boolean.

A new class `Transmitter [Javadoc: net.sf.saxon.event.Transmitter]` is available as a new kind of JAXP `Source`, recognized in all Saxon interfaces that accept a `Source`. A Transmitter writes events to a `Receiver [Javadoc: net.sf.saxon.event.Receiver]`. This is useful when the input to a streamed transformation is supplied programmatically.

A new simplified interface for defining context-free extension functions is available as part of the s9api package: `Processor.registerExtensionFunction [Javadoc: net.sf.saxon.s9api.Processor]`, where the argument implements the interface `ExtensionFunction [Javadoc: net.sf.saxon.s9api.ExtensionFunction]`. TODO: documentation, examples, tests

Changes to system programming interfaces

The `Receiver [Javadoc: net.sf.saxon.event.Receiver]` interface (widely used internally within Saxon) has changed to reduce the dependency on the `NamePool [Javadoc: net.sf.saxon.om.NamePool]`. On the `startElement()` and `attribute()` calls, the names and types of elements and attributes are now passed as object references rather than integer codes. Similarly on the `namespace()` call, the integer namespace code is replaced with a reference to a `NamespaceBinding` object.

Integer namespace codes allocated from the name pool are no longer used. They have been replaced with the `NamespaceBinding [Javadoc: net.sf.saxon.om.NamespaceBinding]` object which contains the prefix and URI as strings. The purpose of this change is to reduce the number of synchronized calls on the `NamePool [Javadoc: net.sf.saxon.om.NamePool]`, and hence to reduce contention; the performance benefit from avoiding string comparisons did not justify the overhead caused by synchronization. This change results in small changes to both the `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` and `Receiver [Javadoc: net.sf.saxon.event.Receiver]` interfaces.

A new method `getSchemaType()` is added to the `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` interface, returning the type annotation as a `SchemaType [Javadoc: net.sf.saxon.type.SchemaType]` object. The existing `getTypeAnnotation()` method which returns the same information as an integer fingerprint remains available for the time being.

The mechanism for injecting trace calls into expressions has been generalised so that an arbitrary `CodeInjector [Javadoc: net.sf.saxon.expr.parser.CodeInjector]` can be supplied. This can be selective about what kind of expression it inserts into the parse tree, and where. This gives a lot more flexibility for tools that add debugging or performance monitoring capabilities to the product. For XQuery this can be controlled at the level of the `StaticQueryContext [Javadoc: net.sf.saxon.query.StaticQueryContext]`, for XSLT using the `CompilerInfo [Javadoc: net.sf.saxon.trans.CompilerInfo]` object.

In the expression tree, the representation of path expressions and axis expressions has changed. The class `PathExpression` has disappeared; instead, the class `SlashExpression` is used, wrapped in a `DocumentSorter` if sorting into document order and elimination of duplicates is required.

A subclass of `SlashExpression`, the `SimpleSlashExpression`, is used for expressions of the form `$p/title` where the left-hand side selects a singleton and the right-hand side is an axis expression; this optimization reduces the number of context objects that need to be created, especially in XQuery where such constructs are very common.

Version 9.3 (2010-10-30)

The documentation now has an Alphabetical index.

At the time of writing, W3C has published draft specifications under the titles XQuery 1.1, XPath 2.1, and XSLT 2.1. However, W3C has also announced its intent to change the numbering, so that the final specifications will be XQuery 3.0, XPath 3.0, and XSLT 3.0. Saxon 9.3 anticipates this change by using the version number 3.0 to refer to the new set of draft specifications.

The interfaces defining callbacks that advanced applications may use to customize Saxon's behaviour were previously scattered around the various packages, some quite difficult to find. Most of them have been moved into a new package `net.sf.saxon.lib`. This package also contains default implementations of these interfaces, and classes defining constants for use in Saxon's various configuration APIs.

The current state of implementation of all standard functions is now documented here.

- Highlights
- Installation on .NET
- Command line and configuration changes
- Extensibility changes
- Extensions
- XSLT 3.0 changes
- Streaming in XSLT
- XPath 3.0 changes
- XPath 2.0 and XQuery 1.0 changes
- XQuery 3.0 and XQuery Update changes
- Functions and Operators
- XML Schema 1.0 changes
- XML Schema 1.1 changes
- Changes to the s9api API
- Saxon on .NET changes
- Serialization
- Running Saxon from Ant
- The SQL Extension
- Internal changes

Highlights

Some of the most significant features of Saxon 9.3 are:

- An installation wizard is provided on .NET
- Support for XML Schema 1.1 is almost complete.
- Support for streamed processing of XSLT is greatly extended, making transformation of very large documents much more feasible.
- For the first time, Saxon has support for parallel processing of XSLT to take advantage of multi-core CPUs.
- Many features from the draft 3.0 specifications are implemented.
- The range of data-types available is extended by the provision of immutable maps.

Installation on .NET

Saxon on .NET now comes with an installer. The downloaded file is an executable which installs the product into a user-selected directory. It also installs the relevant DLLs in the Global Assembly Cache, and creates some entries in the Windows registry, notably the path name of the directory where the software is installed. This registry entry is used when locating the license file for Saxon-PE and Saxon-EE, replacing the previous mechanism which used the `SAXON_HOME` environment variable.

License keys issued for Saxon-SA 9.1 and earlier on .NET, with the filename `saxon-license.xml`, are no longer recognized. If you have such a license key that is still valid for new software releases, please contact Saxonica for a replacement in the `saxon-license.lic` format.

Command line and configuration changes

The code for the `net.sf.saxon.Transform` and `net.sf.saxon.Query` command line interfaces has been refactored. Impact: (a) some legacy options are no longer supported (for example, `-ds`, `-dt`, and the ability to separate keyword from value using a space rather than a colon, e.g. "`-o output.html`"). (b) `-opt:?` now requests help on that particular option, (c) options that were previously freestanding now accept "on" as a value, e.g. `-t` can be written `-t:on`.

A new option `-init:initializer` is available on all command line interfaces. The value is the name of a user-supplied class that implements the interface `net.sf.saxon.lib.Initializer [Javadoc: net.sf.saxon.lib.Initializer]`; this initializer will be called during the initialization process, and may be used to set any options required on the `Configuration [Javadoc: net.sf.saxon.Configuration]` programmatically. It is particularly useful for such tasks as registering extension functions, collations, or external object models, especially in Saxon-HE where the option does not exist to do this via a configuration file. Saxon only calls the initializer when running from the command line, but of course the same code may be invoked to perform initialization when running user application code.

On the `Transform` command line interface, the `-traceout` option now governs the destination of trace output from the standard `TraceListener [Javadoc: net.sf.saxon.lib.TraceListener]` (-T option) as well as from the `trace()` function. The `-TP` option (for timing profile information) is extended so a filename can be specified: `-TP:filename`.

On the `Transform` command line interface, the option `-xsltversion:2.0` or `-xsltversion:3.0` indicates whether the XSLT processor should implement the XSLT 2.0 specification or the XSLT 3.0 (also known as 2.1) specification. The default value `-xsltversion:0.0` indicates that this decision should be made based on the `version` attribute of the `xsl:stylesheet` element. Similar options to set the XSLT processor version are available in the `XsltCompiler [Javadoc: net.sf.saxon.s9api.XsltCompiler]` class (s9api on Java, Saxon.Api on .NET), and via new options in `FeatureKeys [Javadoc: net.sf.saxon.lib.FeatureKeys]` and in the configuration file.

The `com.saxonica.Validate` interface accepts some additional options as a result of these changes: `-dtd`, `-ext`, `-opt`, `-y`. The option `-xsdversion` was already accepted, but not documented.

The `com.saxonica.Validate` interface has a new option `-stats:filename` which produces an output document showing which schema components were used during the validation, and how often. The output is in XML, allowing further processing to produce profiles and coverage reports for the schema. (There are corresponding internal APIs that allow the same effect when validation is invoked from an application, but they are not currently exposed through s9api.)

The command `com.saxonica.CompileStylesheet` now uses the `-key:value` argument style throughout. It now accepts a `-config:filename` argument. The compiled stylesheet output may be specified using `-csout:filename`

All commands now accept `--F:value` where F is the name of a string defined in FeatureKeys (the part after `"http://saxon.sf.net/feature/"`), and value is the string value of the feature; or `--F` as a synonym for `--F:true`.

Most places in the code that previously wrote to `System.err`, or that used `System.err` as a default destination, now default instead to using `Configuration.getStandardErrorOutput() [Javadoc: net.sf.saxon.Configuration#getStandardErrorOutput]`. This can be set to a different destination by calling `Configuration.setStandardErrorOutput() [Javadoc: net.sf.saxon.Configuration#setStandardErrorOutput]` (which expects a `PrintStream`), or by setting the write-only configuration property `FeatureKeys.STANDARD_ERROR_OUTPUT_FILE [Javadoc: net.sf.saxon.lib.FeatureKeys#STANDARD_ERROR_OUTPUT_FILE]` (which expects a filename, which will be appended to). In the configuration file the corresponding setting is `global/@standardErrorOutputFile`. Note that this only redirects Saxon output; other application output written to `System.err` is unaffected. Output written directly by command-line interfaces such as `net.sf.saxon.Transform` is unaffected. Examples of affected output are the default destination of the `ErrorListener`; the default destination for `TraceListener` and `xsl:message` output; the default destination for optimizer tracing and "explain" output.

Extensibility changes

Reflexive extension functions must now use the "strict" URI format (for example `"java:java.util.Date"`) rather than the "liberal" format (which allows for example `"http://my.com/extensions/java.util.Date"`), unless the configuration property `ALLOW_OLD_JAVA_URI_FORMAT [Javadoc: net.sf.saxon.lib.FeatureKeys#ALLOW_OLD_JAVA_URI_FORMAT]` is set. This change was documented for 9.2 but not implemented (the default value for the flag was to allow the old URI syntax). Note that because dynamic extension functions require at least Saxon-PE, this flag is not recognized in Saxon-HE.

Extensions

A new extension is introduced to allow maps to be maintained. A map is a dictionary data structure that maps atomic values to arbitrary XDM sequences. The map can be built incrementally, but it is immutable (as with operations on sequences, adding an entry to a map creates a new map, leaving the existing map unchanged). The implementation optimizes this behind the scenes to avoid rebuilding the map from scratch each time an entry is added, and to allow memory to be shared. For details see The map extension.

This facility is particularly useful in streaming transformations in conjunction with `xsl:iterate`, as it allows arbitrary information to be "remembered" for later use during the course of a streaming pass through the source document.

A new extension function `saxon:current-mode-name()` is introduced for use in XSLT, returning the name of the current mode as a QName.

In XSLT, the extension attribute `saxon:threads` is introduced on `xsl:for-each` to allow items in the input sequence to be processed in parallel, using the specified number of threads. Multithreading will be used only when requested; it can also be suppressed at the Configuration level, and it is disabled when a stylesheet is compiled with tracing enabled, because the trace output would otherwise be unintelligible.

XSLT 3.0 changes

(Also affects XSLT 2.0): Saxon no longer detects or reports the recoverable error XTRE0270 (conflicting definitions of `xsl:strip-space` and `xsl:preserve-space`). Instead it always takes the optional recovery action, which is to use whichever declaration comes last. Previously this was the behaviour when applying stripping to an existing tree, but not when using a stripping filter during tree construction. The change is made in the interests of simplifying and speeding up the code (matching of whitespace stripping rules no longer shares the same code as template rule matching).

(Also affects XSLT 2.0): The call `system-property("xsl:is-schema-aware")` now returns true or false depending on whether the particular stylesheet is schema-aware (which is true if it uses `xsl:import-schema` or if schema-awareness was selected via an API or command line option). Previously it returned true if the stylesheet was processed using the schema-aware version of the Saxon product, regardless of configuration settings.

XSLT 3.0 features are available only in Saxon-EE or (in some cases) Saxon-PE, and they need to be enabled explicitly. From the command line this can be done using the option `xsltversion:3.0`; the default is to take the version from the `xsl:stylesheet` element, so that XSLT 3.0 features are enabled if and only if the stylesheet itself specifies `version="3.0"`.

XPath 3.0 constructs such as higher-order functions (to the extent they are implemented) are automatically available when XSLT 3.0 is enabled.

The `xsl:evaluate` instruction is implemented.

The `xsl:copy` instruction now has an optional `select` attribute, defaulting to `select="."`.

The `saxon:iterate`, `saxon:break`, `saxon:continue`, and `saxon:finally` instructions are renamed `xsl:iterate`, `xsl:break`, `xsl:next-iteration`, and `xsl:on-completion`, and are available only if XSLT 3.0 support is enabled. The `xsl:break` instruction is now allowed to take a sequence constructor as its content.

The `saxon:try` and `saxon:catch` elements are renamed `xsl:try` and `xsl:catch`, and are available only if XSLT 3.0 support is enabled.

The syntax of match patterns has been extended, to include the forms `$x`, `$x//a/b/c`, `doc(X)`, `doc(X)//a/b/c`, `element-with-id(X)`, `element-with-id(X)//a/b/c`, as well as the two-argument form of `id()` and the three-argument form of `key()`. The keyword `"union"` is allowed as an alternative to the `"|"` operator.

The `unparsed-text-lines()` function is implemented.

The `copy-of()` and `snapshot()` functions are implemented. (There is a restriction in `snapshot()`, in that it does not yet handle attribute or namespace nodes.

The `xsl:analyze-string` instruction accepts the enhancements to regular expressions and flags defined in XPath 3.0. It also now accepts an empty sequence as the value of the `select` attribute, treating it in the same way as a zero-length string.

Streaming in XSLT

Many more constructs are now acceptable in streaming templates. For full details see Streaming Templates.

Explicit support for streaming is available for the functions `data()`, `avg()`, `count()`, `distinct-values()`, `empty()`, `exists()`, `min()`, `max()`, `string()`, `string-join()`, `sum()`; for the operators "," and the `(= | != | <= | < | >= | >)` family, and for the instructions `xsl:attribute`, `xsl:apply-imports`, `xsl:apply-templates`, `xsl:choose`, `xsl:comment`, `xsl:copy`, `xsl:copy-of`, `xsl:document`, `xsl:element`, `xsl:for-each`, `xsl:iterate`, `xsl:result-document`. Implicit support is available for all functions and operators that process singleton items, and for many functions and instructions that operate on sequences - such as `subsequence()` and `xsl:for-each-group` - with the caveat that their input is buffered in memory.

XPath 3.0 changes

Support for XPath 3.0 (previously known as XPath 2.1) is provided. To enable this, call `setLanguageVersion("2.1")` on the s9api `XPathCompiler [Javadoc: net.sf.saxon.s9api.XPathCompiler]`, or equivalent calls on the `net.sf.saxon.sxpath.XPathEvaluator [Javadoc: net.sf.saxon.sxpath.XPathEvaluator]`. Within XSLT, XPath 3.0 syntax is supported if the version attribute on the principal stylesheet module is set to "3.0". From JAXP XPath interfaces, support for XPath 3.0 is supported by casting the `XPath` object to `net.sf.saxon.xpath.XPathEvaluator [Javadoc: net.sf.saxon.xpath.XPathEvaluator]` and calling `xpath.getStaticContext().setXPathLanguageLevel(new DecimalValue("2.1"))`.

The following features are available when XPath 3.0 is enabled:

- The let expression (`let $x := expr, $y := expr return expr`).
- The new context-independent QName syntax (`"uri":local`, also `"uri":*`)
- Facilities associated with higher-order functions: function literals, partial function application, dynamic function invocation, inline functions, plus associated functions such as `map()`, `filter()`, `fold-left()`, and `fold-right()`.
- New functions and operators: see Changes to functions and operators
- Enhancements to regular expressions (non-capturing groups `(?:xxxx)`) and flags (the `q` flag)

The s9api API, together with lower-level APIs for running XPath (including the `StaticContext [Javadoc: net.sf.saxon.expr.StaticContext]` interface) have been enhanced to allow the required type of the context item to be supplied as a property of the static context.

XPath 2.0 and XQuery 1.0 changes

Erratum XQ.E34 is implemented. This affects what can appear after a "/" at the start of an expression. Certain constructs that can appear at the start of a relative path expression are now recognized, when they were previously rejected as errors: for example `/element{a}{b}` or `/ ` or `/unordered{x}`. Some other constructs that were previously accepted are now rejected, for example `/ instance of document-node()` (this must now be written `(/) instance of document-node()`).

In XQuery 1.0 and XQuery 1.1, the pragmas `saxon:stream` and `saxon:validate-type` are now ignored (with a warning) if running Saxon-HE or Saxon-PE. Previously they caused a static error.

XQuery 3.0 and XQuery Update changes

In the absence of an `xquery version` declaration in the query prolog, Saxon now permits XQuery 3.0 syntax if it is enabled from the command line or API. An `xquery version` declaration that specifies version="3.0" is accepted only if XQuery 3.0 has been enabled from the command line or API.

The syntax for production `AnyFunctionItem` is now `function(*)` rather than `function()`. For backwards compatibility with Saxon 9.2, the older syntax continues to be supported for the time being.

The new `switch` expression is implemented.

The custom syntax for partial function application (for example `concat("[", ?, ", ", ?, "]")` is implemented. (For the time being, the `partial-apply()` function remains available as well.)

The syntax for "outer for" has been changed (W3C bug 6927). In place of `"outer for $x in E"`, write `"for $x allowing empty in E"`.

In a FLOWR expression, the `where` clause can now be repeated. (The construct `where X where Y` is simply another way of writing `where X and Y`.)

The new context-independent QName syntax `"uri":local` is recognized if XQuery 3.0 is enabled.

The qualifiers `deterministic`, `nondeterministic`, `public`, and `private` are now recognized on function declarations. However they currently have no effect.

In XQuery Update, the tree constructed by the `copy-modify` (or "transform") expression will now automatically be a mutable tree, regardless of the default tree model for the `Configuration [Javadoc: net.sf.saxon.Configuration]`.

On the `net.sf.saxon.Query` command line interface, the `-traceout` option now governs the destination of trace output from the standard `TraceListener [Javadoc: net.sf.saxon.lib.TraceListener]` (-T option) as well as from the `trace()` function. The `-TP` option (for timing profile information) is extended so a filename can be specified: `-TP:filename`.

Functions and Operators

The current state of implementation of all standard functions is now documented here.

The new function `fn:analyze-string()` is implemented.

The higher-order functions `map()`, `filter()`, `fold-left()`, `fold-right()`, and `map-pairs()` are implemented.

The functions `head()` and `tail()` are implemented.

The functions `pi()`, `sqrt()`, `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, and `atan()` are implemented - note that these are in the namespace `http://www.w3.org/2005/xpath-functions/math`. Unlike the other new functions, these are available whenever Saxon-PE or Saxon-EE are in use, regardless of whether XQuery 3.0 or XPath 3.0 are enabled - this is conformant because they are in a namespace to which the 1.0/2.0 specifications attach no restrictions.

The single-argument version of `string-join()` is implemented.

The two-argument version of `round()` is implemented.

The zero-argument forms of `data()`, `node-name()`, and `document-uri()` are implemented.

The new `format-integer()` function is implemented.

The new regular expression flag "q" is recognized. This causes all characters in the regex to be treated as ordinary characters, for example "." will match a single period, rather than matching any character. This is only recognized when XPath 3.0 or XQuery 3.0 is enabled.

Non-capturing groups are recognized, with the syntax `(?:aaaa)` where aaaa is any regular expression.

XML Schema 1.0 changes

There is a rule in both XSD 1.0 and XSD 1.1 that components redefined using `xs:redefine` must have been defined immediately within the schema document referenced by the `xs:redefine` element. Previous releases of Saxon did not enforce this rule, instead allowing the redefined component to be either in the redefined schema document or in an indirectly included schema document. Saxon 9.3 checks the rule, but reports any violations in the form of a warning rather than a hard error, to avoid invalidating schemas that previously worked.

Elements and attributes of type `xs:ENTITY` or `xs:ENTITIES` are now checked against the list of unparsed entities declared in the document. (This check is not performed during validation invoked by XSLT or XQuery validation, but it is performed during standalone schema validation.)

There is a new switch `FeatureKeys.VALIDATION_COMMENTS [Javadoc: net.sf.saxon.lib.FeatureKeys#VALIDATION_COMMENTS]` to control whether comments are written into an instance document when validation fails. Previously this always happened if the switch `FeatureKeys.VALIDATION_WARNINGS [Javadoc: net.sf.saxon.lib.FeatureKeys#VALIDATION_WARNINGS]` was on; it can now be controlled separately, and the default is off. A corresponding switch has been added to the configuration file. The command-line option `-outval:recover` sets both switches on. Internally the switches are held in the `ParseOptions [Javadoc: net.sf.saxon.lib.ParseOptions]` object and can thus be set differently for different validation episodes. For the `FeatureKeys.VALIDATION_WARNINGS [Javadoc: net.sf.saxon.lib.FeatureKeys#VALIDATION_WARNINGS]` switch, the two flags previously held in the default `ParseOptions` and in local `Configuration [Javadoc: net.sf.saxon.Configuration]` data have been combined into one, preventing the use of inconsistent settings.

Static type checking is now applied to the XPath expressions in the `xs:field` and `xs:selector` elements of identity constraints. Warnings are reported if the paths select nothing, or if the field expression potentially selects multiple nodes, an empty sequence (in the case of `xs:key` only), or nodes whose type is not a simple type or a complex type with simple content.

XML Schema 1.1 changes

The `xs:override` declaration is implemented. Pending resolution of specification bug 9661, a declaration that appears within `xs:override` and does not override anything in the overridden schema is included in the schema; the other possible interpretation of the specification is that it should be ignored. Restrictions: circular `xs:override` chains are not allowed, and there has been no testing of the interaction of `xs:override` with `xs:redefine`.

Saxon recognizes the attribute `saxon:extensions` on the `xs:schema` element: the value is a space-separated list of strings giving the names of extensions that are enabled in this schema document. If the extension `id-xpath-syntax` is enabled, then the XPath syntax permitted in the `xs:selector` and `xs:field` elements of uniqueness, key and keyref constraints is extended to allow any . For example, this allows the use of attribute-based predicates, such as `xpath="employee[@location='us']"`

The new primitive type `xs:precisionDecimal` is implemented - at least, to the extent required for schema validation. It is not recommended yet to use this type in documents subject to query and transformation, or to use it for free-standing atomic values, since (a) there is no specification for how operations such as arithmetic and comparison should behave in XPath, (b) there are gaps in the implementation in this area, (c) very limited testing has been done, and (d) the type could yet be dropped (it is marked as a "feature at risk", and there is some political objection to it).

Inheritable attributes are implemented (for use in Conditional Type Assignment only: the only effect of declaring an attribute to be inheritable is that it can be referenced in the XPath condition of the `xs:alternative` element).

An element may now have more than one ID attribute. Several ID attributes or children of an element may have the same value. ID and IDREF values appearing within a list, or as a member type of a union, are now recognized. (Most of these changes have been applied also to XSD 1.0, with the exception that multiple ID attributes are not allowed on an element. Saxon 9.2 and earlier releases correctly allowed an element to have more than one ID-valued child element, but incorrectly reported an error if more than one ID-valued child of the same parent element had the same ID value. The rules for list types and union types with an ID or IDREF member are unclear in XSD 1.0, so for simplicity, the XSD 1.1 rules have been implemented unconditionally.

It is now an error for the outermost element of the document (the validation root) to be of type `xs:ID`. (The rules in the W3C spec have always implied this, though many cases in the W3C test suite consider this to be valid. The Working Group has confirmed in its decision on bug #9922 that this is intended to be invalid.)

In element wildcards, `notQName="##definedSibling"` is implemented.

An `xs:all` group may now contain an `xs:group` element to refer to a named model group declaration, provided the named model group definition in turn contains an `xs:all` group, and that `minOccurs` = `maxOccurs` = `1`.

The way that `xs:anyURI` values are checked (to see if they are valid URIs) has changed. In accordance with the W3C specifications, there is now no checking at all when XSD 1.1 is selected at the `Configuration [Javadoc: net.sf.saxon.Configuration]` level - any string may be used. When 1.0 is selected, strings are checked to be valid URIs (as defined by the `java.net.URI` class) only when performing schema validation, or explicit casting from string to `xs:anyURI`. There is no longer any check performed by methods such as `namespace-uri()` or `namespace-uri-from-QName`, regardless whether XSD 1.0 or XSD 1.1 is selected in the Configuration. It is also possible to configure the checking that is performed to use a user-supplied `URIChecker`, for example one that performs stricter or more liberal checking.

Changes to the s9api API

The `XsltTransformer [Javadoc: net.sf.saxon.s9api.XsltTransformer]` interface has two new methods, `setURIResolver()` and `getURIResolver()`. These define the `URIResolver` used for resolving calls to `doc()` and `document()`.

The `DocumentBuilder [Javadoc: net.sf.saxon.s9api.DocumentBuilder]` class has a new method `newBuildingStreamWriter()`. This gives access to a class that allows a Saxon document tree to be built programmatically by writing Stax `XMLStreamWriter` events. This is a lot easier than the previous alternatives, of generating SAX events or Saxon `Receiver [Javadoc: net.sf.saxon.event.Receiver]` events. This mechanism is supported by an underlying class `StreamWriterToReciever [Javadoc: net.sf.saxon.event.StreamWriterToReceiver]` which converts Stax `XMLStreamWriter` events into Saxon `Receiver` events. this class does not cache the `NamePool`: it may therefore generate a high level of contention if used in a multithreading environment.

Similarly, the `DocumentBuilder [Javadoc: net.sf.saxon.s9api.DocumentBuilder]` class has a new method `newContentHandler()`. This returns a SAX `ContentHandler` to which events may be sent to build a tree programmatically. Although slightly less convenient than the `XMLStreamWriter` interface, this is useful for the many cases where an existing application already generates SAX events.

The specification of the `XdmDestination [Javadoc: net.sf.saxon.s9api.XdmDestination]` class has been clarified to state that the event stream written to the `XdmDestination` must constitute either a single tree that is rooted at a document or element node, or an empty sequence; and the implementation has been changed to enforce this. This means that when this class is used for the destination of an XQuery query, an exception is thrown if the query returns atomic values, nodes other than document or element nodes, or sequences of multiple nodes. Previously the effect in such cases was poorly specified and could lead to internal exceptions with poor diagnostics.

The `XdmNode [Javadoc: net.sf.saxon.s9api.XdmNode]` class now has a public constructor allowing a `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` to be wrapped.

The `Processor [Javadoc: net.sf.saxon.s9api.Processor]` class now has a public constructor allowing a `Configuration [Javadoc: net.sf.saxon.Configuration]` to be wrapped.

The `Processor [Javadoc: net.sf.saxon.s9api.Processor]` class now has a number of convenience factory methods allowing a `Serializer [Javadoc: net.sf.saxon.s9api.Serializer]` to be constructed. There is also a method `setProcessor()` that allows the `Serializer` to retain a connection with a `Processor`, and hence a configuration. This enables new methods on the `Serializer` to be simplified, avoiding the need to supply a `Processor` (or Configuration) on the method call. Eventually the free-standing constructors on `Serializer` may be deprecated.

The `Serializer [Javadoc: net.sf.saxon.s9api.Serializer]` class has two new convenience methods, allowing serialization of an `XdmNode [Javadoc: net.sf.saxon.s9api.XdmNode]` to an arbitrary `Destination [Javadoc: net.sf.saxon.s9api.Destination]`, or more simply to a Java string.

The `Serializer [Javadoc: net.sf.saxon.s9api.Serializer]` class has a new method `getXMLStreamWriter()`, allowing an `XMLStreamWriter` to be constructed as a front-end to this `Serializer`. This is a very convenient way of generating serialized XML output from a Java application: for an example of its use, see the XSLT test suite driver in the samples directory.

The `XPathCompiler [Javadoc: net.sf.saxon.s9api.XPathCompiler]` object now has the ability to maintain a cache of compiled XPath expressions. If this feature is enabled, any attempt to compile an expression first causes a lookup in the cache to see whether the same expression has already been compiled. The cache is cleared if any changes to the static context are made (for example, changing the namespace declarations in force).

The `XPathCompiler [Javadoc: net.sf.saxon.s9api.XPathCompiler]` object has two new convenience methods, `evaluate()` and `evaluateSingle()`, allowing an expression to be compiled and executed with a single call. This works especially well when the compiler is also caching compiled expressions.

Running a pipeline of XSLT transformations by using each `XsltTransformer [Javadoc: net.sf.saxon.s9api.XsltTransformer]` as the `Destination [Javadoc: net.sf.saxon.s9api.Destination]` of the previous one is now more efficient; the code has been changed so that the second transformation does not start until the stack and heap for the first one have been released. This has entailed a minor change to the `Destination` interface: it now has a `close()` method; and it also means that the `XsltTransformer` is not serially reusable. You should create a new `XsltTransformer` for each transformation (while reusing the `XsltExecutable [Javadoc: net.sf.saxon.s9api.XsltExecutable]`, of course).

The `XQueryEvaluator [Javadoc: net.sf.saxon.s9api.XQueryEvaluator]` has a new method `callFunction()` that allows any user-declared function within the compiled query to be called directly from the Java application. This in effect enables the creation of a query library containing multiple functions that can be invoked from the calling application. Note that to compile a query, it must still have a "main query" to satisfy the syntax rules of the XQuery language.

Saxon on .NET changes

The `XsltExecutable` object has a new `Explain()` method that gives a diagnostic representation of the compiled code (as an XML document).

The `Equals()` and `GetHashCode()` methods on `XdmNode` are now defined so that two `XdmNode` instances are equal if and only if they represent the same node (that is, they reflect the XPath "is" operator).

The `XPathCompiler` object now has the ability to maintain a cache of compiled XPath expressions. If this feature is enabled, any attempt to compile an expression first causes a lookup in the cache to see whether the same expression has already been compiled. The cache is cleared if any changes to the static context are made (for example, changing the namespace declarations in force).

The `XPathCompiler` object has two new convenience methods, `Evaluate()` and `EvaluateSingle()`, allowing an expression to be compiled and executed with a single call. This works especially well when the compiler is also caching compiled expressions.

The error handling in the XPath API has been improved so that static and dynamic errors occurring during XPath evaluation now result in a `Saxon.Api.StaticError` or `Saxon.Api.DynamicError` being thrown, rather than exposing the underlying Java exceptions.

The `XQueryEvaluator` has a new method `CallFunction` that allows any user-declared function within the compiled query to be called directly from the Java application. This in effect enables the creation of a query library containing multiple functions that can be invoked from the calling application. Note that to compile a query, it must still have a "main query" to satisfy the syntax rules of the XQuery language.

There is a new overload of `DocumentBuilder.Build()` that allows the XML document to be supplied from a `TextReader` rather than a stream (including a `StringReader`, which makes it easier to build from XML held as a string literal).

Serialization

Two new serialization methods are introduced, saxon:base64Binary and saxon:hexBinary. These are useful when writing binary output files such as images. They can be used in conjunction with the `xsl:result-document` instruction, using a result tree whose text nodes contain, in base64 or hexBinary format, the binary data to be output. These serialization methods require Saxon-PE or higher.

The existing serialization method `saxon:xquery` is now available only in Saxon-PE or higher.

The `suppress-indentation` serialization parameter (previously available as an extension in the Saxon namespace) is now implemented.

A new serialization property `saxon:line-length` is introduced. Its value is an integer, with default value 80. With both the HTML and XML output methods, attributes are output on a new line if they would otherwise extend beyond this column position. With the HTML output method, furthermore, text lines are split at this line length when possible. In previous releases, the HTML output method attempted to split lines that exceeded 120 characters in length.

In the XQJ interface, serialization properties are now validated. The XQJ spec is not very clear about how all parameters should be supplied: follow the conventions of JAXP interfaces such as `Transformer.setOutputProperty()`. In particular, the values for boolean properties should be set to the string "yes" or "no".

Running Saxon from Ant

The custom Ant task for Saxon is no longer supported. Instead, all the required functionality is available through the standard Ant `xslt` task. In particular, it is now possible (with Saxon-PE and

Saxon-EE) to specify the name of a Saxon configuration file as an attribute child of the `factory` element, which provides full control over the Saxon configuration used to run the transformation.

A new class `com.saxonica.jaxp.ValidatingReader [Javadoc: com.saxonica.jaxp.ValidatingReader]` has been introduced. This implements the SAX2 `XMLReader` interface and accepts a number of Apache-defined properties, allowing it to be used as a plug-in replacement for Xerces to support the Ant `xmlvalidate` and `schemavalidate` tasks, using the Saxon schema processor including the option of using XSD 1.1 for validation. For details see Running validation from Ant.

The SQL Extension

The `sql:connect` instruction now supports an attribute `auto-commit="{yes|no}"` to control this property of the JDBC connection.

A new instruction `sql:execute` is available to execute an arbitrary SQL statement (returning no result). It takes two attributes, `connection` and `statement`. For details see sql:execute

Internal changes

There has been some reorganization of the structure of classes and packages. Many of the interfaces that are intended for applications to implement (such as `CollectionURIResolver [Javadoc: net.sf.saxon.lib.CollectionURIResolver]`), and also classes defining constants for use as parameters in an API (such as `FeatureKeys [Javadoc: net.sf.saxon.lib.FeatureKeys]` and `SaxonOutputKeys [Javadoc: net.sf.saxon.lib.SaxonOutputKeys]`) have been moved to the new package `net.sf.saxon.lib`. Classes and packages that are purely for internal use have in same cases been buried in a more deeply nested package hierarchy, to make it easier to find the classes that are of interest to applications. For example, implementations of `SequenceIterator [Javadoc: net.sf.saxon.om.SequenceIterator]` have been pushed down into `net.sf.saxon.om.iter`. Classes concerned with serialization have been moved out of the `event` package into a new `serialize` package.

There is no longer a separate parser for XSLT Patterns; instead, patterns are parsed as XPath expressions, and the resulting expression tree is then converted to a pattern object.

In the representation of a stylesheet tree, there is now a distinction between a stylesheet document (`XSLStylesheet [Javadoc: net.sf.saxon.style.XSLStylesheet]`), and a stylesheet module (`StylesheetModule [Javadoc: net.sf.saxon.style.StylesheetModule]`). This caters for the case where the same stylesheet document is imported several times with different import precedence. The new structure allows several `StylesheetModule`s therefore to share the same source code, but with different precedence. This also paves the way to allowing stylesheet modules eventually to be parsed once and shared between different stylesheets (but this is not easy, because references such as variable references and function calls may be resolved differently in the different cases).

The class `PreparedStyleSheet [Javadoc: net.sf.saxon.PreparedStyleSheet]` now subclasses `Executable [Javadoc: net.sf.saxon.expr.instruct.Executable]`, and duplication of functionality between these two classes has been eliminated.

Dropped the methods (deprecated since Saxon 8.9) `build() [Javadoc: net.sf.saxon.sxpath.XPathEvaluator#build]` and `setStripSpace() [Javadoc: net.sf.saxon.sxpath.XPathEvaluator#build]` in `net.sf.saxon.sxpath.XPathEvaluator [Javadoc: net.sf.saxon.sxpath.XPathEvaluator]`.

In the `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` interface, the `copy()` method has been changed to take a bit-significant `copyOptions` argument replacing the previous `whichNamespaces` and `typeAnnotations` arguments. (It also allows an additional option for requesting that the copy should be mutable.)

There is a new optimization for the expression `(A intersect B)`; if the first operand (say) is a singleton, the operator is rewritten as `singleton-intersect` and the run-time evaluation does a serial search of the second operand to see if the first item is present. This avoids an unnecessary sort of the second operand.

There have been improvements to schema-aware type checking for the descendant axis, in particular (a) for expressions starting at the document node, in cases where the type of the document node is known in the form `document-node(schema-element(X))`, and (b) where the structure is recursive (or more generally, where the descendant element can be reached by different routes, but has the same type in each case). The result is that misspelt names appearing in a path after a "//" operator are more likely to be detected and reported; and in some cases more efficient code will be generated to handle the atomized result of the path expression.

When parameters are passed to a stylesheet or query, Saxon generally applies the function conversion rules to the supplied values. For example, if the required type is `xs:double` then it is acceptable to supply an integer. Two changes have been made in this area. Firstly, the existing code was too liberal in the case of numeric parameters: if the supplied value and the required type were both numeric, it applied the casting rules rather than only allowing numeric promotion. Secondly, the XQJ specification requires the supplied value to match the required type without conversion or promotion, so Saxon now provides an option in the XQuery interface to suppress conversion, and this option is always set in the case of XQJ applications.

In the JAXP XPath API, Saxon's implementation of `XPathFactory` now automatically registers JDOM, DOM4J, and XOM as supported external object models if a Saxon-PE or Saxon-EE configuration is in use.

Version 9.2 (2009-08-05)

Note that Saxon 9.2 requires J2SE 5 or later. It no longer works with JDK 1.4

Note in particular the packaging changes. Saxon now comes in three editions: Home Edition, Professional Edition, and Enterprise Edition.

- Highlights
- Installation and Licensing
- S9API interface
- Saxon on .NET
- XSLT
- XQuery 1.0
- XQuery Updates
- XQuery 1.1
- XML Schema
- Streaming

- Functions and Operators
- XML Parsing and Serialization
- External Object Models
- Extensibility
- Extensions
- Optimizations
- Internals

Highlights

This page lists some of the most important new features introduced in Saxon 9.2.

- Saxon 9.2 requires JDK 1.5
- Saxon 9.2 comes in three editions: Home, Professional, and Enterprise, replacing the previous split between Basic and Schema-Aware.
    - Home Edition (HE) includes most of what was in Saxon-B, with the exception of Saxon extensions and extensibility features
    - Professional Edition (PE) includes everything that was in Saxon-B, plus some additional features previously only available in Saxon-SA, such as XQuery 1.1 support and higher-order functions
    - Enteprise Edition (EE) is the successor to Saxon-SA, and includes all the capability of Saxon-SA plus new features introduced in Saxon 9.2.
- Configuration information can now be specified in an optional configuration file
- A new mechanism for creating "integrated extension functions" is defined, removing the reliance on Java reflection and the dependency on ad-hoc rules for conversions between XPath types and Java classes
- Separate compilation of query modules in now possible (Saxon-EE only)
- Support for XQuery Updates is aligned with the Candidate Recommendation
- Selected features from the draft XQuery 1.1 Recommendation are implemented, notably Higher Order Functions
- A number of Saxon extension functions have been reimplemented to use higher order functions, leading to a slight change in interface
- More features from XML Schema 1.1 have been implemented, including "open content", and the ability for an element to belong to multiple substitution groups.
- More capabilities are available for running transformations in streaming mode, in particular, it is now possible to traverse a streamed document using recursive template rules provided that the template rules are sufficiently simple.

Installation and Licensing

JDK dependency

Saxon on the Java platform now requires J2SE 5 (often called JDK 1.5) or later. It no longer works with JDK 1.4.

Repackaging

Saxon is now available in three editions: Home (HE), Professional (PE), and Enterprise (EE). The Home Edition is open-source, available free of charge, and runs without a license file. The Professional and Enterprise editions both require a license key obtainable from Saxonica.

The Saxon-SA product has been renamed Saxon-EE (for "Enterprise Edition") to reflect the fact that it contains many added-value features beyond schema-awareness: for example, streaming, XQuery Updates, compilation of queries to Java code, separate compilation of query libraries, and an advanced query optimizer.

The JAR and DLL files are renamed accordingly (for example saxon9ee.jar, saxon9ee.dll).

The `Configuration` class for Saxon-EE (previously `com.saxonica.validate.SchemaAwareConfiguration`) is now renamed `com.saxonica.config.EnterpriseConfiguration`. This change reflects that fact that use of this Configuration enables all features that are exclusive to Saxon-EE, of which schema-awareness is only one. The Configuration class for Saxon-PE is named `com.saxonica.config.ProfessionalConfiguration`, while that for Saxon-HE is simply `net.sf.saxon.Configuration`.

The JAXP factory classes for Saxon-EE (previously `com.saxonica.SchemaAwareTransformerFactory` and `SchemaAwareXPathFactory`) are renamed `com.saxonica.config.EnterpriseTransformerFactory` and `com.saxonica.config.EnterpriseXPathFactory`.

Similarly, Saxon-PE (professional edition) offers `com.saxonica.config.ProfessionalTransformerFactory` and `com.saxonica.config.ProfessionalXPathFactory`.

Some features that were previously available in the open-source product Saxon-B are not included in the Saxon-HE (home edition) product build. These features are all optional extras, in the sense that they are not required for conformance to the W3C or Java API standards. The relevant features are:

- The traditional mechanism for binding extension functions by reflexion (searching the classpath for matching names) is now available only in Saxon-PE and Saxon-EE. In Saxon-HE, the only way to define extension functions is the new mechanism of "integrated extension functions", which need to be explicitly registered with the Configuration.
- All extension functions in the Saxon namespace `http://saxon.sf.net/` now require at least Saxon-PE.
- All EXSLT extension functions now require at least Saxon-PE.
- The XSLT extension instructions `saxon:assign`, `saxon:call-template`, `saxon:doctype`, `saxon:entity-ref`, `saxon:import-query`, `saxon:script`, and `saxon:while` now require Saxon-PE.
- The SQL extension: that is, the extension instructions `sql:connect`, etc, is not available in Saxon-HE. The code for this extension remains available as an open-source plug-in for use with Saxon-PE or Saxon-EE.
- External object model support for JDOM, DOM4J, and XOM is not available for Saxon-HE "out of the box". However, the source code for these extensions remains available in open-source form, and can be used by compiling it and registering it with the Configuration.
- Memo functions, previously available in Saxon-B, now require Saxon-PE or Saxon-EE. Any request in a stylesheet or query to create a memo function is ignored under Saxon-HE with a warning.

- The ability to "compile" stylesheets (that is, to create a Java serialization of the internal representation) is now available only in Saxon-EE. The command to achieve this is renamed `com.saxonica.CompileStylesheet`. Such stylesheets can be executed under Saxon-HE, but because they have always required a dedicated `Configuration`, this is now packaged for use only from the command line (if you need to run it from a different application, you can copy the relevant code from the `net.sf.saxon.Transform` source, and you must take responsibility for ensuring that the application runs in its own `Configuration`.)

- Localization support for a number of languages other than English in `xsl:number` and in the `format-dateTime()` family of functions is provided by means of modules in package `net.sf.saxon.option.local`. These modules are included in the JAR files for Saxon-PE and Saxon-EE; if required for Saxon-HE they can be compiled from source code. In all cases they are no longer picked up automatically by virtue of class naming conventions, instead they must be explicitly registered with the Configuration either by using the method `setLocalizationClass()` or, in Saxon-PE and Saxon-EE, via the configuration file.

Some features that were previously available only in Saxon-SA are now available in Saxon-PE (without open source code). These include:

- The PTree persistent XML tree format

- A number of extension functions: saxon:analyze-string(), saxon:call(), saxon:find(), saxon:index(), saxon:for-each-group(), saxon:format-dateTime(), saxon:format-number(), saxon:function(), saxon:generate-id(), saxon:try()

- Support for a subset of the new facilities in XQuery 1.1 (grouping, format-number(), format-date(), etc)

Schema-awareness is now a property of a compiled query, stylesheet, or XPath expression. By default, these executables are schema-aware if they contain an import schema declaration in the source code, or if a schema was imported programmatically into the static context for the compilation. If the executable is not schema-aware, then all the data supplied at run-time must be untyped. The reason for this is that there is a considerable performance penalty if it is not known statically whether data will be typed or untyped; therefore, code that is not explicitly declared to be schema-aware is now compiled to handle untyped data only. (This allows the type annotations `xs:untyped`, `xs:anyType`, and `xs:untypedAtomic`. Of these, `xs:anyType` will appear only in nodes constructed from within a query, and only when construction mode is "preserve".)

For XSLT a transformation can be set to be schema-aware, even if it does not import a schema, by setting the Configuration property `FeatureKeys.XSLT_SCHEMA_AWARE` to true. For XQuery, the same effect can be achieved by setting the Configuration property `FeatureKeys.XQUERY_SCHEMA_AWARE`.

The command-line interfaces `Transform` and `Query` will now load an enterprise configuration if they can (that is, if Saxon-EE and a valid license file can be located). The `-sa` option is now needed only to enable schema-awareness in a transformation or query that does not import a schema. This might be needed, for example, if the transformation or query uses untyped input but validates its output. The option is no longer needed to enable other Saxon-EE features such as advanced optimization or streaming.

Licensing changes

Saxon-PE and Saxon-EE (professional and enterprise editions) are available under commercial license terms. These impose the usual commercial restrictions, for example redistribution of the software is allowed only under an explicit agreement.

The open source product, Saxon-HE, is available under the same conditions as its predecessor, Saxon-B: that is, the Mozilla Public License.

The effect of this is that there are very few restrictions on applications built using Saxon-HE: the JAR file can be distributed with the application, and the application can be issued under any licensing terms you choose, whether commercial or open source. The only restriction you need to watch out for is that there is a requirement to distribute the notices contained in the `notices` directory whenever you distribute the JAR file itself. Some popular software distribution mechanisms such as `maven` are currently unable to satisfy this obligation.

License keys

Saxon-EE and Saxon-PE on .NET now uses the same license key files as Saxon on Java. For the time being, Saxon-EE on .NET will also work with previously issued .NET license keys, but all new license keys issued will be in cross-platform format.

For Saxon-EE and Saxon-PE on Java, it is no longer necessary for the directory containing the license key file to be on the classpath. Instead, the license file `saxon-license.lic` can be installed in the directory containing the `saxon9ee.jar` or `saxon9pe.jar` file, where Saxon will find it automatically. Saxon now looks first in this location, and then on the classpath.

Since the license key directory no longer needs to be on the classpath, the class `net.sf.saxon.Transform` is now the registered entry point for all three JAR files: saxon9he.jar, saxon9pe.jar, and saxon9ee.jar, making it possible to run all three products using the `-jar` option on the command line.

There has been some abuse of evaluation licenses, notably in developing countries. Two measures have been introduced to discourage the use of evaluation licenses for production work:

- Saxon now disables the use of evaluation licenses for a short period each day (typically five minutes, but random). These events are designed to be sufficiently rare that genuine evaluation projects are not impacted, but sufficiently frequent to cause a nuisance when attempting to run a production workload using an evaluation license.
- Occasionally and at random, in about 1% of runs, when running with an evaluation license Saxon will insert asterisks into the output when serializing.

S9API interface

Separate compilation of XQuery modules is available (under Saxon-EE only). An overloaded method `compileLibrary()` is available in the `XQueryCompiler` class to compile a library module; any subsequent compilation using the same `XQueryCompiler` may import this module (using `import module` specifying only a module URI - any location hint will be ignored), and the global functions and variables declared in the library module will be imported without incurring the cost of recompiling them.

The `XPathCompiler` has a new option to permit undeclared variables in XPath expressions. This allows an expression to be compiled without pre-declaring the variables that it references. It is possible to discover what variables are used in the expression (so that they can be initialized) by means of new methods provided on the `XPathExecutable` object.

The class `XdmValue` has a new method `append()` allowing a new `XdmValue` to be constructed by concatenating two existing instances of `XdmValue`.

The classes `DocumentBuilder` and `XdmDestination` have a new method `setTreeModel()` (and a corresponding accessor `getTreeModel()`) to indicate that what tree model should be used for the constructed tree. This allows selection of a linked tree in the case where XQuery Update access is required, or of the new condensed tiny tree. These methods are defined in terms of a new `TreeModel` class which in principle defines an extensibility point where new user-defined tree models can be supported.

The class `XdmNode` has a new method `getColumnNumber()` allowing the column number in the original lexical XML to be obtained, in cases where line numbers have been preserved.

The `Processor` object is now accessible to the code of extension functions by calling `context.getConfiguration().getProcessor()`, assuming that the method in question has a first argument of type `net.sf.saxon.expr.XPathContext`. This is useful when the extension function wants to create new nodes or invoke Saxon operations such as XSLT or XQuery processing.

A new mechanism is provided in the s9api `Processor` for declaring so-called extension functions. Unlike traditional extension functions invoked as Java methods through reflexion, an integrated extension function is implemented as a pair of classes: a class that extends the abstract class `net.sf.saxon.functions.ExtensionFunctionDefinition`, which defines static properties of the extension function, and a second class which extends `net.sf.saxon.functions.ExtensionFunctionCall`, and represents a specific call on the extension function, and provides a `call()` method to evaluate its result.

Many Saxon extension functions have been re-implemented using this mechanism; examples are `saxon:parse()` and `saxon:serialize()`.

The `XsltTransformer` class now has methods to get and set an `ErrorListener` for dynamic errors.

Saxon on .NET

Saxon on .NET is now built using IKVM 0.40. However, the OpenJDK Classpath library has been customised to reduce its size, by removing parts that Saxon does not need.

The Saxon DLL file now contains a cross-compiled copy of the Apache Xerces-J XML parser. The Sun fork of Xerces (which is part of the standard OpenJDK) is not included. Xerces is now the preferred XML parser; to use the Microsoft System.Xml parser instead, set the configuration option `PREFER_JAXP_PARSER` to false, or use Saxon API interfaces that take an `XmlReader` as an explicit argument. When a `DocumentBuilder` is used, the supplied `XmlResolver` will be used to dereference external entity references whichever parser is used.

Interfaces that take an `XmlReader` as an argument (for compiling a stylesheet or a schema) now use that `XmlReader` without modification: they no longer wrap a supplied `XmlTextReader` in an `XmlValidatingReader`, as these classes have been deprecated since .NET 2.0. It is therefore the user's responsibility to supply a correctly-configured `XmlReader`.

Saxon now uses the regular expression library provided in OpenJDK in preference to the .NET regular expression library. This avoids the need to maintain two copies of very similar code in Saxon, and it takes advantage of the Java regex handling of high Unicode characters.

The `XPathCompiler` has a new option to permit undeclared variables in XPath expressions. This allows an expression to be compiled without pre-declaring the variables that it references. It is possible to discover what variables are used in the expression (so that they can be initialized) by means of new methods provided on the `XPathExecutable` object.

The class `XdmValue` has a new method `Append()` allowing a new `XdmValue` to be constructed by concatenating two existing instances of `XdmValue`.

The `Processor` object is now accessible to the code of extension functions by calling `context.getConfiguration().getProcessor()`, assuming that the method in question has a first argument of type `net.sf.saxon.expr.XPathContext`. This is useful when the extension function wants to create new nodes or invoke Saxon operations such as XSLT or XQuery processing.

The `XdmDestination` object now has a `TreeModel` property, allowing a query or transformation result to be written to a LinkedTree, which makes it amenable to processing using XQuery Update.

A number of classes have been added to the API to represent types (notably `XdmSequenceType` and `XdmItemType` with subtypes such as `XdmAtomicType`). These were introduced primarily to support the new class `ExtensionFunction` which provides a way of implementing extension functions that does not rely on dynamic loading, and that can take advantage of information in the static and dynamic context. Instances of `ExtensionFunction` can be registered with the `Processor`.

XSLT

Static type checking is now implemented for non-tunnel parameters on `xsl:call-template` in the same way as for function calls: that is, the supplied value is compared against the required type, conversion code is generated if necessary, and errors are reported if the static type of the supplied value is incompatible with the required type. Tunnel parameters and parameters for `xsl:apply-templates` are checked dynamically as before. One effect of this change is that declaring the required type of parameters on named templates now gives a performance benefit as well as improving debugging and robustness.

In `<xsl:number level="any">`, the rules have been changed for the case where the node being numbered matches the pattern given in the `from` attribute: such a node is now numbered 1, whereas previously it was numbered according to its distance from the previous node that matched the `from` pattern, if any. This implements the change defined in W3C bug 5849 [http://www.w3.org/Bugs/Public/show_bug.cgi?id=5849].

For the three serialization parameters `doctype-system`, `doctype-public`, and `saxon:next-in-chain`, supplying "" (a zero-length string) as the value of the parameter is taken as setting the parameter to "absent". This is equivalent to omitting the parameter, except that it overrides any setting that would otherwise be used. For example, this allows a value set in `<xsl:output>` to be overridden in an importing stylesheet, in an `<xsl:result-document>` instruction, in the JAXP `setOutputProperties()` method, or from the command line (where the syntax is simply `!doctype-system=`). (Note that this takes a slight liberty with the W3C and JAXP specifications.)

The `doc-available()` function, when it returns false, now ensures that the document remains unavailable for the rest of the transformation: previously, if called repeatedly it would check repeatedly, and therefore could return different results on different calls. Also, once `doc-available()` has returned false, subsequent calls on `doc()` or `document()` are now guaranteed to fail. A call on `doc-available()` that returns false does not prevent the document being created using `xsl:result-document`, but any such document will not be available during the same transformation.

The new function `element-with-id()`, introduced in the errata for Functions and Operators, is available. It behaves the same as the `id()` function, except in the case of ID-valued elements, where it returns the parent of the element having the `is-ID` property, rather than the element itself.

For `<xsl:number>`, numbering sequences have been added for format tokens x2460 (circled digits), x2474 (parenthesized digits), and x2488 (digit followed by full stop). In each case the numbering sequence only handles numbers in the range 1-20; numbers outside this range are formatted using the format token "1" (that is, as conventional decimal numbers).

The option `input-type-annotations="strip"` is now honoured for a document supplied in the form of a pre-built tree, by creating a view of the tree in which all nodes appear as untyped. Previously it was honoured only when the tree was built by the XSLT processor.

The `TimedTraceListener`, used for timer profiling, is now capable of writing the profile output to a destination other than `System.err`. This option cannot however be enabled from the command line, only from the Java API.

A stylesheet that does not use an `xsl:import-schema` declaration is now (by default) compiled with schema-awareness disabled. This means that it will not be able to handle schema-typed input documents, or to validate temporary trees created within the stylesheet, though it can still validate the final output tree. This is for performance reasons: generating code to handle typed input data when it will not be encountered adds to the execution cost. It is possible to override this setting from the s9api API on Java or from the Saxon.Api on .NET. From the command line, schema-awareness is set automatically if the -sa option or any other option implying schema-awareness is used (for example -val:strict). From JAXP, schema-awareness is set automatically if the schema-aware TransformerFactory is used.

The amount of compile-time checking when schema-awareness is used has been further increased. In particular, if the expected type of a constructed element is known, Saxon now attempts to check (a) that the sequence constructor delivering the content of the element is capable of delivering a sequence of elements that matches the content model (previously it only checked that each child element could legitimately belong to the content model), and (b) that the sequence constructor is capable of creating each mandatory attribute required by the complex type of the element.

In the interests of performance, the decision whether to treat an ambiguous template rule match as a fatal error, a warning, or as fully recoverable, is now made at stylesheet compile time rather than at run-time.

The extension attribute `saxon:allow-all-built-in-types` is no longer recognized.

Command line changes

If the filename specified in the -o option is in a directory that does not exist, the directory is now created.

A new option `-config:filename` is available. This refers to a configuration file in which many configuration options can be specified. Options specified directly on the command line override corresponding options in the configuration file. The format of the configuration file is given in The Saxon configuration file.

Saxon-defined serialization parameters can now be defined on the command line using a lexical QName, for example `!saxon:indent-spaces=3` as an alternative to the Clark-format expanded name.

The command line interface (`net.sf.saxon.Transform`) now allows a stylesheet parameter to be supplied as the value of an XPath expression. For example, `java net.sf.saxon.Transform -xsl:style.xsl ?p=current-dateTime()` sets the stylesheet parameter `p` to the value of the current date and time (as an instance of `xs:dateTime`). The fact that the value needs to be evaluated as an XPath expression is signalled by the leading "?" before the parameter name. Note that the context for evaluating the expression includes only the "standard" namespaces, no context item, and no variables.

A new option is available (`-dtd:recover` on the command line) to perform DTD validation but treat the error as non-fatal if it fails.

A new option `-now` is available on the command line to set the current date and time (and the implicit timezone). This is designed for testing, to enable repeatable results to be obtained.

XQuery 1.0

The syntax for `declare option saxon:output "parameter=value"` now allows the value to be zero-length.

The default serialization options have been changed to align with the defaults in Appendix C.3 of the W3C specification: specifically, the default for `indent` is now "no". The old default of "yes" can be achieved by setting the value from the command line, from the Java API, from the Query prolog, or from the Saxon configuration file.

For the two serialization parameters `doctype-system` and `doctype-public`, supplying "" (a zero-length string) as the value of the parameter is taken as setting the parameter to "absent". This is equivalent to omitting the parameter. For example, this allows a value set in the query prolog to be overridden from the API or from the command line (where the syntax is simply `!doctype-system=`).

The `doc-available()` function, when it returns false, now ensures that the document remains unavailable for the rest of the query: previously, if called repeatedly it would check repeatedly, and therefore could return different results on different calls. Also, once `doc-available()` has returned false, subsequent calls on `doc()` are now guaranteed to fail. A call on `doc-available()` that returns false does not prevent the document being created using `put()`, but any such document will not be available during the same query.

The new function `element-with-id()`, introduced in the errata for Functions and Operators, is available. It behaves the same as the `id()` function, except in the case of ID-valued elements, where it returns the parent of the element having the `is-ID` property, rather than the element itself.

A query that does not use an `import schema` declaration is now (by default) compiled with schema-awareness disabled. This means that it will not be able to handle schema-typed input documents, or to validate temporary trees created within the query, though it can still validate the final output tree. This is for performance reasons: generating code to handle typed input data when it will not be encountered adds to the execution cost. It is possible to override this setting from the s9api API on Java or from the Saxon.Api on .NET, or from the configuration file. From the command line, schema-awareness is set automatically if the -sa option or any other option implying schema-awareness is used (for example `-val:strict`). With XQJ, schema-awareness is set automatically if Saxon-EE is loaded.

The amount of compile-time checking when schema-awareness is used has been further increased. In particular, if the expected type of a constructed element is known, Saxon now attempts to check (a) that the sequence constructor delivering the content of the element is capable of delivering a sequence of elements that matches the content model (previously it only checked that each child element could legitimately belong to the content model), and (b) that the sequence constructor is capable of creating each mandatory attribute required by the complex type of the element.

Command line changes

If the filename specified in the -o option is in a directory that does not exist, the directory is now created.

A new option `-config:filename` is available. This refers to a configuration file in which many configuration options can be specified. Options specified directly on the command line override corresponding options in the configuration file. The format of the configuration file is given in The Saxon configuration file.

A new option is available (`-dtd:recover` on the command line) to perform DTD validation but treat the error as non-fatal if it fails.

In the command line interface, Saxon-defined serialization parameters can now be defined using a lexical QName, for example `!saxon:indent-spaces=3` as an alternative to the Clark-format expanded name.

The command line interface (`net.sf.saxon.Query`) now allows an external variable to be supplied as the value of an XPath expression. For example, `java net.sf.saxon.Query -xsl:style.xsl ?p=current-dateTime()` sets the external variable `p` to the value of the current date and time (as an instance of `xs:dateTime`). The fact that the value needs to be evaluated as an XPath expression is signalled by the leading "?" before the parameter name. Note that the context for evaluating the expression includes only the "standard" namespaces, no context item, and no variables.

XQJ changes

XQJ support has been updated to the final JSR 225 specification published on 24 June 2009.

This involves the removal of two methods, the overloads of `bindDocument()` on `XQDataFactory` and `XQDynamicContext` that took an `XMLReader` as argument. It's unlikely anyone was using these methods because the specification was so weird, but it should be easy to replace any call with a call on the overload that accepts a `Source` (by supplying a `StreamSource` that wraps the `XMLReader`).

XQuery Updates

The XQuery update code has been amended to classify expressions as "updating" or "vacuous" as in the Candidate Recommendation. To ensure all cases of invalid updating expressions are properly detected, Saxon now delays some of its rewriting of conditional and typeswitch expressions until the type-checking phase.

In XQuery Update, following a decision on W3C bug 5702, deleting a node that has no parent is no longer an error.

The XQuery Update implementation has been significantly improved, responding to clarifications in the W3C specifications, and an improved test suite. Many of the changes are minor, affecting behaviour in error or edge cases. It has proved necessary to change the `MutableNodeInfo` interface to reflect the fact that an attribute's identity cannot be adequately inferred from its name (an attribute can now be renamed without changing its identity, and conversely, it can be replaced by another attribute having the same name). In the linked tree implementation, attribute identity is now associated with the identity of the element node object together with the index position of the attribute; index positions are not reused when an attribute is deleted. A method `isDeleted` has been added to allow testing whether an object refers to a node that has been deleted; no further operations should be attempted with such an object.

XQuery 1.1

These new features are available only with Saxon-PE or Saxon-EE, and require XQuery 1.1 to be enabled (a) from the command line (-qversion:1.1) or Configuration and (b) from the query prolog (`xquery version "1.1";`).

The try/catch syntax from the draft XQuery 1.1 specification is implemented, but without the ability to declare variables to receive error information. This feature cannot be used with XQuery Updates.

A subset of the grouping syntax from the draft XQuery 1.1 specification is implemented. The `group by` clause must be preceded in the FLWOR expression by (a) a single `for` clause, which selects the sequence to be grouped, and (b) a single `let` clause, which defines the grouping key; the "group by" clause must name the variable that is declared in the `let` clause. For example: `for $x in //employee let $k := $x/department group by $k return ...`. Within the `return` clause, `$x` refers to the content of the current group, and `$k` to the current grouping key.

The "outer for" clause of a FLWOR expression is implemented. The implementation is functionally complete, but there is no optimization.

Computed namespace node constructors are supported, in the form `namespace prefix {uri-expression}` or `namespace {prefix-expression} {uri-expression}`.

In the query prolog, it is now possible to provide a default value for an external variable (for example, `declare variable $ext external := 0;`.

The `declare context item` declaration in the query prolog is implemented. This allows a required type and a default value to be declared for the context item. At present (the rules aren't entirely clear) it is possible to specify a value from the calling API, or to not specify a value, regardless whether "external" is specified or not. At present there is no interaction with the API facilities for defining a required type for the context item: both can be used independently.

The expression `validate as type-name { expr }` is implemented.

The functions `format-date()`, `format-time()`, and `format-dateTime()`, as specified in XSLT 2.0, are now also available in XQuery 1.1.

The function `format-number()` is now available, along with the new syntax in the Query Prolog to declare a (named or default) decimal-format. (This has entailed some internal change in the way decimal formats are managed, since XQuery allows each module to have its own set of named decimal formats.)

Higher-order functions

The new facility for higher-order functions is fully implemented, with one or two restrictions.

The syntax `my:function#3` is now available. This is synonymous with the extension available in earlier releases, `saxon:function('my:function', 3)`. This has also been extended so that it works with all functions; the Saxon extension previously worked only with user-written functions.

The `SequenceType` syntax `function()` is now available to denote the type of a function item, that is, the type of the result of `my:function#3` or `saxon:function('my:function', 3)`. You can also use a full type signature, for example `function(xs:int, xs:int) as xs:string*`.

The type `function()` is implemented as a new subtype of `Item` represented by the Java class `net.sf.saxon.om.FunctionItem`. Note that any code that assumes every Item is either a node or an atomic value is potentially affected.

Dynamic function calls can now be written, for example, as `$f(x, y)` rather than `saxon:call($f, x, y)` as previously. In this expression `$f` can be replaced by any primary expression or filter expression whose value is a function item.

Inline (anonymous) functions can be written, for example `function ($x as xs:integer) as xs:boolean {$x mod 2 eq 0}`. Such a function will typically be used as an argument in a function call expecting a parameter of type `function()`.

The functions `fn:function-name()`, `fn:function-arity()`, and `fn:partial-apply()` are implemented.

Saxon applies function coercion when a function is passed to another function, or when it is returned as a function result. However it also implements a proposed change to the specification whereby function coercion is not used for operations such as "instance of". These follow stricter type checking rules: a function `F(A,B)->T` is an instance of a type `F(X,Y)->U` if every T is an instance of U, every X is an instance of A, and every Y is an instance of B.

XML Schema

XML Schema 1.0

The implementation of `xs:redefine` has been revised slightly, to better reflect the notion of "pervasiveness" described (with tantalising lack of detail) in the language specification. Every component now has a redefinition level; a component defined within `xs:redefine` has a redefinition level one higher than that of the component it redefines. If two different but identically-named components have the same redefinition level, this is an error; but if they have different redefinition levels, the higher one wins. An example where this comes into play is if a module R.XSD contains two `xs:redefine` elements; the first one redefines A.XSD, which itself includes B.XSD, and the second redefines B.XSD, providing a revised version of a type T contained in B.XSD. Previously Saxon reported this as an error, on the grounds that the schema returned by the first `xs:redefine` and the schema returned by the second `xs:redefine` could not be combined because they contained incompatible definitions of type T. This schema is now accepted as valid, and the redefined type T wins.

In the command line interface, a new option `-config:filename` is available. This refers to a configuration file in which many configuration options can be specified. The configuration described by this file must be an EnterpriseConfiguration. Options specified directly on the command line override corresponding options in the configuration file. The format of the configuration file is given in The Saxon configuration file.

The bogus `gMonth` format `--MM--` is no longer accepted. This format appeared in error in the original XML Schema 1.0 Recommendation, and was subsequently corrected by erratum. It still appears in a number of books on XML Schema. It was recognized in all Saxon releases until and including 9.1. The correct format is `--MM`.

XML Schema 1.1

The type `xs:error` has been implemented.

The facility for conditional inclusion of parts of schema documents, based on attributes such as `vc:minVersion` and `vc:maxVersion`, is now implemented. This works whether the schema processor is in 1.0 or 1.1 mode, allowing a schema that is processed in 1.0 mode to ignore facilities such as assertions that require XSD 1.1.

For this purpose the set of types that are "automatically known" to the processor includes only the built-in types (which are currently the same for XSD 1.0 and XSD 1.1, with the exception of `xs:error` which is available only in 1.1), and the set of facets are the built-in facets, which includes `xs:assert` when running in 1.1 mode, but not when running in 1.0 mode.

An element declaration may now appear in more than one substitution group.

The `ref` attribute of `xs:unique`, `xs:key`, and `xs:keyref` has been implemented.

The `notNamespace` and `notQName` attributes are now supported on `xs:any` and `xs:anyAttribute` wildcards. However, the option `notQName="##definedSibling"` is not yet implemented.

Although these attributes are only available when XSD 1.1 is enabled (command line switch `-xsdversion:1.1`), a side-effect of the change is that wildcard unions and intersections that were not expressible in 1.0 are now expressible, and this change applies whether or not 1.1 is enabled. This means that some rather obscure conditions that are errors in 1.0 but not in 1.1 are no longer detected as errors.

The algorithm for testing subsumption among `xs:all` content models now performs a more intelligent analysis of wildcard particles.

Open content is implemented. The `xs:defaultOpenContent` element can appear as a child of `xs:schema`, and and `xs:openContent` as a child of `xs:complexType`, `xs:extension` or `xs:restriction`. This works with sequence, all, empty and mixed content models.

There are some limitations in subsumption testing: for a type R to be a valid restriction of B, the "primary content" of R must be a valid restriction of the primary content of B, and the "open content" of R must be a valid restriction of the open content of B.

Default attributes are implemented (that is, the `defaultAttributes` attribute of `xs:schema`, and the `defaultAttributesApply` attribute of `xs:complexType`).

An `xs:all` content model may now be derived by extension from another `xs:all` content model.

A new facet `xs:explicitTimezone` is available with values `required`, `optional`, or `prohibited`. This allows control of whether or not the timezone part of a `date`, `time`, `dateTime`, `gYear`, `gYearMonth`, `gMonth`, `gMonthDay`, or `gDay` is present or absent.

The new built-in data type `xs:dateTimeStamp` (an `xs:dateTime` with timezone required) is implemented.

Saxon XSD Extensions

In assertions, and on all elements representing facets (for example `pattern`), Saxon supports the attribute `saxon:message="message text"`. This message text is used in error messages when the assertion or facet is not satisfied.

The XSD 1.1 specification allows vendor extensions in the form of vendor-defined primitive types and vendor-defined facets. Saxon 9.2 exploits this freedom to provide a new facet, `saxon:preprocess`. This is a pre-lexical facet (like `xs:whiteSpace`) in that it is used to transform the supplied value before validation. The preprocessing is done using an arbitrary XPath expression which takes a string as input and produces a string as output. For example `<saxon:preprocess action="normalize-unicode($value)"/>` can be used to perform Unicode normalization, while `<saxon:preprocess action="upper-case($value)"/>` normalizes the value to upper case. This is done before application of other facets such as the `pattern` and `enumeration` facets. It is also possible to provide another XPath expression to reverse the process, for use when the typed value is converted back to a string. For more information see The saxon:preprocess facet.

Streaming

The `saxon:iterate` instruction and its subsidiary instructions `saxon:break`, `saxon:continue`, and `saxon:finally` are now available only in Saxon-EE. This reflects the fact that the instructions are designed primarily for use with streaming, which is available only in Saxon-EE.

The behaviour of `saxon:iterate` has been changed in the case where a `saxon:continue` instruction does not specify values for all the parameters declared on the containing `saxon:iterate` instruction. Any parameters for which no value is supplied now retain their previous value (that is, the effect is the same as specifying `<xsl:with-param name="p" select="$p"/>`). Previously such parameters reverted to their default value. A further change is that it is no longer possible to specify `required="yes"` on `saxon:iterate` parameters. These changes are in line with the (as-yet-unpublished) XSLT 2.1 draft specification.

Saxon 9.2 introduces the concept of . This allows hierarchic processing of a document using `<xsl:apply-templates>` to operate in a streaming pass over the document, without building the tree in memory. The templates, of course, have to conform to strict rules to make them streamable. Nevertheless a great many simple transformations can be implemented this way: for example, renaming elements, deleting selected elements, computing new attribute values, and so on.

For more details see Streaming Templates.

Functions and Operators

In regular expressions, the rules on back-references as defined in errata E4 and E24 have now been implemented. Backreferences of more than two digits are now recognised; a back-reference is recognized as the longest sequence of digits after "\" that is either one digit or is longer than the number of preceding left parentheses; a backreference must not start with the digit zero; a backreference N is an error if it appears before the closing bracket corresponding to the Nth opening bracket.

The rules for regular expressions in the draft XML Schema 1.1 specification clarify the ways in which hyphens may be used within a character class expression (that is, within square brackets). To implement these rules, Saxon now disallows an unescaped hyphen at the start or end of a character range (for example `[--a]`).

The option `alphanumeric=codepoint` is now available in collation URIs to request alphanumeric collation (integers embedded in the string are sorted as integers) with codepoint collation for the "alpha" parts of the string.

The `collection()` function now allows directories of text files to be read, provided the text uses characters that are legal in XML. This is achieved using the additional query parameter `unparsed=yes` in the collection URI. The resulting files are returned in the form of document nodes, each having a single text node as a child. The platform default encoding is assumed.

XML Parsing and Serialization

Parsing

If a `SAXSource` containing an `XMLReader` is supplied to Saxon, Saxon now respects the `ErrorHandler` associated with the `XMLReader` rather than replacing it with its own.

Serialization

Some very basic support for HTML 5 has been added. If the serialization method is "html" and the version is "5.0", a heading `<!DOCTYPE HTML>` will be output regardless of the `doctype-system` and `doctype-public` properties.

A new serialization option `saxon:recognize-binary` has been added for use with the `text` output method (only). If set to yes, the processing instructions `<?hex XXXX?>` and `<?b64 XXXX?>` will be recognized; the value is taken as a hexBinary or base64 representation of a character string, encoded using the encoding in use by the serializer, and this character string will be output without validating it to ensure it contains valid XML characters. This enables non-XML characters, notably binary zero, to be output. For example, `<?hex 0c?>` outputs an ASCII form feed. Also recognized are `<?hex.EEEE XXXX?>` and `<?b64.EEEE XXXX?>`, where EEEE is the name of the encoding of the base64 or hexBinary data: for example `hex.ascii` or `b64.utf8`.

A new UTF8 writer, contributed by Tatu Saloranta, is used in place of the standard Java UTF8 writer. The effect is to speed up serialization by around 20%; for a transformation that copies its input to its output, the improvement is about 10% overall.

External Object Models

Note that the support modules for JDOM, XOM, and DOM4J are not packaged with Saxon-HE, but they can be built from source code if required.

Saxon's JDOM interface now supports use of the `id()` and `idref()` functions.

The DOM interface now supports the `unparsed-entity-uri()` and `unparsed-entity-system-id()` functions.

The DOM interface now filters out zero-length text nodes. (It also treats nodes whose nodeValue is null as if it were a zero-length string: this was a bug fix also applied to the 9.1 branch.)

The XOM interface (`XOMObjectModel`) and the JDOM interface (`JDOMObjectModel`) both now implement the new `TreeModel` interface, which means they can be set as the desired tree model for example in a s9api `DocumentBuilder` or `XdmDestination`. In fact, they can even be set as the selected tree model in a `Controller`, in which case the selected model will be used for temporary trees constructed within a query or transformation.

Extensibility

A new way of implementing extension functions is available. These are known as extension functions. Unlike traditional extension functions invoked as Java methods through reflexion, each integrated extension function is implemented as a separate class that extends the abstract class `net.sf.saxon.functions.ExtensionFunctionDefinition`, together with a class that extends `net.sf.saxon.functions.ExtensionFunctionCall`. The first class must implement a number of methods providing information about its argument types and result types, while the second provides the `call()` method which is invoked to call the function. A key advantage of this kind of extension function is that it can exercise compile-time behaviour, for example saving information about the static context in which it is called, or optimizing itself based on the expressions supplied in the function arguments. Another advantage is that you have complete control over the function name, for example you can choose any namespace you like.

Integrated extension functions need to be registered in the Saxon Configuration. You can do this via the API, or (in Saxon-PE and Saxon-EE) via the configuration file. This means that in Saxon Home Edition it is not possible to use integrated extension functions when running from the standard command line interface.

The old way of binding extension functions ("reflexive extension functions"), by mapping the namespace URI of the function name to a Java class, is not available in Saxon Home Edition. Also, it has changed so that by default, it only recognizes namespaces using the recommended format `java:full.class.name`. If you need to recognize arbitrary namespaces ending in the Java class name (as allowed in previous releases on XSLT, but not XQuery), find the `JavaExtensionLibrary` by calling `Configuration.getExtensionBinder("java")`, and on this object call `setStrictJavaUriFormat(false)`.

The `ExtensionFunctionFactory` (which can be used to change the details of the calling mechanism for reflexive extension functions, for example to perform diagnostic tracing) is now set as a property of the `JavaExtensionFunctionLibrary` or `DotNetExtensionFunctionLibrary`, rather than directly via the `Configuration`.

Under .NET, if an extension function expects a value of type `XdmAtomicValue`, it is now possible for the calling XPath expression to supply a node; the node will be automatically atomized.

The factory mechanism for handling XSLT extension instruction namespaces has changed. Previously it was required that the namespace URI should end in the Java class name of a class implementing the `ExtensionElementFactory` interface. It is now possible to use any namespace URI you like. The namespace you choose must be associated with the name of the `ExtensionElementFactory` class in the Configuration, which you can do programmatically by calling `Configuration.setExtensionElementNamespace()`, or from the configuration file. Element extensibility is not available in Saxon Home Edition, which also means that the SQL extensions are not available.

Extensions

A new extension function `saxon:adjust-to-civil-time()` is available. This adjusts a date/time value to its equivalent in the civil time of a named timezone, taking account of summer time (daylight savings time) changes. For example `adjust-to-civil-time(xs:dateTime('2008-07-10T12:00:00Z', 'America/New_York')` returns `2008-07-10T08:00:00-04:00`

Two new XSLT extension instructions are available (in Saxon-EE only): `<saxon:try>` and `<saxon:catch>`. These complement the existing `saxon:try()` extension function, but are more powerful, and more convenient in many cases because they catch errors at the XSLT level. For details see saxon:try.

A new extension function `saxon:parse-html()` is available. This works in the same way as `saxon:parse()`, but it assumes the input is HTML rather than XML. The TagSoup parser must be available on the classpath for this to work.

The extension functions `saxon:file-last-modified()` and `saxon:last-modified()` have been changed. There is now only one function, `saxon:last-modified()`, and it takes a URI as its argument. It can now be a relative or absolute URI (if relative, it is resolved against the base URI from the static context). To get the last-modified date of the file from which a particular node was loaded, use `saxon:last-modified(document-uri(/))`. The function now throws a dynamic error if given an invalid URI or a URI that cannot be resolved or dereferenced; this can be caught using try/catch if needed. The function returns an empty sequence if the resource can be retrieved but no date/time is available.

In the `saxon:try()` extension function, the ability to supply a function as the second parameter (to be called if evaluation of the first parameter fails) is withdrawn. The second argument is now always evaluated directly.

The support for the EXSLT date-and-time library has been updated. Some new methods are available (add(), difference(), seconds(), duration(), sum()), and implementations of existing functions have been upgraded to conform more closely to the spec, in particular by checking for erroneous input. One change that may be noticed is that the function `date()`, with no arguments, now returns the current date with a timezone.

The extension function `saxon:function()` has changed so the argument can now be a system function as well as a user-defined function. If the function name is unprefixed, it is now assumed to be in the default function namespace.

The extension functions `saxon:highest()`, `saxon:lowest()`, `saxon:sort()`, and `saxon:leading()` have been reworked to take advantage of higher-order functions. The second argument (if present) is now a function item rather than a dynamic expression.

The extension function `saxon:after()`, which has been undocumented for many years, is finally dropped from the code.

The extension functions `saxon:index()` and `saxon:find()` now impose the same comparison rules as value comparisons: for example, if the value indexed is untyped atomic, then supplying an integer as the argument to `saxon:find()`results in a type error. Previously, it resulted in a "no match" (typically, `saxon:find()` returned an empty sequence).

The extension function `saxon:type-annotation()`, when applied to a document node, now returns `xs:anyType` if the document has been schema-validated, or `xs:untyped` otherwise. Previously it return `xs:untypedAtomic`. For comment, processing-instruction, and namespace nodes it now returns `xs:string`.

The XSLT extension instruction `saxon:script` is dropped. It suffered a major design problem: the bindings it declared were global (applying to a Configuration as a whole, and in part to all Configurations in the Java VM, rather than to a particular stylesheet), and the cost of fixing this problem would not be justified by the level of usage of this legacy feature. Equivalent facilities are now available via the configuration file, or using the Configuration API.

Optimizations

A new configuration option is available to control the optimization level. This appears as the `-opt` option on the Query and Transform interfaces, and as `FeatureKeys.OPTIMIZATION_LEVEL` on APIs such as `Configuration.setConfigurationProperty()` and `TransformerFactory.setAttribute()`. The value is an integer in the range 0 (no optimization) to 10 (full optimization); currently all values other than 0 result in full optimization but this is likely to change in future. The default is full optimization; this feature allows optimization to be suppressed in cases where reducing compile time is important, or where optimization gets in the way of debugging, or causes extension functions with side-effects to behave unpredictably. (Note however, that even with no optimization, lazy evaluation may still cause the evaluation order to be not as expected.)

A function call appearing within a loop, but with no dependency on the loop variables, can now be moved out of the loop, provided the function does not create new nodes. Previously, the worst-case scenario was assumed: that the function could create new nodes, and that it therefore needed to be called repeatedly even if the arguments were unchanged. The analysis of whether the function creates new nodes is now done in all cases except for recursive functions, where the worst-case is still assumed.

Note that "creates new nodes" here means "creates new nodes and returns a result that depends on the node identity". A function that creates new nodes and immediately atomizes them is not considered to be creative, and can safely be moved out of a loop. By contrast, a function whose result depends on the XSLT `generate-id()` function is considered creative in all cases.

The design of the `LargeStringBuffer` used to hold the content of text nodes in the tiny tree has changed to use fixed-length segments instead of variable-length segments. The result is that in general, locating text is faster, with the downside that more data copying is needed for unusually long text nodes. Overall, in the XMark benchmark, this shows an improvement of 5% in query execution times; occasionally 10%.

A new variant of the tiny tree data structure can be selected at user option. This is the "condensed tiny tree". While building a condensed tiny tree, the system checks before creating a text or attribute node whether there is already another node with the same string value; if so, the value is only stored once. The tree thus takes a little longer to build (perhaps 10Mb/sec rather than 15Mb/sec) but will typically occupy less memory. The saving in memory obviously depends greatly on the nature of the data. This option is selected from the Transform or Query command line using the option `-tree:tinyc`, or from the API using the value "tinyTreeCondensed" for the configuration option `TREE_MODEL_NAME`, or the value `Builder.TINY_TREE_CONDENSED` in various `setTreeModel()` interfaces.

By default the tiny tree now maintains a cache of the typed values of element and attribute nodes. The typed value is held in this cache if it is an instance of string, untypedAtomic, or anyURI (which means that the cache is only populated for Saxon-EE). The typed value is placed in the cache the first time it is computed during the course of a query or transformation (not at the time of initial validation). If this uses excessive memory, or if it delivers no benefit for the query/transformation in question (which can happen if each element/attribute is only processed once, for example) then there is a configuration option `USE_TYPED_VALUE_CACHE` to disable it.

In XQuery FLWOR expressions, the rewriting of the "where" conditions into predicates applied to the individual "for" clauses is now done more vigorously. Previously it was done only for terms in the where condition that were potentially indexable, for example a value comparison; it is now done for expressions of any kind. Where the FLWOR expression is evaluated by nested looping, this can significantly reduce the number of iterations of the inner loop. The rewrite is also less likely to be prevented by the presence of references to the context item within the predicate (in most cases these can now be converted into reference to a variable declared and bound to the context item at an outer level). Finally, a predicate of the form `where not(A or B or C)` is now converted into `where not(A) and not(B) and not(C)` before this redistribution of predicate terms is attempted.

A simple set of rewrites for boolean expressions have been introduced: (A and true()) is rewritten as (A), while (A or false()) is rewritten as (A). Of course the importance of these is that they simplify the expression making it a candidate for further more powerful optimizations, such as indexing.

Global variables can now be indexed in Saxon-EE. Previously this was done only for local variables. A global variable V will be indexed if there is any filter expression of the form `$V[@X = Y]` where @X represents any expression whose value depends on the context node, and Y represents any expression whose value does not depend on the context node. (Variations are possible, of course: the operands can be in either order, and the operator can be "eq" rather than "=".)

When Saxon-EE extracts expressions from templates and functions into new global variables, it now ensures that if an expression appears more than once, only a single global variable is created. (This depends on the expressions being recognized as equal, which does not happen in all cases.) A particular benefit occurs with stylesheets that make heavy use of attribute sets (typically, XSL-FO stylesheets): any attribute set whose value has no context dependencies is now computed once as a global variable (its value being a sequence of attribute nodes, which are copied each time the attribute set is referenced).

The functions `concat()` and `string-join()` are now capable of operating in push mode. This means that with a query such as `<a>{string-join(//a, '-')}</a>`, the output of the `string-join()` function is streamed directly to the serializer, rather than being constructed as a string in memory. The same applies to the `select` expression of `xsl:value-of`; for example the expression `<xsl:value-of select="1 to $n"/>` now streams its output to the serializer without allocating memory for the potentially-large string value.

An optimization for `translate()`, using a hashmap rather than a serial search to map individual characters, was present in earlier releases but only activated if the second and third arguments were string literals. The optimization is now activated for run-time lookups as well, provided the product of the lengths of the first and second arguments exceeds 1000 (a threshold obtained by doing some simple measurements).

Some changes have been made to the `NamePool` (and the way it is used) to reduce contention. Whenever a nameCode is allocated, a corresponding namespaceCode is now allocated at the same time, which means that users of this nameCode can be confident that the namespaceCode is already in the NamePool, avoiding the need for another synchronized method call (which was often being done at run time).

Internals

The move to J2SE 5 has enabled a number of internal changes:

- It has enabled simplification of the code in a number of areas; Saxon can now assume the existence of classes such as `javax.xml.namespace.QName` and other newer features of JAXP, and it can exploit methods in `BigDecimal` that were not available in JDK 1.4.
- The JDK 1.4 regular expression handler has been dropped.
- Many uses of the Java collection classes within the Saxon code have been converted to use generics.
- Saxon no longer includes its own tables of character sets. Instead, if an encoding other than ASCII, UTF8, UTF16, or ISO-8859-1 is requested for serialization, Saxon relies on the character encoding data in the `java.nio` package to determine which characters are available (unavailable codepoints are represented as character references if they appear in element or attribute values, and cause serialization errors if they appear in comments, element or attribute names, etc.). As a consequence of this change, the configuration interfaces to add additional character encodings have been dropped.
- There is no longer a need to support multiple versions of the DOM interface, which means that it has become possible to bring the DOM support code back into the main JAR file.
- The s9api interface code was always compiled under JDK 1.5, and was shipped in a separate JAR file to enable the rest of the code to work with JDK 1.4. This separation is no longer necessary, so s9api is available in the same JAR file as everything else.

Version 9.1 (2008-07-02)

- Highlights
- XQuery Updates
- XML Schema 1.1
- XML Schema 1.0
- XSLT 2.0
- XQuery 1.0
- XQJ (XQuery API for Java)
- S9API
- JAXP

- Extensibility
- Extensions
- Diagnostics and Tracing
- Saxon on .NET
- Internal APIs
- Serialization
- Optimization

Highlights

This page lists some of the most important new features introduced in Saxon 9.1.

The following apply to Saxon-SA only:

1. Streaming extensions: more streaming capabilities in XSLT, and addition of streaming capabilities to XQuery.
2. XQuery Updates: Saxon now implements the (draft) XQuery Updates specification.
3. XML Schema 1.1 support: Saxon 9.1 implements many of the more significant features from the draft XML Schema 1.1 specifications, including Assertions, Conditional Type Assignment (co-occurrence constraints), and `xs:all` groups with flexible occurrence limits.
4. Schema implementation now uses counters

In Saxon-B, the most significant feature is probably support for the latest (almost-final) draft of the XQJ specifications (Java API for XQuery).

XQuery Updates

Saxon 9.1 introduces support for XQuery Updates.

To run an updating query from the command line, use the option `-update:on` on the command line. From Java, compile the query in the normal way, and use the method `XQueryExpression.runUpdate()` to execute the query. At present it is not possible to run updates using the higher-level interfaces in XQJ or S9API, or using the .NET API.

To enable updating internally, an extension of the `NodeInfo` interface has been introduced, called `MutableNodeInfo`. Currently the only tree implementation that supports this interface is the linked tree (formerly called the standard tree). In principle updating should work with any tree model that supports this interface, though at present there are probably some dependencies on the specific implementation.

The linked tree has been improved so that it can now handle schema type annotations, and there have been improvements to the way in which line number information is maintained.

Saxon does not currently include any locking code to prevent concurrent threads attempting to update the same document. This is something that applications must organize for themselves.

At the Java API level, updated documents are not automatically written back to disk. Rather, the `runUpdate()` method returns a set containing the root nodes of documents that have been updated. This may include documents that were read using the doc() or collection() functions as well as documents supplied as the context node of the query or by binding external variables. It may also include documents (or elements) that were created by the query itself - but only if they have been updated after their initial construction. There is a helper method `QueryResult.rewriteToDisk` allowing such documents to be written back to disk if required, at the URI determined by their `document-uri()` property, but this must be explicitly invoked by the application. Clearly if the application does not have write access to the URI, this operation will fail.

Alternatively, updated documents may be written back to disk using the `put()` function. Note that because `put()` delivers no result, there is a danger that it will be "optimized out" of the query. Saxon implements `put()` using the runtime code for `xsl:result-document`, and the implementation currently imposes similar restrictions: in particular, it must not be called while evaluating a variable or function.

Applications should access the updated document via the root nodes returned by the `runUpdate()` function. Use of `NodeInfo` objects that were obtained before running the update is in general unsafe; while it will probably work in most cases, there are some cases (particularly with attributes) where the `NodeInfo` object returned to the application is a snapshot created on demand, and this snapshot will not be updated when the underlying tree is changed.

Updates are not currently atomic, as required by the specification. In particular, if the update fails revalidation against the schema, it is not rolled back.

XML Schema 1.1

There is now a configuration flag to enable use of XML Schema 1.1 syntax; if this flag is not set, all new XML Schema 1.1 features will be disabled. The flag can be set using `-xsdversion:1.1` on the command line (of `Query`, `Transform`, or `Validate`), or by calling `configuration.setConfigurationProperty(FeatureKeys.XSD_VERSION, "1.1")`.

(often called co-constraints) is implemented. Any XPath expression may be used to define the condition, so long as it only accesses the attributes and namespaces of the element in question. Rules on type subsumption not yet implemented.

The attribute is supported for both `xs:assert` and `xs:alternative` (but not yet for `xs:field` or `xs:selector`). The `xpathDefaultNamespace` attribute on `xs:schema` is also recognized.

A model group defined with an `<xs:all>` compositor may now have arbitrary `minOccurs` and `maxOccurs` values on the element particles within the group. Much more analysis is now done to determine whether a sequence of choice group is a valid restriction of a type that uses an `xs:all` compositor; some of this will also apply to XSD 1.0 schemas. For example, substitution groups are now taken into account, and the derived type is allowed to have an `xs:choice` content model (each branch of the choice must be a valid restriction of the base content model.)

Element wildcards (`<xs:any>`) are now allowed in an a model group defined using `<xs:all>`.

Local element and attribute declarations can now have a `targetNamespace` attribute, provided that they appear within an `xs:restriction` element that restricts a complex type. This makes it easier to define a restriction of a complex type that has been imported from another namespace, since it is now possible for the restricted type to declare local elements and attributes having the same names as those from the base type.

The reporting of validation errors on `xs:assert` has been improved: if the assertion takes the form `empty(expression)` then the validator will not only report an error if the result of the expression is not empty; it will also identify all the nodes (or atomic values) that were present in the result of the expression, enabling easier detection and correction of the problem. This also works for the expression `not(exp)` provided that `exp` has a static item type of `node()`.

Saxon 9.1 also allows assertions on simple types. The assertion is defined by means of an `xs:assert` element acting as a facet, that is, it is a child element of the `xs:restriction` child of the `xs:simpleType` element. This can be any kind of simple type (atomic, list, or union). The value of the `test` attribute must be an XPath expression. The expression is evaluated with no context item, but with the variable `$value` set to the typed value of the element or attribute. The assertion is satisfied if the effective boolean value of the expression is true. For example, for an atomic type that restricts xs:date, the assertion `<xs:assert test="$value lt current-date()"/>` indicates that the date must be in the past. For a list-valued type, the following assertion indicates that all items in the list must be distinct: `<assert test="count($value) eq count(distinct-values($value))"/>`. The XPath expression is allowed to invoke external Java functions, allowing full procedural validation logic. The XPath expression has access only to the value being validated, it cannot access any nodes in the document. For further details see Assertions on Simple Types.

XML Schema 1.0

Saxon now implements enumeration facets on union and list types as the authors of the specification intended. Although the spec as written has problems (bug 5328 has been raised), the intent is that the enumeration facet as written should be interpreted as an instance of the type being restricted. Previously enumeration facets on union and list types were doing a string comparison on the lexical value.

The reporting of `keyRef` validation errors has been improved. Multiple errors can now be reported in a single schema validation run, and the line number given with the error message reflects the location of the unresolved `keyRef` value, rather than the end of the document as before.

A new configuration option is available to control whether the schema processor takes notice (and attempts to dereference) `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes encountered in an instance document that is being validated. This is available as the named property `FeatureKeys.USE_XSI_SCHEMA_LOCATION` on the `TransformerFactory` and `Configuration` classes, via methods on the S9API and .NET `SchemaValidator` classes, and the XQJ class `SaxonXQDataSource`, and via the `-xsiloc` option on the command line interfaces `Validate`, `Transform`, and `Query`.

New methods have been added to class `com.saxonica.schema.SchemaCompiler` to allow setting of "deferred validation mode". In this mode a sequence of calls on `readSchema()` can be made, followed by a single call on `compile()`. The effect is to defer all generation of the finite state machines used for run-time validation until `compile()` is called. This avoids repeated (and wasted) recompilation of complex types every time new elements are added to a substitution group, or every time a new complex type is derived by extension from an existing type. This facility was developed with XBRL as the primary use case, and has the effect of reducing compilation time for this collection of schema documents from 400 seconds to 560 milliseconds.

When `minOccurs` and numeric `maxOccurs` constraints (other than 0, 1, or unbounded) appear on an element or wildcard particle, Saxon now implements a finite state machine using simple counters to count the number of occurrences, rather than "unfolding" the FSM as previously. This removes the limits on the values of `minOccurs` and `maxOccurs`, as well as the cost in time and memory of handling large finite values of `minOccurs` and `maxOccurs`. The unfolding technique is still used when `minOccurs` and `maxOccurs` appear on other kinds of particle, specifically on sequence or choice groups, or when "vulnerable" repeated element and wildcard particles appear within a model group that can itself be repeated (a particle is vulnerable if all the other particles in the model group are optional). A side-effect of this change is that the diagnostics are more specific when a validation failure occurs.

XSLT 2.0

The warning that is issued when a stylesheet that specifies `version="1.0"` is now suppressed by default when the transformation is run via an API rather than from the command line. The default can be changed by calling `Configuration.setVersionWarning(true)`. User feedback suggests that this warning is often an irritant and there are environments where it is hard to suppress it. The XSLT specification says that the warning SHOULD be produced unless the user has requested otherwise; therefore be informed that calling the Saxon API without setting this switch counts as "requesting otherwise".

Tail call optimization is now implemented for `xsl:next-match` as well as `xsl:call-template` and `xsl:apply-templates`. This caters for mutual recursion involving a mixture of these three instructions.

The `type-available()` function can now be used to check for the availability of Java classes. For example `type-available('jt:java.util.HashMap')` returns true, where the prefix `jt` is bound to the URI `http://saxon.sf.net/java-type`.

The system property `xsl:supports-namespace-axis`, introduced in erratum E14, is now recognized (and returns the value "yes").

In `xsl:number`, the specification classifies characters as alphanumeric if they are in one of the Unicode categories Nd, Nl, No, Lu, Ll, Lt, Lm or Lo. Saxon was previously using the Java method `Character.isLetterOrDigit()` which turns out not to be precisely equivalent to this definition. This has been corrected.

An additional option `-xsd:schemadoc1.xsd;schemadoc2.xsd...` is available on the command line. This supplies a list of additional schema documents to be loaded. These are not automatically available in the static context of the stylesheet, but they are available for use when validating input documents (or result documents). The argument can also be used to supply the schema location of a schema document imported by the stylesheet, in the case where the `xsl:import-schema` declaration refers only to the target namespace of the schema and not to its location.

An additional command line option `-traceout` allows the output from the `trace()` function to be directed to a file, or to be discarded. A corresponding option is available in the API (classes `Controller` and `XsltTransformer`).

A new extension instruction `saxon:iterate` is available experimentally, with subsidiary instructions `saxon:continue`, `saxon:break`, and `saxon:finally`. This is designed partly for easier coding of operations that otherwise require explicit recursion, but mainly to enable streamed processing of input files. For details see saxon:iterate.

The code supporting the creation and testing of patterns, as defined in XSLT, has now been decoupled from the XSLT engine, allowing patterns to be used in a non-XSLT environment (for example, in an XProc processor). Java APIs for invoking this functionality have been added to the `sxpath.XPathEvaluator` and `s9api.XPathCompiler` classes. These APIs compile the pattern into an object that masquerades as an XPath expression; when evaluating this expression, the result is true if the pattern matches the context node, false if it does not.

The AntTransform task, a customized Ant task for invoking Saxon XSLT transformations, is no longer issued as an intrinsic part of the Saxon product, but can be downloaded as a separate package from SourceForge: see https://sourceforge.net/project/showfiles.php?group_id=29872.

XQuery 1.0

The change defined by bug 5083 is implemented: namespace URIs declared in a direct element constructor may contain doubled curly braces ("{{" or "}}") to represent single curly braces, and must not contain curly braces unless they are so doubled.

The streaming copy optimization, previously available only with XSLT, is now available also with XQuery. This allows a subset of XQuery expressions to be evaluated in streaming mode, that is, without building the tree representation of the source document in memory. The facility is available only in Saxon-SA. It can be invoked either using the extension function `saxon:stream()` or by means of the pragma `(# saxon:stream #){ expr }`. Use the `-explain` switch on the command line to check whether the optimization is successful.

An additional option `-xsd:schemadoc1.xsd;schemadoc2.xsd...` is available on the command line. This supplies a list of additional schema documents to be loaded. These are not automatically available in the static context of the query, but they are available for use when validating input documents (or result documents). The argument can also be used to supply the schema location of a schema document imported by the query, in the case where the query refers only to the target namespace of the schema and not to its location.

For consistency, the command line now allows the file name containing the query text to be specified using the option `-q:filename`, and an inline query can be specified using `-qs:querytext` (place the option in quotes if the query text contains spaces). The existing convention of specifying the query file as the last option before any query parameters will continue to work.

An additional command line option `-traceout` allows the output from the `trace()` function to be directed to a file, or to be discarded. A corresponding option is available in the API (classes `DynamicQueryContext` and `XQueryEvaluator`).

The native API for XQuery, specifically the `StaticQueryContext` object, now allows external variables to be declared, as an alternative to declaring them in the Query prolog. This facility has not yet been added to S9API, pending user feedback.

The extension `saxon:validate-type` now allows validation of attribute nodes (the expression may be a computed attribute constructor). This was previously accepted in the syntax, but had no effect.

The ability to compile queries into Java source code has been extended, in that some constructs that were not previously supported can now be compiled. Some restrictions remain. One important enhancement is that many calls to Java extension functions can now be compiled. This is particularly beneficial because it means that these calls no longer rely on Java reflection, which gives the opportunity for a substantial performance improvement.

XQJ (XQuery API for Java)

Saxon's implementation of XQJ has been brought into line with the 1.0 "final draft" released on 20 Nov 2007. The final draft includes a set of compatibility tests, and Saxon now passes all these tests, with the exception of four tests whose results have been queried as they appear inconsistent with the documentation.

This change means that the XQJ interfaces now have their proper package names (`javax.xml.query.*` in place of `net.sf.saxon.javax.xml.query.*`, and applications will need to be updated accordingly. The interfaces together with the implementation classes are in the JAR file saxon9-xqj.jar .

To pass the test suite, quite a few changes were necessary, but few of these will affect applications. Many of them concern the error behaviour when null arguments are passed to methods, or when operations are attempted on a connection that has been closed - neither of which are things that real applications are likely to do.

One more noticeable change is that the rules for processing a forwards-only sequence are now strictly enforced. These rules prevent you accessing the same item in the result of a query more than once, unless you save a local copy using the `getItem()` method.

As required by the licensing conditions for XQJ, a conformance statement is provided.

The XQJ implementation now allows a Java object to be supplied as an "external object", whose instance methods can be invoked as external functions within the query. TestL in the sample application `XQJExamples.java` provides an illustration of how to do this. It is necessary to create an `XQItemType` representing the type of the item: this will always have a name whose URI is `http://saxon.sf.net/java-type` and whose local name is the qualified name of the Java class. The standard XQJ method `createItemFromObject()` will recognize a type name in this form and handle the object accordingly.

A number of named properties have been implemented to allow Saxon's behavior to be configured via the `setProperty()` method of `XQDataSource`; for details see the Javadoc of the Saxon implementation class `SaxonXQDataSource`. For configuration settings that are not in this list, `SaxonXQDataSource` provides a method `getConfiguration()` that allows access to the underlying `Configuration` object.

A Saxon-specific method has been added to the `SaxonXQConnection` class, which is Saxon's implementation of `XQConnection`. The method `copyPreparedExpression()` allows an `XQPreparedExpression` created under one connection to be copied to a different connection. This makes it much easier to compile a query once and run in concurrently in multiple threads, a facility that is notoriously absent from the XQJ specification. When a prepared expression is copied in this way, the compiled code of the query and the static context are shared between the two copies, but each copy has its own dynamic context (for query parameters and suchlike).

There have been changes to the implementation of XQJ methods involving input or delivery of results using the Stax `XMLStreamReader` interface. These methods are now implemented using the "pull events" mechanism introduced in Saxon 9.0, replacing the older `PullProvider` interface.

A new `EventIterator` is available for use in the event-pull pipeline, namely `NamespaceMaintainer`. This maintains the namespace context (it acts as a `NamespaceResolver`). This is used by XQJ when delivering results in the form of a StAX `XMLStreamReader`, underpinning the methods on the `XMLStreamReader` interface that allow namespace prefixes to be resolved.

S9API

A new method `Processor.writeXdmValue(XdmValue, Destination)` has been added, allowing any XDM value (for example, a document node) to be written to any `Destination` (for example, a `Serializer`, a `Validator`, or a `Transformer`). For usage examples, see the S9APIExamples demonstration program.

A new constructor has been added to `XdmValue` allowing an `XdmValue` to be constructed from a sequence of items.

A new interface `MessageListener` is available. A user-written implementation of `MessageListener` may be registered with the `XsltTransformer` to receive notification of `xsl:message` output. The class has a single method `message()`, which is called once for each message; the parameters include the message content (as an XML document), a boolean indicating whether `terminate="yes"` was requested, and a `SourceLocator` to distinguish which `xsl:message` instruction generated the output.

New methods have been added to `ItemTypeFactory` to allow creation of ItemTypes and XdmAtomicValues representing external Java objects, which can be passed to a stylesheet or query for use in conjunction with external functions (extension functions). An example of how to achieve this has been added to the sample application `S9APIExamples.java` (test QueryF).

New methods are provided on the `XQueryCompiler` allowing the query to be supplied as a `File`, as a `Reader`, or as an `InputStream`. Methods are also provided to specify the encoding of the query source text.

New methods are available on classes `XQueryEvaluator` and `XsltTransformer` to allow the output from the `trace()` function to be directed to a specified output stream, or to be discarded.

A new method is available on the `XPathCompiler` class to import a schema namespace for reference within the body of the expression.

New methods are available on `XQueryExecutable` and `XPathExecutable` to obtain the result type of the query or expression, as determined by static analysis.

New methods are available on `ItemType` and `OccurrenceIndicator` to determine whether one type subsumes another.

These methods make it easier to write application code that can handle query results when the type of the result is not known .

A new method is available on `XPathCompiler` allowing XSLT patterns to be compiled. A pattern is treated as an expression that returns true if the pattern matches the context node, false otherwise.

A new method is available on `XPathSelector` to return the effective boolean value of the XPath expression.

JAXP

Saxon now supports the `nextSibling` property of a `DOMResult`, introduced in JDK 1.5. This property allows you to specify more precisely the insertion point for new data into an existing DOM tree.

All `TransformerFactory` features that accept a `Boolean` value now also accept the string values "true" and "false". This is useful when the value is set from a configuration file that only permits strings to appear. Many properties that expect the value to be a user-written callback now have an alternative that allows the class name to be supplied as a string, rather than supplying the instance itself. Other properties that expected a symbolic constant have been supplemented by a method that accepts a string. This change also affects the underlying methods in the `Configuration` class.

In Saxon's implementation of the JAXP `Validator` and `ValidatorHandler` interfaces, validation errors (failure of an instance to conform to a schema) are now reported using the `error()` method of the `ErrorListener`, rather than the `fatalError()` method as previously. This means it is normally possible to report more than one error during a single run. Although JAXP does not specify this behavior explicitly, it brings Saxon into line with the reference implementation. (However, one difference with the reference implementation remains: at the end of the validation run, Saxon throws an exception if any validation errors have been reported, whereas Xerces exits as if validation were successful.)

The Configuration property `FeatureKeys.STRIP_WHITESPACE` now affects the result of an `IdentityTransformerHandler`. Previously the setting `ignorable` affected the result, but the setting `all` did not.

A new kind of `Source` is available, the `EventSource`. This represents a source of `Receiver` events, in much the same way as a `SAXSource` represents a source of SAX events; except that it is the `EventSource` itself that supplies the events, not some parser contained in the Source object. This is an abstract class that can be subclassed by user applications; it defines a method `send(Receiver out)` that is called to generate the Receiver events. The particular use case motivating the introduction of this class was a streaming transformation where the input was programmatically generated by the application; this was achieved by having the `URIResolver` return an `EventSource` to generate the events, which the streaming transformation then filtered. Generating `Receiver` events directly proved to be 10-20% faster than generating SAX events.

Extensibility

In addition to the recommended and documented URI format `java:com.package.name.ClassName` for use when calling Java extension functions, Saxon has always supported other formats for compatibility with other XSLT processors. Specifically, Saxon accepted any URI containing the package and class name after the last "/", or the entire URI if there is no "/". On XQuery in particular this causes unnecessary failed attempts to locate corresponding classes on the classpath when in fact the namespace URI of a function is that of an imported module. Two changes have been made to solve this problem: firstly, imported modules are now searched before attempting to dynamically load a Java class. Secondly, when running XQuery or XPath, the only URI format now recognized for Java extension functions is the recommended form `java:com.package.name.ClassName`. Other formats remain available in XSLT to allow a level of compability with other Java-based XSLT processors.

The rules for type conversions when calling extension functions have been aligned more closely with the XPath 2.0 rules, rather than the more flexible XPath 1.0 rules. For example it is no longer possible to supply a string as an argument value when calling a method that expects an integer or a boolean (but supplying an untypedAtomic is fine). Similarly, it is not possible to supply a boolean when a string is expected. Instead, such conversions must now be done explicitly. The implementation now does more static type checking, resulting in more errors being detected at compile time and in greater run-time efficiency as in many cases decisions on which conversions to perform are now made at compile time rather than at run-time.

There may be cases where this type checking causes things to fail that previously worked. One example is where an extension function declares a return type of `java.lang.Object`, and the application then tries to use the returned object in a context where a string (say) is required. In this situation an explicit cast to string (or a call to the string() function) may be required.

It is now possible in XQuery to compile queries (generating Java source code) that contain calls to extension functions. There are still some restrictions (not all argument and result types are handled).

As part of these changes, the `ExternalObjectModel` interface has been redesigned; developers of integration modules for third-party object models (like JDOM, XOM etc) will need to implement a couple of additional methods and can delete a number of existing methods that are no longer used.

It is now possible to call Java methods that expect a JDOM or DOM4J node as their argument, provided that the actual node passed as a parameter from the query or stylesheet is a wrapper around a JDOM or DOM4J node respectively. Returning JDOM or DOM4J nodes from an extension function remains unsupported.

Extensions

An optional second argument is available for `saxon:expression()`. This allows the namespace context for the XPath expression to be supplied explicitly (by default, it is taken from the namespace context of the calling query or stylesheet). The namespace context is supplied in the form of an element node, whose in-scope namespace bindings are used to declare namespace prefixes available for use within the expression. One way to use this is to supply the element node that actually contains the XPath expression; another way is to construct an element specially for the purpose.

A new extension function `saxon:in-summer-time()` is available. It determines whether a given date/time is in summer time (daylight savings time) in a given country or named timezone, to the extent that this information is available in the Java timezone database.

Saxon's support for XOM is extended: there is now a `net.sf.saxon.xom.XOMWriter` class that allows the output of a query or transformation to be captured directly in the form of a XOM document. The class implements `Receiver` and `Result`; an instance of this class can therefore be used directly as an argument to the JAXP `Transformer.transform()` method or the `XQueryExpression.run()` method.

A new extension function `saxon:unparsed-entities()` is available, allowing the application to obtain a list of the names of the unparsed entities declared in a document.

The elements in the SQL extension now by default use the namespace `xmlns:sql="http://saxon.sf.net/sql"`. They can however be registered under a different namespace if required. The extension is not available unless it is registered with the Configuration, either using the API `Configuration.setExtensionElementNamespace()`, or using a configuration file. Either way, Saxon-PE or higher is required (although the extension itself is open source code).

The implementation of the `sql:insert` and `sql:update` extension instructions has been simplified. This may result in some error conditions being detected that were not detected before. The names of tables and columns supplied to these instructions are now validated against SQL rules; if the names are not valid against SQL rules and are not already quoted then they will be enclosed in double-quotation marks.

The `saxon:path()` extension function now accepts a node as an argument. Previously it worked only on the context node.

Diagnostics and Tracing

If requested using the `-l` (letter ell) option on the command line (or the equivalent in the API), Saxon now maintains column numbers as well as line numbers for source documents. The information is available to applications using a new extension function `saxon:column-number()`, or at the level of the Java API via a new method on the `NodeInfo` interface, `getColumnNumber()`. (Third-party implementations of `NodeInfo` will need to implement this method; by default it can return -1). Note that the information is only as good as that supplied by the XML parser: SAX parsers report for an element the line and column of the ">" character that forms the last character of the element's start tag.

Errors that occur during schema validation of an input document now display both line number and column, as do static errors detected in a stylesheet or schema. Dynamic errors occurring during expression evaluation still contain a line number only.

After a dynamic error, Saxon now outputs a stack trace - that is, a representation of the XSLT or XQuery call stack. This feature is now available in Saxon-B, it was previously only in Saxon-SA. The stack trace has been improved at the same time (it now shows changes to the context item made by `xsl:apply-templates` or `xsl:for-each`).

The information in the stack trace is also available programmatically through the method `iterateStackFrames()` on the `XPathContext` object.

The formatted print of the stack trace can be retrieved as a string from within a query or stylesheet using the new extension function `saxon:print-stack()`.

There are some internal changes as a result of this development, which may be noticeable to applications that do debugging or tracing. The `InstructionInfoProvider` interface has disappeared; instead all expressions (including instructions) now implement `InstructionInfo` directly, as do container objects such as `UserFunction` and `Template`. Generally the `getProperties()` method of `InstructionInfo` is not so well supported; applications requiring properties of expressions should cast the `InstructionInfo` to the required class and get the information directly from the expression tree.

The `InstructionInfo` object no longer contains a `NamespaceResolver` - it is no longer needed because all names are now represented as expanded names.

New methods are available to allow the output from the `trace()` function to be directed to a specified output stream, or to be discarded.

A new option `FeatureKeys.TRACE_LISTENER_CLASS` allows the TraceListener to be nominated as a class name, rather than as an instance of the class. This is useful in environments such as Ant where the values of configuration properties must be supplied as strings. A new instance of the class is created for each query or transformation executed under the Configuration. The existing option `FeatureKeys.TRACE_LISTENER` remains available.

Saxon on .NET

Saxon on .NET is now built using version 0.36.0.11 of IKVMC. The main difference this makes is that the class library used is now the OpenJDK class library rather than the GNU Classpath library. This in turn has different licensing conditions.

There is now an option `processor.SetProperty("http://saxon.sf.net/feature/preferJaxpParser", "true")` whose effect is to cause Saxon to use the XML parser in the OpenJDK class library in preference to the (Microsoft) `System.Xml` parser. This is useful when the stylesheet or query uses the `id()` or `idref()` function when attribute types are defined in a DTD: the Microsoft XML parser does not report attribute types to the application, so these functions fail to find anything when the source document was built using this parser. Using the JAXP parser gets around this problem.

Saxon on .NET is now built and tested on .NET 2.0. It should be compatible with .NET 1.1 or .NET 3.5, but this cannot be guaranteed. Saxon is not tested on Mono, though users have reported running it successfully.

New constructors have been added to the class `DomDestination`, allowing new content to be attached to an existing document, document fragment, or element node.

A new method is available on the `XPathCompiler` class to import a schema namespace for reference within the body of the expression.

A new method `Processor.WriteXdmValue(XdmValue, XmlDestination)` has been added, allowing any XDM value (for example, a document node) to be written to any `XmlDestination` (for example, a `Serializer`, a `Validator`, or a `Transformer`).

The `WriteTo` method on `XdmNode` has been changed so it will write to any `XmlWriter`, not only an `XmlTextWriter` as before.

A new property `MessageListener` has been added to the `XsltTransformer` object. This allows the output of <xsl:message> instructions to be intercepted. Each call of <xsl:message> generates a document node, which is passed in the form of an `XdmNode` to the supplied message listener. Additional parameters indicate whether the <xsl:message> instruction specified `terminate="yes"`, and the location in the stylesheet of the originating <xsl:message> instruction.

The `XmlResolver` supplied as a property of various classes including the `DocumentBuilder`, the `XsltCompiler`, the `XsltTransformer`, and the `XQueryEvaluator`, is now used not only when resolving URIs at the Saxon level (for example in calls to the `doc()` function or in `xsl:import` and `xsl:include`), but also by the XML parser in resolving URIs referring to external entities, including an external DTD. Note that this means it is unwise to return anything other than a `Stream` from the `GetEntity()` method, since this is the only return value that the Microsoft `XmlTextReader` can handle.

Extension functions (external functions) may now use `System.Xml.XmlNode` as an argument type, provided that the node that is actually passed in the call is a Saxon wrapper around an `XmlNode`. Similarly, an `XmlNode` may also act as the return type. This also applies to subtypes of `XmlNode`, and to arrays of `XmlNode`. However, this facility is only available when Saxon is invoked via the .NET API, not when it is invoked from the command line. Note that returning `XmlNode` values may be expensive if the extension function is called frequently, as new wrappers are created each time; the calling stylesheet or query should also not rely on the identity of nodes that are returned in this way.

Extension functions (external functions) may also use the types `Saxon.Api.XdmValue` and its Saxon-defined subtypes as an argument or return type. This facility is only available when Saxon is invoked via the .NET API, not when it is invoked from the command line.

New methods are available to allow the output from the `trace()` function to be directed to a specified output stream, or to be discarded.

The sample applications for .NET have been rewritten, and the test drivers for the W3C XQuery and XSLT test suites have been repackaged within a simple forms-based application called `TestRunner.exe`.

In previous releases the documentation stated that the SQL extension was untested on .NET. This time I tried it and found it wasn't working, probably due to the absence of JDBC drivers in the OpenJDK class library. In Saxon 9.1 I have therefore excluded the relevant classes from the .DLL build. It would make more sense on .NET to implement this extension directly over the .NET data access classes.

The tooling for creating the API documentation on .NET has changed. The NDOC tool, which was used in previous releases, is no longer maintained (or usable), while the promised Microsoft replacement, SandCastle, is unfinished and poorly documented, and I couldn't get it to work. I ended up writing my own documentation generator in XSLT, taking the C# source code and the generated apidoc.xml documentation as input. There are few things missing in the resulting HTML, for example there is little information about inherited methods, but I think that what is there is more easily accessible (it follows the Javadoc style of putting all the information about one class on one page).

Internal APIs

The three methods `isId()`, `isIdref()`, and `isNilled()` have been moved from the `ExtendedNodeInfo` interface into `NodeInfo`, which means they must now be implemented by all concrete classes implementing `NodeInfo`. The `ExtendedNodeInfo` interface has been dropped.

The `SequenceIterator` interface now has a `close()` method. This should be called by any consumer of iterator events if the iterator is not read to completion. Currently the only effect is where the events derive from streamed processing of an input document; in this case the `close()` call causes the parsing of the input document to be abandoned. This means for example that an expression such as `exists(saxon:stream(('doc.xml')//x))` will stop parsing the input document as soon as an <x> element is found. Any user-written implementations of `SequenceIterator` must be changed to provide an implementation of this method; it can safely do nothing, but if the `SequenceIterator` uses another `SequenceIterator` as input, the call to `close()` should be passed on.

To allow further application control over dynamic loading in environments with non-standard class loaders or other specialist requirements, dynamic loading of classes (and instantiation of these classes) is now delegated to a new `DynamicLoader` class owned by the `Configuration`. This allows the application to substitute a subclass of its own to intercept the calls that cause classes to be loaded dynamically.

The classes used to represent path expressions have been refactored (leading to some change in -explain output). The two classes `PathExpression` and `SimpleMappingExpression`, which contained a lot of repeated code, have been replaced by a structure in which the general class `SlashExpression` has two subclasses, `PathExpression` and `AtomicMappingExpression`, for use when the rhs operand of "/" is known to deliver nodes or atomic values respectively. The expression parser initially generates a `SlashExpression`, and this is replaced by a `PathExpression` or `AtomicMappingExpression` if possible during the type checking phase. If the type of the rhs cannot be determined, the `SlashExpression` is retained as a concrete class and is evaluated at run-time. The new `PathExpression` class is not responsible for sorting and deduplicating nodes; when a `PathExpression` is created, it is always wrapped in a `DocumentSorter` that has this responsibility, and the `DocumentSorter` is subsequently removed if the path expression is found to be presorted, or if sorting is found to be unnecessary because of the context where the path expression is used.

Serialization

A new serialization attribute `saxon:double-space` is available. The value is a space-separated list of element names; the effect is to cause an extra blank line to be inserted in the output before the start tag of these elements. The option has no effect unless `indent="yes"` is set.

Saxon now honours a request to output using UTF-16 with no byte-order-mark. Perviously a byte-order-mark was inserted (by the Java IO library) whether requested or not.

Optimization

The XQuery expression `for $x at $p in EXPR return $p` is now rewritten as `1 to count(EXPR)`.

A filter expression that filters a constant sequence is now evaluated at compile time, provided the predicate does not use any variables and has no dependencies on the dynamic context other than the context item, position, and size.

"Loop-lifting" (extraction of subexpressions to prevent them being repeatedly evaluated within a loop) is now extended to XQuery `for` expressions that have a position variable.

Adjacent literal text nodes in the content of an element or document constructor are now merged at compile time. (These can arise as a result of early evaluation of expressions in the content sequence.)

The `AttributeValidator`, which checks whether required attributes are present and expands default values during schema validation, has been rewritten for efficiency, to do most of the work of setting up the necessary data structures at schema compile time rather than on a per-element basis at validation time. It also uses two different implementations of the main data structure to handle the typical case with a small number of attributes, and the more difficult but unusual case where large numbers of attributes are declared.

When an attribute has an enumeration type, space is saved on the instance tree by using references to the attribute value as held in the compiled schema, avoiding holding multiple copies of the same string.

In XSLT, multiple identical key definitions are now merged. These can arise when the same stylesheet module is imported several times in different places. Previously, this led to the construction of multiple indexes, whose results were merged at run-time.

The set of XPath expressions for which streamed evaluation is possible (using `saxon:stream()` or equivalent interfaces) has been slightly extended. It can now include expressions that return a union of elements and attributes; previously it was required to return exclusively elements, or exclusively attributes. It now allows multiple predicates (previously only a single predicate was allowed).

In XSLT, tail-call optimization is now performed for a call on `xsl:call-template` that appears within an `xsl:for-each` instruction, provided that it can be statically determined that the select expression of the `xsl:for-each` returns at most one item.

Version 9.0 (2007-11-03)

- Highlights
- New Java API
- Command line changes
- XSLT changes
- XPath changes
- Extensions
- Schema-related changes
- Changes to existing APIs
- Pull processing in Java
- Serialization
- Localization
- Optimization
- Diagnostics
- NamePool changes
- Expression tree changes

Highlights

This page lists some of the most important new features introduced in Saxon 9.0

1. There is a new Java API, called . Existing APIs remain supported.
2. The interfaces have received a revamp, while retaining backwards compatibility for most options.
3. The schema processor now supports , as defined in XML Schema 1.1.
4. A new extension function allows in XQuery.
5. It is now possible to (the schema component model) as XML.
6. There is a new model for of queries., improving the ability to integrate into a pull-based pipeline architecture.
7. The latest draft of the specification (XQuery API for Java) is implemented
8. Number and date formatting has been added for a number of including Belgian French, Flemish, Dutch, Danish, Swedish, and Italian
9. A number of new have been introduced. These include function and variable inlining, wider use of automatic indexing, wider use of tail call optimization, hashing for large xsl:choose expressions, and a speed-up of the DOM interface.
10. analyzes a query and discards the parts of the source tree that are not needed to answer the query, giving a significant saving in tree-building time and memory.
11. Optimized can now be output ("explained") in an XML format, making it amenable to processing or graphical rendition.

Please note that queries compiled into Java code are not backwards-compatible at this release; they must be recompiled.

New Java API

A brand new Java API is available in package `net.sf.saxon.s9api` (pronounced "snappy", stands for Saxon 9 API). This is closely modelled on the successful .NET API. The design aims were to:

- Provide an API that is well documented and uncluttered by implementation detail
- Provide uniformity across XSLT, XPath, and XQuery (for example, allowing XSLT and XQuery to be combined naturally in a single pipeline)
- Maximise the ability to create compiled objects that are reusable across multiple threads
- Reuse existing JAXP classes and interfaces where appropriate, but abandon them when they no longer deliver the right functionality
- Provide an accurate representation of the XDM data model
- Provide access to all mainstream Saxon functionality needed by typical users
- Provide escape hatches to underlying implementation classes and methods for power users
- Exploit new Java 5 features such as type-safe enumerations, generics, and iterables, while retaining support for JDK 1.4 users

The new API does not replace any existing APIs: it is a separate layer on top of the Saxon product. It is provided in a separate JAR file, saxon9-api.jar, and can be used only with Java 5. The main Saxon jar files, however, continue to work with JDK 1.4.

For details see the Javadoc: go to package `net.sf.saxon.s9api` and start with the `Processor` class.

Saxon will continue to support standardized interfaces such as JAXP and XQJ. However, JAXP has become cluttered and inconsistent, and is still missing support for many XSLT 2.0 features. XQJ has been designed with a client-server architecture in mind, and while it is usable with Saxon, it is not really optimized for the kind of applications where Saxon is deployed. For users who want to exploit Saxon fully in their applications, rather than retaining portability across XSLT/XPath/XQuery implementations, the new API offers a much cleaner choice.

Command line changes

There has been a significant reorganization of the command line parameters for the commands used to control XSLT and XQuery, though all commonly-used options should continue to work unchanged. Options now generally take the form `-keyword:value`, with the values "on" and "off" for boolean options. Groups of related options have been combined, so `-ds` and `-dt` are now `-tree:linked` and `-tree:tiny`. For options that were previously in the form "-keyword value", the old form (with a space) is still accepted for the time being. Details of the commands are at Running XSLT from the Command Line and Running XQuery from the Command Line.

The option `-x:classname` has been added for XQuery to set the parser class name used for source files, this is to allow use of a catalog resolver.

XSLT changes

The standard URI resolver for the unparsed-text() function now follows the spec more closely by examining the media type of the retrieved resource (which will typically be available only if it is obtained via an HTTP connection). In this case, if the media type indicates XML, the encoding is inferred as described in the XML specification, ignoring any encoding that was requested in the second argument. The URI resolver now also has a debug flag allowing tracing of the process of connection, examination of HTTP headers, and inference of an encoding.

The system-property() function has changed so that, with the exception of properties in the XSLT namespace, the function is always evaluated at run-time rather than at compile time. This is because the values of system properties in the execution environment may differ from their values at compile time. When the configuration property ALLOW_EXTERNAL_FUNCTIONS is set to false access to system properties is now disabled: this is because some of them, such as `user.dir`, could be considered to compromise security.

The implementation of format-dateTime() and similar functions has changed in the way it handles the `[ZN]` component. This requests the time-zone as a named timezone. This is problematic, because the information is not really available: the data type maintains only a time zone offset, and different countries (time zones) use different names for the same offset, at different times of year. If the value is a date or dateTime, and if the country argument is supplied, Saxon now uses the Java database of time zones and daylight savings time (summer time) changeover dates to work out the most likely timezone applicable to the date in question. This is still a guess, but it is now more likely to be right.

If the timezone is formatted as [ZN,6] (specifically, with a minimum length of 6 or more) then the Olsen timezone name is output (again, this requires the country to be supplied). The Olsen timezone name generally takes the form `Continent/City`, for example "Europe/London" or "America/Los_Angeles". If the date/time is in daylight savings time for that timezone, an asterisk is appended to the Olsen timezone name.

The formatting of noon/midnight in the 12-hour clock has been changed. Although in the USA it seems to be fairly common (though not universal) to represent noon as "pm" and midnight as "am", this usage does not appear to be widely accepted in the UK. Therefore, if a maximum width of 8 or more characters is specified (thus: `[Pn,*-8]`), noon and midnight will now be written in full.

Saxon now implements the rule that within a single stylesheet module, all the `use-when` expressions will see stable values for functions such as `current-dateTime()`.

The implementation of `xsl:copy-of` has changed so that when a document or element node is copied, if line numbering was requested at the configuration level, then line numbers of element nodes are copied from the source document to the result document. For other instructions that create elements, the line number of the resulting element will reflect the position in the stylesheet or query of the instruction that created the element. The line number is accessible using the saxon:line-number() function. When running from the command line, this works only if the `-l` option is specified, or if a trace listener is in use.

The custom Ant task for running Saxon has been fleshed out with additional attributes to give greater control over the transformation process. Detail are here.

XPath changes

Static type checking has become a little stronger. One change is that in a conditional expression (which includes `xsl:choose` in XSLT and `typeswitch` in XQuery, each branch of the conditional must satisfy the required type, as must the default branch if there is a default.

In regular expressions, a back-reference can no longer appear within a character class expression (thus "(abc|def)[\1]" is now illegal). This change has been agreed as an erratum by the W3C working groups, on the grounds that such an expression is meaningless.

In regular expressions, the interpretation of the character class escapes [\c] and [\i] is now sensitive to the selected version of XML in the configuration: if the configuration is set to XML 1.1, then the XML 1.1 definitions of `Letter` and `NameChar` are used in place of the XML 1.0 definitions. Internally, the routines for classifying characters according to their validity in different contexts in XML 1.0 and XML 1.1 have been reorganized: the data tables for XML 1.0 and XML 1.1 have been merged into a single table, and this is now used also by the regular expression routines to support the `\c` and `\i` character class escapes. One side-effect of this change is that Saxon now includes no Apache code, which slightly simplifies the license conditions.

The implementation of durations has changed to use a two-component model (months and seconds) rather than a six component model (years, months, days, hours, minutes, seconds). Previously Saxon was capable of maintaining unnormalized durations (for example 18 months) but there were no longer any XPath functions or operators that relied on this. The change raises some implementation-defined limits. Some operations that break the implementation-defined limits may now be detected rather than causing incorrect results. The change may also affect Java applications that relied on maintaining durations in unnormalized form.

The implicit timezone is now a genuine part of the dynamic context. In previous releases, it was a property of the `Configuration`, which meant it was known at compile time. The effect of this is that a compiled query or stylesheet can now be used across a change of timezone. As a result, operations that depend on the implicit timezone can no longer be evaluated statically, even if the operands are known.

The rule in `min()` and `max()` that the returned value should be an instance of the lowest common supertype of the values in the input sequence is now correctly implemented. For example, `max((xs:unsignedInt(3), xs:positiveInteger(2)))` returns the value 3 with type label `xs:nonNegativeInteger`.

When casting from a value other than xs:string to a user-defined type on which a pattern facet is defined, the specification requires the canonical lexical representation of the target value to be valid against the facet. In previous releases Saxon was checking not the canonical lexical representation as defined in XML Schema, but the result of casting to a string as defined in XPath. This has now been fixed. Cases where the two are different include: xs:decimal when the value is a whole number; xs:double and xs:float; xs:dateTime and xs:time when the value is in a timezone other than Z; xs:date with a timezone outside the range -12:00 to +11:59.

Extensions

When an extension function has a declared return type of `java.lang.Object` in Java, or `System.Object` in .NET, the inferred static type is now `xs:anyAtomicType` rather than `external object`. This means that if the value actually returned at run-time is, say, an integer, and the value is used where an integer is required, then the call will not be rejected as a compile-time type error.

The extension function `saxon:transform()` has changed: in the third argument, which is used to pass parameters to the stylesheet, the argument must now be a sequence of element nodes, attribute nodes, or document nodes. If a document node is supplied, it is treated as equivalent to passing its element children.

Two new extension functions `saxon:compile-query()` and `saxon:query()` are available. They allow a query to be constructed dynamically as a string, compiled, and repeatedly executed. The input to the function can (if required) be created using the new `saxon:xquery` output method.

A new extension function `saxon:result-document()` is available in Saxon-SA, for the benefit of XQuery users. Modelled on the `xsl:result-document` instruction, it allows a query to generate multiple output documents and serialize them to filestore.

The implementation of the EXSLT `math:power()` function has been extended to cater for numeric data types other than xs:double. For example, the result of `math.power(2, 128)` is now calculated using integer rather than double arithmetic, and the result is an integer. For the full rules, see EXSLT

In Saxon's implementation of the EXSLT dates-and-times library, the current date/time that is used is now aligned with the XPath 2.0 `current-dateTime()` function, and thus returns the same value for the duration of a query or transformation.

The extension function `saxon:typeAnnotation()` now accepts any item. For a node it returns the type annotation; for an atomic value it returns the type label. The return type is now xs:QName (it was previously a lexical QName returned as a string).

The extension functions `yearMonthDurationFromMonths()` and `dayTimeDurationFromSeconds()`, which have been undocumented since Saxon 8.1, have finally been dropped. The same effect can be achieved by multiplying a duration of one month or one second by the appropriate integer.

The extension function `saxon:tokenize()`, also undocumented since Saxon 8.1, has been dropped. Use `fn:tokenize()` instead.

Three new extension functions are available to find when a file was last modified. Two variants of `saxon:last-modified()` test the last modified date of the file from which a node was obtained; the function `saxon:file-last-modified()` takes an absolute URI as its argument.

Schema-related changes

The command line interface `com.saxonica.Validate` has been completely redesigned, allowing multiple schema documents to be loaded and multiple instance documents to be validated.

This release of Saxon introduces preliminary support for assertions in a schema, based on the current (31 August 2006) draft of XML Schema version 1.1. This allows a complex type to contain an assertion about the content of the corresponding element expressed as an arbitrary XPath 2.0 expression. Please note that this facility in the Working Draft is likely to change, and the Saxon implementation will change accordingly. For further details see Assertions.

The XML Schema specification imposes a rule that when one type R is derived from another type B by restriction, then every element particle ER in the content model of R must be compatible with the corresponding element particle EB in B. One aspect of this is that the identity constraints defined in the declaration of ER (that is, unique, key, and keyref) must be a superset of the constraints defined for EB. The specification doesn't say how to decide whether two constraints are equivalent for this purpose, and Saxon has previously ignored this requirement. At this release a check is introduced which partially implements the rule. Specifically, Saxon will count the number of constraints that are defined, and will report an error if EB has more constraints of any particular kind (unique, key, or keyref) than ER has. If EB has at least one constraint and ER has one or more, then Saxon will output a warning saying that it was unable to check whether the constraints were compatible with each other.

It is now possible when requesting validation of an instance to specify the required name of the top-level element in the document being validated. This is possible through the option `-top:clarkname` on the `com.saxonica.Validate` command, or via a new property on the `AugmentedSource` object. The property is also available on the `DocumentBuilder` in the .NET API and in the new s9api Java API. A validation error occurs if the document being validated has a top-level element with a different name.

I discovered that Saxon allows you to use the types `xs:dayTimeDuration` and `xs:yearMonthDuration` in a schema as built-in types. XML Schema 1.0 doesn't recognize these types (though I can't find a rule that says it is absolutely non-conformant to accept them). I have changed the code to give an interoperability warning if they are used. I have also disallowed the use of the type `xs:anyAtomicType`, which has no defined validation semantics.

The mechanisms for comparing values in the course of schema validation and processing have now been separated completely from the mechanisms used when implementing XPath operators. This means that the semantics of comparison and ordering should now follow the XML Schema specification precisely. Previously some operations were implemented according to the XPath semantics.

A duplicate `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation` attribute is now ignored (previously it was rejected under the rule that such an attribute cannot appear after the first element in the relevant namespace). Duplicates can arise naturally from XInclude processing, so they are now accepted and ignored. The schema specification permits this but does not require it. To be considered duplicates, the declarations must match in the namespace URI and in the absolutized schemaLocation URI.

Result tree validation

Saxon now does more extensive compile-time checking where an `xsl:document` or `xsl:result-document` instruction requests validation of the result tree. This means that validation errors that were previously detected at stylesheet execution time are now sometimes detected at compile time. Previously these checks were only done when validation was requested on an element-constructor instruction.

Expansion of attribute and element defaults

When the input or output of a query or transformation is validated, it is now possible to request that fixed and default element and attribute values defined in the schema should not be expanded. This is done using the option `-expand:off` on the command line, or equivalent options in the `TransformerFactory` and `Configuration` APIs.

The same option also applies to DTD-based attribute default expansion, provided that the XML parser reports sufficient information to the application.

Serializing a Schema Component Model

It is now possible to export the contents of the schema cache held in the `Configuration` object to an XML file (with the conventional extension `.scm` for Schema Component Model). The contents can subsequently be reloaded. This is faster than reloading the original source schema documents, because it allows most of the validation to be skipped, along with the sometimes expensive operation of constructing and determinizing finite state machines. This facility is intended to be used in conjunction with XQuery Java code generation: it allows the schemas that were imported by a compiled query to be saved on disk alongside the compiled query itself, for rapid reloading at run time.

The serialized SCM file is also designed to be easy for applications to process. The representation of schema components is more uniform than in source .xsd documents (there are fewer defaults, and fewer alternative ways of expressing the same information). This makes it a suitable representation for applications that need to process or analyze schema information, as an alternative to using the Java API.

> This has proved useful within Saxon itself. Saxon's schema analyzer was previously written using ad-hoc parsing techniques to validate schemas against the rules defined in the schema-for-schemas. The addition of `assert` and `report` elements threatened to make this even more complex. So a simple XSLT transformation was written to take the finite state machines in the SCM version of the schema-for-schemas and generate Java code from them. This means that Saxon's schema validation logic is now derived directly from the published schema-for-schemas, while retaining the efficiency of hard-coded Java.

Changes to the Schema Component Model API

Changes have been made to the API for the schema component model (package `com.saxonica.schema`) to align it more closely with the abstract model defined in the W3C specifications.

All named components now consistently expose methods `getName()` and `getTargetNamespace()` to provide access to the local part of the name and the namespace URI respectively. The wide variety of existing names for these accessors have been retained for the time being as deprecated methods. The new names are chosen because they correspond to the names used for these properties in the W3C schema component model.

The class `FacetCollection` has disappeared; its functionality has been merged into `UserSimpleType`.

The class `Compositor` has been renamed `ModelGroup`, and its subclasses such as `ChoiceCompositor` have been renamed accordingly. In the W3C schema model, the compositor (all, choice, sequence) is one of the properties of the `ModelGroup`. This is now available using the method `getCompositorName()` on the `ModelGroup` object.

`Particle` is now an abstract class rather than an interface, and the previous abstract class `AbstractParticle` no longer exists. There are three subclasses of `Particle`, namely `ElementParticle`, `ElementWildcard`, and `ModelGroupParticle`. This means there is now a destinction between the `ModelGroupParticle`, which represents a reference to a `ModelGroup`, and the `ModelGroup` itself. The class `ModelGroupDefinition` (which represents a named model group) no longer implements `Particle`; it is now a subclass of `ModelGroup`.

The class `ModelGroupParticle` replaces `GroupReference`; it is no longer necessarily a reference to a (named) `ModelGroupDefinition`, but now can be a reference to any (named or unnamed) `ModelGroup`.

`ElementWildcard` and `AttributeWildcard` are no longer subclasses of `Wildcard`; instead `Wildcard` is now a helper class to which these two classes delegate. Instead, `ElementWildcard` is now a subclass of `Particle`. The `getTerm()` method of `ElementWildcard` returns the `Wildcard` object (previously it returned the `ElementWildcard` object itself).

The use of exceptions `SchemaException` and `ValidationException` has been made more consistent. A `SchemaException` indicates that the schema is invalid, and should occur only while the schema is being loaded and validated. A `ValidationException` indicates that an instance document is invalid against the schema, and should occur only during instance validation. Errors relating to the consistency of a stylesheet or query against a valid schema should result in an `XPathException` being thrown. An inconsistency in the schema found during instance validation is an internal error, and should result in an `IllegalStateException`, except for unresolved references to missing schema components (which is defined in the schema spec not to constitute a schema invalidity), which results in an `UnresolvedReferenceException`. Because it can occur almost anywhere, `UnresolvedReferenceException` is an unchecked exception.

Changes to existing APIs

The behavior of `configuration.buildDocument()` has changed for cases where the supplied `Source` object is a tree. In particular, if it is a `DOMSource` then the DOM Document node will normally be wrapped in a Saxon wrapper object rather than being copied to a TinyTree. This has the effect of reinstating the pre-8.9 behaviour of methods in the XPath API that are given a DOMSource as an argument; the XPath expressions will now return nodes in the original DOM tree rather than copies of these nodes.

Support for DOM Level 2 implementations has been reinstated; however Saxon no longer attempts to detect whether the DOM supports level 3 interfaces; instead, when a Level 2 DOM implementation is used, the configuration setting `config.setDOMLevel(2)` must be used. (Saxon now has compile-time references to DOM level 3 methods, but will not call such methods if this configuration setting is in force.)

The class `StaticQueryContext` has been split into two: user-facing methods used to initialize the context for a query are still in `StaticQueryContext`, but methods intended for system use, which update the context as declarations in the query prolog are parsed, have been moved to a new class `QueryModule`. The class `StaticQueryContext` no longer implements the `StaticContext` interface.

As part of the above change, some methods on `StaticQueryContext` have been dropped. This notably includes the method `declareVariable()` which never worked in a satisfactory way because the variable was not reusable across multiple queries. External variables should always be declared from the query prolog.

A new factory method `Configuration.makeConfiguration()` is available. This creates a schema-aware configuration if Saxon-SA is installed and licensed, otherwise it creates a non-schema-aware configuration. This option is useful for applications that want to take advantage of the enhanced Saxon-SA optimizer in cases where it is available, but do not otherwise depend on Saxon-SA functionality.

A new method on `Configuration` is introduced to copy a `Configuration`. The main motivation for this is to eliminate the high cost of repeatedly checking the Saxon-SA license key in applications that create many separate Configurations.

The rule that all documents used within a single query, transformation, or XPath expression must be built using the same `Configuration` has been relaxed slightly, so the requirement is only that they must be "compatible" Configurations, which means in practice that they must use the same `NamePool` and `DocumentNumberAllocator`. Although the rule has been relaxed slightly, it is also now enforced on a number of interfaces where previously no checking took place (which could lead to unpredictable failures later). This applies in particular to XPath APIs.

A new option is available in the `Configuration` to indicate that calls to the `doc()` or `document()` functions with constant string arguments should be evaluated when a query or stylesheet is compiled, rather than at run-time. This option is intended for use when a reference or lookup document is used by all queries and transformations. Using this option has a number of effects: (a) the URI is resolved using the compile-time URIResolver rather than the run-time URIResolver; (b) the document is loaded into a document pool held by the `Configuration`, whose memory is released only when the `Configuration` itself ceases to exist; (c) all queries and transformations using this document share the same copy; (d) any updates to the document that occur between compile-time and run-time have no effect. The option is selected by using `Configuration.setConfigurationProperty()`\ or `TransformerFactory.setAttribute()` with the property name `FeatureKeys.PRE_EVALUATE_DOC_FUNCTION`. This option is not available from the command line because it has no useful effect with a single-shot compile-and-run interface.

A convenience method `QueryResult.serialize(NodeInfo node)` has been provided, allowing a node to be serialized as XML; the result is returned as a String.

There is also a convenience method `Navigator.getAttributeValue(NodeInfo node, String uri, String localName)` making it easier for Java applications to get an attribute of an element.

In the `NodeInfo` interface, the rules for the `copy()` method have changed so that when an element is copied, its namespaces must be output without duplicates (or without a declaration being cancelled by an undeclaration). The method no longer relies on the recipient removing such duplicates.

The method `NodeInfo#sendNamespaceDeclarations` has been deleted.

The class `NameTest` has a new constructor taking a URI and local name as strings, making it easier to construct a `NameTest` for use in calls to `iterateAxis()`. In addition, the abstract class `NodeTest` now has only one abstract method, making it easier to write a user-defined implementation of `NodeTest` for filtering the nodes returned by `iterateAxis()`.

Methods that construct or convert atomic values no longer return ValidationErrorValue in the event of a failure. There were a couple of problems with this mechanism: although it was designed to eliminate the costs of throwing an exception, it failed to take into account the cost of creating the exception before throwing it, which is surprisingly expensive as it involves capturing a stack trace. Secondly, the mechanism wasn't type safe as it didn't force callers to check the return value for an error. These methods (and a number of others) now return a `ConversionResult` which is essentially a union type of `AtomicValue` and `ValidationFailure`. Code calling these methods therefore has to consciously cast the result to `AtomicValue`, preferably after checking that the result is not a `ValidationFailure`.

The two exception classes `StaticError` and `DynamicError` are no longer used: instead, their common base class `XPathExpression` is now a concrete class and is used to represent both static and dynamic errors. It includes a field to distinguish the two cases, though in fact there is very little code that cares about the difference (the only time the difference is significant is when dynamic errors occur during early compile-time evaluation of constant subexpressions). Any application code that contains a catch for `StaticError` or `DynamicError` should be changed to catch `XPathException` instead. Since nearly all API methods declared "throws XPathException" already, this is unlikely to be a major issue.

There has been some tidying up of methods on the `AtomicValue` class, especially methods for converting values between types and for setting the type label.

The .NET API

In the .NET API, the class `XsltCompiler` now has an overload of the `Compile()` method that takes input from a `TextReader` (for example, a `StringReader`).

The class `XdmAtomicValue` now has a static factory method that constructs an "external object", that is, an XPath wrapper around a .NET object. This can be passed as a parameter to a stylesheet or query and used as an argument to extension functions.

The class `TextWriterDestination` (despite its name, which is unchanged) now wraps any `XmlWriter`. It was previously restricted to wrap an `XmlTextWriter`.

The XQJ API

Saxon's implementation of the XQuery API for Java (XQJ) (jsr-225) has been upgraded to conform to the version 0.9 specifications released on 12 June 2007. The specifications can be downloaded at http://jcp.org/en/jsr/detail?id=225. There are some incompatibilities: applications will need to be changed, though probably not extensively.

Please note that this API is still a draft, and Saxon's implementation will change when the specifications change.

Pull processing in Java

The internal mechanisms and API for pull processing have been substantially rewritten in this release.

There is a new method `iterateEvents()` on the `Expression` class, which evaluates the expression and returns its result as a sequence of `PullEvent` objects. A `PullEvent` can be an `Item` (that is, a `NodeInfo` or `AtomicValue`). It can also be a StartElement, EndElement, StartDocument, or EndDocument event. Within the content of a document or element, the child nodes can be represented either as complete nodes (including element nodes, and potentially even document nodes, in which case the document wrapper should be ignored) or as nested event sequences using further StartElement and EndElement events.

An `EventIterator` is also a `PullEvent`, so a stream of events can contain nested streams. If you want to process the events without having to handle this nesting, you can flatten the sequence by calling the static method `EventStackIterator.flatten()`.

To serialize the results of a query, there is a static method `EventIteratorToReceiver.copy()` which reads the events from a pull pipeline (an `EventIterator`), and pushes them to a push pipeline (a `Receiver`). However, if you are serializing the results then it probably makes sense in most cases to evaluate the query in push mode to start with.

A sequence in which all the `PullEvents` are `Item` objects is called a sequence. A sequence in which all document and element nodes are split into their constituent events is called a sequence. You can turn any sequence of PullEvents into a fully composed sequence by wrapping it in a `SequenceComposer`. This will physically construct any document or element nodes that were represented in the sequence in decomposed form. Similarly, you can turn any sequence into a fully decomposed sequence by wrapping it in a `Decomposer`.

Serialization

The serialization property `byte-order-mark="yes"` is now honored when the selected encoding is `utf-16le` or `utf-16be`.

The HTML serialization method now uses named entity references in preference to decimal character references for characters outside the specified encoding, where an entity reference is known. Since the list of known entity references is confined to characters in the range xA0 to xFF, this only really affects the outcome when encoding="us-ascii". More detailed control is available using `saxon:character-representation`, though this may have to be changed in future since it is technically non-conformant - vendor-defined serialization attributes are no longer allowed to cause behaviour that contradicts the provisions of the serialization specification.

Saxon now implements the change to the specification made as a result of bug 3441 [http://www.w3.org/Bugs/Public/show_bug.cgi?id=3441]: with the HTML and XHTML output methods, any generated `<meta>` element is now produced earlier in the serialization pipeline, which has the effect that characters in this element are subject to substitution by means of character maps.

A new serialization method `saxon:xquery` is available. This is intended to be useful when generating an XQuery query as the output of a query or stylesheet. This method differs from the XML serialization method in that "<" and ">" characters appearing between curly braces (but not between quotes) in text nodes and attribute nodes are not escaped. The idea is to allow queries to generated, or to be written within an XML document, and processed by first serializing them with this output method, then parsing the result with the XQuery parser. For example, the document `<a>{$a &lt; '&lt;'}</a>` will serialize as `<a>{$a < '&lt;'}</a>`.

With the XML output method, indentation is now suppressed for any element that is known to have mixed content: specifically, any element that is validated against a user-defined type (not xs:anyType or xs:untyped) that specifies `mixed="true"` in the schema. No whitespace will be added to the content of such an element. For simplicity, the option applies to all the descendants of the element, even if there are descendants that do not allow mixed content.

A new serialization parameter `saxon:suppress-indentation` is introduced for the XML output method. (It does not affect the HTML or XHTML output methods.) The value of the attribute is a whitespace-aeparated list of element names, and it works in the same way as `cdata-section-elements` (for example, values in `xsl:output` and `xsl:result-document` are cumulative). Its effect is that no indentation takes place for the children or descendants of any of the named elements (just as if they specified `xml:space="preserve"`. This option is useful where parts of the output document contain mixed content where whitespace is significant.

Localization

Number and date formatting has been added for Danish (da), Dutch (nl), Swedish (sv), Italian (it), Belgian French (fr-BE), and Flemish (nl-BE). Thanks to Karel Goossens of BTR Services, Belgium, for supplying these.

The class hierarchy has been changed so that non-English numberers no longer inherit from `Numberer_en`, but from a new `AbstractNumberer` class; the functionality that is intended to be generic across languages (but overridable) has been separated from the functionality that is specific to English.

Optimization

When the descendant axis is used in a schema-aware query or stylesheet, and the type of the context node is statically known, the step that uses the descendant axis is now replaced by a sequence of steps using the child axis (and the descendant or descendant-or-self axes, if necessary) that restricts the search to the parts of the tree where the required element can actually be found. If it is not possible for the descendant element to exist within the subtree, a compile-time warning is produced (in the same way as previous releases do for the child and attribute axes).

Expressions using the axis step `child::*` now have their static type inferred if the schema only allows one possible element type in this context. This may lead to warnings being produced when a path expression using such a construct cannot select anything.

In previous releases of the Saxon-SA join optimizer, document-level indexes (keys) were not used to index expressions that required sorting into document order, for example `doc('abc.xml')//b/c[@d=$x]"`. This restriction has been removed.

Saxon-SA is now better at detecting when there is an indexable term in a predicate masked by other terms that are not indexable, for example `doc('abc.xml')/a/b/c[@d gt 5 and @e=$x]"` which will now be indexed on the value of `@e`

Saxon has always gone to some efforts to ensure that the result of a path expression is not sorted at run-time if the path is , that is, if the nested-loop evaluation of the path expression will deliver nodes in document order anyway. One situation where this is not possible is with a path of the form $v/a/b/c/d, in the case where Saxon cannot determine statically that $v will be a singleton. In this situation Saxon was effectively generating the expression sort($v/a/b/c/d). This has now changed in Saxon-SA so that in the case where the tail of the path expression (a/b/c/d) is naturally sorted, Saxon now generates a conditional sort expression, which performs the sort only if the condition `exists($v[2])` is true. (Note: it is not possible to rewrite the expression as `sort($v)/a/b/c/d`, because this can result in duplicates if $v is not a peer node-set, that is, if it contains one node that is an ancestor of another.)

This optimization (which is available only in Saxon-SA) benefits many queries of the form:

```
let $x := doc('abc.xml')//item[@code='12345']
return $x/price, $x/value, $x/size
```

Where the expression `EXP1` in `for $i in EXP1 return EXP2` is known to be a singleton, the expression is rewritten `let $i := EXP1 return EXP2`. This creates the opportunity for further simplifications.

Local variables are now inlined if they are bound to a constant value. Previously variables were inlined only in cases where there is just one reference to the variable. This creates the opportunity for further static evaluation of constant subexpressions.

In Saxon-SA, function calls are now inlined, provided certain conditions are met. These conditions are currently rather conservative. The function that is inlined must not call any user-defined functions, and it must not exceed a certain size (currently set, rather arbitrarily, to 15 nodes in the expression tree). It must also not contain certain constructs: for example, various XSLT instructions such as xsl:number and xsl:apply-templates, any instruction that performs sorting, or a "for" expression with an "at" variable.

When a function call is inlined, the original function remains available even if there are no further calls to it. This is because there are interfaces in Saxon that allow functions in a query module to be located and invoked dynamically.

If a subexpression within a function or template body does not depend on the parameters to the function, does not create new nodes, and is not a constant, then it is now extracted from the function body and evaluated as a global variable. This might apply to an expression that depends on other global variables or parameters, or to an expression such as doc('abc.xml') that is never evaluated at compile time. This optimization applies only to Saxon-SA. However, during testing of this optimization a considerable number of cases were found where Saxon was not taking the opportunity to do "constant folding" (compile-time evaluation of expressions) and these have been fixed, benefitting both Saxon-B and Saxon-SA.

Static type checking when applied to a conditional expression is now distributed to the branches of the conditional. ("Static type checking" here means checking that the static type of an expression is compatible with the required type, and generating run-time type checking code where this proves necessary). This means that no run-time checking code is now generated or executed for those branches of the conditional that are statically type-safe. This in turn means that if one branch of the conditional is a recursive tail call, tail call optimization is no longer inhibited by the unnecessary run-time type check on the value returned by the recursive call. Another effect of the change is that a static type error may now be reported if any branch of the conditional has a static type that is incompatible with the required type; previously this error would have been reported only when this branch was actually executed. This change affects XPath if/then/else, XSLT's `xsl:if` and `xsl:choose`, and XQuery typeswitch.

Tail-call optimization on `xsl:call-template` has also been improved. In the past this optimization was never applied if the named template declared a return type. This restriction is removed. To enable this, the static type inferencing on xsl:call-template has been improved. (Note however that declaring a return type on a match template will still generally inhibit tail call optimization, because calls on `xsl:apply-templates` cannot be statically analyzed.)

Saxon-SA now optimizes certain multi-branch conditional expressions into an efficient switch expression. The expressions that qualify are XSLT `xsl:choose` instructions or multi-way XPath/XQuery `if () then ... else if () then ...` expressions where all the conditions take the form of singleton comparisons of the same expression against different literal constants, for example `@a = 3`, `@a = 7`, `@a = 8`. The expression on the left must be identical in each case, and the constants on the right must all be of the same type. The expression is optimized by putting the constant values (or collation keys derived from them) in a hash table and doing a direct lookup on the value of the expression.

There has been some tuning applied to the DOM interface, specifically the wrapper code which implements the `NodeInfo` interface on top of `org.w3.dom.Node`. The frequently-used iterator for the child axis was creating nodes for all the children in a list, to ensure that adjacent text nodes were properly concatenated. This has changed so that the creation of nodes is now incremental.

Saxon now tries more aggressively to precompile regular expressions that are known at compile time, where these are used in the XPath functions `matches()`, `tokenize()`, and `replace()`, or in the `xsl:analyze-string` instruction. Previously, this was only done (in general) when the regex was written as a string literal or a constant subexpression. It is now done also when the regex can be reduced to a string literal during earlier stages of optimization. In particular, it now handles the case where the expression is written in the content of an XSLT variable, as this is a popular coding idiom because it avoids problems with escaping curly braces and other special characters.

There are some improvements in the optimization of expressions used in a context where the effective boolean value is required. These now all use the same logic (implemented as a static method in class BooleanFn). Expressions known to return nodes are now wrapped in a call of exists(), which takes advantage of the ability of some iterators to report whether any nodes exist without materializing the node. Expressions of the form `A | B` appearing in a boolean context are rewritten as `exists(A) or exists(B)`, which eliminates the costs of sorting into document order and checking for duplicates. Calls to normalize-space() in a boolean context are optimized to simply test whether the string contains any non-whitespace characters.

There has been a complete redesign of the optimization of expressions such as `SEQ[position() gt 5]` - specifically, filter expressions that perform an explicit test on the position() function. These are generally rewritten into a call of `subsequence()`, `remove()`, or the new internal function `saxon:item-at()`, or (typically for `S[position() != 1]`) into a `TailExpression`. Where necessary, conditional logic is added to the call to handle the case where the expression being compared to `position()` is not guaranteed to be an integer, or might be an empty sequence. This redesign eliminates the need for the expression types `PositionRange` and `SliceExpression`.

Compile-time performance has been improved for expressions containing long lists of subexpressions separated by the comma operator. Lists longer than 5000 or so items were blowing the Java stack, and the compile time was also quadratic in the number of subexpressions.

Document Projection

Document Projection is a mechanism that analyzes a query to determine what parts of a document it can potentially access, and then while building a tree to represent the document, leaves out those parts of the tree that cannot make any difference to the result of the query.

In this release document projection is an option on the XQuery command line interface. Currently it is only used if requested.

Internally, the class `PathMap` computes (and represents) the set of paths within a document that are followed by an expression, that is, for each document accessed by an expression, the set of nodes that are reachable by the expression. A PathMap can be set on an `AugmentedSource` supplied to the `Configuration.buildDocument()` method to request filtering of the document while constructing the tree. If `-explain` is specified, the output includes feedback on the use of document projection.

Diagnostics

The format of the "explain" output which displays a compiled and optimized expression tree has changed. The format is now XML, making the raw output amenable to further processing, for example filtering or graphical display. The `-e` flag on the Query command line is extended to allow `-explain:filename`, so that the output can more easily be captured in a file.

The options `-explain` and `-explain:filename` have been added to the `Transform` command line allowing an expression tree to be generated for an entire stylesheet. The extension attribute `saxon:explain` remains available for more selective reporting.

> Because the explain output is now XML, it is amenable to analysis using XQuery or XSLT. The Saxon XQuery test driver now allows the test catalog to contain assertions (in the form of XPath expressions) about the content of the expression tree. This allows tests to be written that check not only that the query produces correct results, but that the expected optimizations are applied.

From the Java level, explain output is available via a new method on the `Expression` class. The old `display()` method is retained (for the time being) for compatibility, but produces output in the new format.

NamePool changes

The `NamePool` is no longer used for names such as variable names which are not used at run-time. This change is made to ease pressure on the NamePool as a shared resource which can become a bottleneck for some high-throughput applications, and which can gradually fill for long-running applications. The problem can arise particularly because the Saxon optimizer generates variable names at random for internal variables, meaning that there is a slow but steady increase in the number of entries in the NamePool even under a very stable workload. The name of a variable is now held internally in a `StructuredQName` object, which holds the prefix, URI, and local name in a structure that is designed for economy in space usage combined with an efficient equals() test.

The same change has been made for other kinds of name such as function names, template names, attribute set names, character map names, mode names, output format names, decimal format names, and key names. In the vast majority of cases these names are resolved at compile time so there was little benefit from using the shared name pool.

Local parameters to XSLT templates, which are matched by name at run-time, still use numeric identifiers for efficient matching, but these are no longer fingerprints allocated from the namepool, they are numbers allocated by the stylesheet compiler locally to a stylesheet.

User applications are unlikely to be affected by the change unless they probe rather deeply into Saxon system-programming interfaces, for example interfaces provided for debuggers, or for defining your own extension function binding factories. But if you provide your own implementation of the `StaticContext` interface, you will need to change the method `bindVariable()` to accept a `StructuredQName` rather than an integer fingerprint.

Expression tree changes

There have been changes to the internal structure of the expression tree generated by the XSLT, XQuery, and XPath processors, and to the way it is navigated. Most notably, the tree no longer contains any parent pointers linking a subexpression to its containing expression. These have been removed primarily because the code for maintaining the parent pointers was complex and prone to bugs. To compensate for the absence of these pointers, the various traversals of the expression tree (simplify, typeCheck, and optimize), now make use of an `ExpressionVisitor` object that maintains references to all the containing expressions in the form of a stack.

Expressions now have a link to a `Container` object that provides access to the outside world, for example to the `Configuration` and `NamePool`. However, this is used only for diagnostics, because it is not guaranteed to be available in 100% of cases, especially while the tree is under construction.

There is now an internal diagnostic switch allowing tracing of the decisions made by the optimizer. Not all rewrites are yet traced in this way.

The class `IfExpression` no longer exists; all conditional expressions including `xsl:if`, `xsl:choose`, XPath `if-then-else`, and XQuery `typeswitch` are now compiled to a (potentially multi-way) `Choose` expression.

Chapter 3. Licensing

Introduction

This section of the documentation provides information about Saxon licensing, including required notices for open-source components that have been incorporated into Saxon, and acknowledgments of contributors.

This documentation relates both to the open source Saxon-HE product and to the commercial products Saxon-PE and Saxon-EE. The conditions of use are of course different in the two cases.

The information in this section applies to both the Java and .NET versions of Saxon, unless otherwise specified.

Saxon-HE

The open-source Saxon-HE product is made available under the Mozilla Public License Version 1.0 (the "License"); you may not use the software except as permitted by the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/

The source code of Saxon-HE can be considered for licensing purpose as having three parts:

- Category A consists of code which was written as part of Saxon either by the initial developer, or by another contributor. All such components are subject only to the MPL 1.0 license. A List of Contributors is provided, for information.
- Category B code was originally produced as part of some other product and subsequently incorporated (with varying degrees of modification) into Saxon by way of source code integration. Many of these components have their own license conditions: these are in all cases licenses similar in form to either the Mozilla Public License, the Apache license, or the BSD license. All these licenses are "non-viral": they permit the code to be combined into a commercial product without requiring the commercial code to become open source. In some cases the license conditions require the origin of the code to be acknowledged, typically by including a notice in all distributions of the product. These notices are provided in the `notices` directory of the Saxon product as distributed, and the documentation provides a table listing all these Third-party code components.
- Category C code consists of components that are included unchanged in the Saxon distribution in binary form, for the convenience of users to avoid the need for a separate download. (In the Java product this includes the ASM library for bytecode generation; in the .NET product it also includes the IKVMC and OpenJDK runtimes, the Apache Xerces parser, and the TagSoup HTML parser.) These are listed as Redistributed components

Also distributed with Saxon-HE is a JAR file, `saxon9-unpack.jar`, which contains proprietary Saxonica code (non-open-source) for executing a "packaged" stylesheet that has been prepared under Saxon-PE or Saxon-EE. For details of this feature see Packaged Stylesheets. This JAR file can be freely redistributed, but in all other respects, the terms and conditions published at http://www.saxonica.com/paid-license.pdf apply. If you do not want to accept these conditions, please delete this JAR file.

Saxon-EE and Saxon-PE

The Enterprise and Professional editions of Saxon are commercial products released under the terms and conditions published at http://www.saxonica.com/paid-license.pdf.

These products include the functionality of Saxon-HE as a subset. The source code for Saxon-PE and Saxon-EE can be considered to be in three parts:

1. Source code for which Saxonica Limited owns the copyright, which Saxonica has chosen to make available to the public under the Mozilla Public License (around 250K lines of code)

2. Source code for which Saxonica Limited owns the copyright, which Saxonica has chosen to retain as proprietary (around 90K lines of code)

3. Open source code developed and licensed by third parties and used by Saxonica under the terms of those licenses (around 20K lines of code)

The code in the third category is in most cases also used in Saxon-HE (the only open source code in Saxon-PE or Saxon-EE that is not also used in Saxon-HE is the ASM bytecode generation library). In all cases the license under which the code was used permits the creation of commercial products derived from this code, and does not "infect" such products with open source obligations. In many cases the relevant license requires a notice to be published; this is satisfied by inclusion of the relevant notices in this docoumentation and also in the `notices` directory of the issued product. In many cases the relevant license also requires any modifications to source code to be published; this is satisfied by issuing the source code of Saxon-HE, which includes all such modifications.

The Saxon SQL extension

The Saxon SQL extension is available as an open-source plug-in to Saxon-PE or Saxon-EE. It will not run with Saxon-HE because it relies on XSLT element extensibility, a feature not available in Saxon-HE. However, the code of the SQL extension itself is open-source and is issued under the Mozilla Public License, meaning that you are free to extend it and customize it to your needs, even though it requires Saxon-PE or Saxon-EE to run.

EXSLT extensions

A substantial number of extension functions defined in EXSLT are available as an open-source plug-in to Saxon-PE or Saxon-EE. These are implemented as reflexive extension functions, and therefore rely on a mechanism not available in Saxon-HE. However, the code of the these extension functions itself is open-source and is issued under the Mozilla Public License, meaning that you are free to extend it and customize it to your needs, even though it requires Saxon-PE or Saxon-EE to run.

Redistribution

Redistribution of Saxon-HE is freely permitted under the terms of the Mozilla Public License. Note that this requires inclusion of all the necessary notices. If any source code changes are made, the license requires that they be published; but you are not required to publish the source code of your own application code.

If you produce a product that includes or requires Saxon-HE, please refer to it prominently as "The Saxon XSLT and XQuery Processor from Saxonica Limited", and include the URL of the home page, which is at http://www.saxonica.com/. As a courtesy, please take reasonable steps to ensure that your users know that they are running Saxon.

Redistribution of Saxon-PE or Saxon-EE as a component of a commercial application is possible under commercial terms; prices are published in the Saxonica online store.

Technical Support (Saxon-HE)

Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

There is no guarantee of technical support, though we are usually able to answer enquiries within a few days. Please subscribe to the mailing list available at http://lists.sourceforge.net/lists/listinfo/saxon-help [http://lists.sourceforge.net/lists/listinfo/saxon-help] and raise any enquiries there; or use the saxon-help forum, also available on the SourceForge pages. Also check the Saxon project pages on SourceForge for details of known errors; all bugs are listed there as soon as there is sufficient evidence to describe the nature of the problem.

Contributors

This page lists contributors to the "Category A" source code of Saxon-HE, as defined above. This list is provided purely for information and does not imply that the contributor has any rights, responsibilities, or liabilities in respect of the code. Definitive information about contributors to each module is included in the standard wording of the Mozilla Public License present in each module of the source code.

The aim is to acknowledge all contributions, however small. The information has been compiled after the event, so there may be contributions that are not mentioned here. I apologize for any omissions and will be happy to rectify them. I will also remove any names from this list on request, though the names cannot be omitted from the source code itself.

All contributors listed in this section explicitly asked or agreed to have their code published as part of the Saxon open source product and thus explicitly or implicitly agreed to its release under the Mozilla public license.

> If you are interested in becoming a contributor, please contact Saxonica before sending any code. You will need to sign a written contributor agreement, perhaps countersigned by your employer; and you will need to discuss technical arrangements such as the format for test material.

The LOC figure is an estimate of the number of lines of code contributed, including comments.

Table 3.1.

Rick Bonnett		250	Enhancements to the Saxon code for accessing relational databases (package net.sf.saxon.sql, modules SQLQuery and SQLClose)	2004?
Erik Bruchez	Orbeon	1800	Code to interface Saxon with DOM4J. Package net.sf.saxon.dom4j.	2006
Dominique Devienne and Dave Hale	Landmark Graphics	1000	Utilities for handling integer sets and maps. Package net.sf.saxon.sort, modules IntHashMap, IntHashSet, IntToIntHashMap	2005?
Ruud Diterwich		300	Code for efficient copying of trees. package net.sf.saxon.event module DocumentSender; package net.sf.saxon.tinytree	2004?

			module TimyElementImpl method copy()	
Efraim Feinstein		100	Number formatting in traditional Hebrew.	2009
Edwin Glaser		1000	Diagnostic code for tracing execution of stylesheets (package net.sf.saxon.trace, various modules; and calls to these routines scattered around the Saxon code)	2001?
Karel Goossens	BTR-Services, Belgium	1000	Number and date formatting for Danish, Swedish, Italian, Dutch, Belgian French, and Flemish. Package net.sf.saxon.number, module Numberer_XX where XX is da, sv, it, nl, frBE, nlBE	2007
Wolfgang Hoschek	Lawrence Berkeley [US] National Laboratory	1800	Code to interface Saxon with XOM. Package net.sf.saxon.xom, all modules	2005?
Dmitry Kirsanov		12	Data used for Cyrillic numbering. Package net.sf.saxon.number, module Numberer_en	2002?
Mathias Payer		140	Enhancements to the Saxon code for accessing relational databases Package net.sf.saxon.sql, module SQLDelete	2002?
Murakami Shinyu		30	Data used for Japanese numbering. Package net.sf.saxon.number,	2002?

			module Numberer_en	
Luc Rochefort (with testing by Laurent Bourbeau and Grégoire Djénandji		250	Number and date formatting in French. Package net.sf.saxon.number, module Numberer_fr	2005?
Gunther Schadow		20	Enhancements to Query command line interface to allow input from stdin. Package net.sf.saxon, module Query	2004?
Simon StLaurent	;	320	EXSLT math library. Package net.sf.saxon.option.exslt, module Math	2004?
Martin Szugat		140	EXSLT random library. Package net.sf.saxon.option.exslt, module Random	June 2004
Claudio Thomas		290	Enhancements to the Saxon code for accessing relational databases. Package net.sf.saxon.sql, module SQLQuery	2003?

Third Party Source Components

These tables lists components in category B as described above. (Category B is open source code that has been integrated at source level, without the involvement of the original author.)

Unlike contributed code, this code was not written specifically for inclusion in Saxon, but was originally published under some other license.

A2 Base64 Encoder/Decoder

Table 3.2.

Origin	Netscape Communications Corp
Description	Encoder and decoder for Base 64
Approximate LOC	400
Saxon packages / modules	Two inner classes within net.sf.saxon.value.Base64Value
Modifications	Minor modifications needed to meet the W3C XML Schema specification for Base64 lexical representation
Availability of source	Apparently no longer available. Originally part of Netscape Directory Server, package

	netscape.ldap.util, modules MimeBase64Encoder and MimeBase64Decoder.
Source version used	Unknown. Snapshot taken in 2004.
License	Netscape 1.1: see http://www.mozilla.org/MPL/NPL-1.1.html

A3 Generic Sorter

Table 3.3.

Origin	CERN (author Wolfgang Hoschek)
Description	Generic sort routines based on published algorithms
Approximate LOC	500
Saxon packages / modules	net.sf.saxon.sort.GenericSorter
Modifications	Minimal modifications needed to integrate the code
Availability of source	Currently available as part of Colt project, http://dsd.lbl.gov/~hoschek/colt/, module cern.colt.GenericSorting
Source version used	Unknown. Snapshot taken in 2004?
License	CERN License: see below

Copyright © 1999 CERN - European Organization for Nuclear Research. Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. CERN makes no representations about the suitability of this software for any purpose. It is provided "as is" without expressed or implied warranty.

A4 Unicode Normalization

Table 3.4.

Origin	Unicode Consortium (author Mark Davis)
Description	Routines for Unicode character normalization
Approximate LOC	3500 (including data sets)
Saxon packages / modules	net.sf.saxon.sort.codenorm.*
Modifications	Core functionality unchanged; rewrote the module that loads the data tables from the Unicode character database; removed dependencies on ICU; fixed a few bugs
Availability of source	Specification of algorithm at http://unicode.org/reports/tr15/, code available via http://www.unicode.org/reports/tr15/Normalizer.html
Source version used	No version number. Snapshot taken in June 2005
License	Unicode license: see below:

A5 XPath Parser

Table 3.5.

Origin	James Clark (www.jclark.com)
Description	Top-down parser and lexical tokenizer for XPath
Approximate LOC	1000 (including data sets)
Saxon packages / modules	net.sf.saxon.expr.*, modules ExpressionParser, Tokenizer, Token
Modifications	Almost entirely rewritten with enhancements to handle XPath 2.0/3.0 and XQuery 1.0/3.0 syntax, improved error reporting, etc.
Availability of source	Derives from James Clark's xt product, which in its original form is at http://www.jclark.com/xml/xt-old.html. Package com.jclark.xsl.expr, modules ExprParser and ExprTokenizer
Source version used	Unknown. Snapshot taken in 1999.
License	James Clark (see below). Apparently copyright has since been transferred to the Thai Open Source Center Ltd.

A6 Regex Translator

Table 3.6.

Origin	James Clark (www.jclark.com)
Description	Translator from XML Schema regular expressions to JDK 1.4 regular expressions
Approximate LOC	1000
Saxon packages / modules	net.sf.saxon.java, modules JDK14RegexTranslator and JDK15RegexTranslator, and net.sf.saxon.dotnet, module DotNetRegexTranslator
Modifications	Significantly enhanced to handle XPath 2.0/3.0 regular expressions as input and to produce JDK 1.5 and .NET regular expressions as output; and to support Unicode 6.0.0.
Availability of source	Available at http://www.thaiopensource.com/download/xsdregex-20020430.zip
Source version used	20020430
License	Thai Open Source Center Ltd.

Redistributed Components

This page describes Category C components as defined above: components that are redistributed with Saxon in binary form, without alteration.

Saxon on Java

See http://asm.ow2.org/

Saxon-EE, on both Java and .NET, includes the ASM bytecode generation library. The code is issued without modification, except that on .NET it is cross-compiled to .NET IL form.

The license can be found at http://asm.ow2.org/license.html

Applicable notice: ASM.txt

Saxon on .NET

See http://www.ikvm.net/. Can be downloaded from http://sourceforge.net/project/showfiles.php?group_id=69637. Saxon links dynamically to this DLL.

License is at http://weblog.ikvm.net/story.aspx/license

Applicable notice: FRIJTERS.txt

The copy of OpenJDK Classpath released with Saxon on .NET is a derived from the version that is released as part of IKVM Runtime (see above). This in turn is derived largely from the Sun OpenJDK distribution, together with some components taken from the Red Hat IcedTea library, compiled into CIL code using IKVMC. Saxon 9.1 uses IKVM 0.40.0.1, which in turn uses OpenJDK 7 b13.

Saxon distributes only the parts of this library that are actually needed. These parts have been rebuilt from source code, but no source modifications have been made.

Saxon links dynamically to this DLL.

The license conditions are at http://openjdk.java.net/legal/gplv2+ce.html. (This is a modified variant of the GNU Public License version 2, with a special clause allowing it to be used as part of a product that is not itself open source.)

Applicable notice: GPL+CLASSPATH.txt

Rather than distributing the Sun version of Xerces that comes with the OpenJDK library, Saxon instead distributes the Apache version of Xerces. (The two versions have diverged considerably. The Apache version is used because it is considered more reliable and is also easier to integrate because it does not have unnecessary dependencies on other parts of the JDK library.)

The two Apache JAR files `xercesImpl.jar` and `resolver.jar` have been cross-compiled to IL code using the IKVMC compiler, but are otherwise unmodified.

The license conditions for Xerces are at http://www.apache.org/licenses/LICENSE-2.0

Applicable notices: APACHE-XERCES.txt, APACHE-RESOLVER.txt

Supporting the `saxon:parse-html()` extension function, the code of TagSoup version 1.2 is included in the Saxon-PE and Saxon-EE distributions on .NET. Apart from cross-compiling from Java bytecode to IL code, the code is unmodified.

Information about TagSoup, including links to the download location for source code, is available at http://home.ccil.org/~cowan/XML/tagsoup/.

The license conditions for TagSoup are at http://www.apache.org/licenses/LICENSE-2.0

Applicable notices: APACHE-TAGSOUP.txt

Published Algorithms and Specifications

The table below lists published specifications and algorithms that formed a significant intellectual input into the development of the product.

Table 3.7.

Java 2 SE 6 interface specifications	Saxon implements many interfaces defined in the Java 2 SE specifications, notably the JAXP interfaces
W3C specifications for XML 1.0, XSLT 2.0, XPath 2.0, XQuery 1.0 and associated documents	Define the language standards that Saxon implements
Using Finite-State Automata to Implement W3C XML Schema Content Model Validation and Restriction Checking. Henry Thompson and Richard Tobin. XML Europe 2003	The Saxon-EE schema processor implements this algorithm to validate instance documents
How to Print Floating Point Numbers Accurately. Guy Steele and Jon White. ACM SIGPLAN 1990	Saxon uses this algorithm to convert floating point numbers to strings
Amélie Marian and Jérôme Siméon. Projecting XML Documents. VLDB'2003, Berlin, Germany, September 2003.	Saxon-EE provides the option of performing document projection (to eliminate parts a a source document that a query cannot reach) using an algorithm similar to the one published in this paper
XQJ (XQuery API for Java). Java Community Process JSR-225. The current Public Draft 0.9	The specification includes a license describing the terms under which an implementation of the specification may be made or distributed.

Specification is Copyright (c) 2003, 2006, 2007 Oracle.	Saxon contains a provisional implementation of this specification, provided for users who wish to gain early exposure to this draft API. This is distributed under the terms of the license in the JSR document.
Date calculation algorithms, taken from http://vsg.cape.com/~pbaum/date/jdalg.htm, http://vsg.cape.com/~pbaum/date/jdalg2.htm, and http://www.hermetic.ch/cal_stud/jdn.htm#comp	Saxon implements these published algorithms for converting dates to Julian day numbers and vice versa

Chapter 4. Saxon Configuration

Introduction

There are many parameters and options that can be set to control the way in which Saxon behaves, and there are many different ways these parameters and options can be set. This section of the documentation brings this information together in one place.

- Configuration interfaces
- The Saxon configuration file
- Configuration Features

Configuration interfaces

At the heart of Saxon is the object `net.sf.saxon.Configuration [Javadoc: net.sf.saxon.Configuration]`. This contains all the current settings of configuration options. All Saxon tasks, such as compiling and running queries and transformations, or building and validating source documents, happen under the control of a `Configuration`. Many resources are owned by the `Configuration`, meaning that related tasks must run under the same `Configuration`. Most notably, the `Configuration` holds a `NamePool [Javadoc: net.sf.saxon.om.NamePool]`, which is a table allocating integer codes to the qualified names that appear in stylesheets, queries, schemas, and source documnets, and during the execution of a stylesheet or query all the resources used (for example, the stylesheet, all its input documents, and any schemas it uses) must all use the same `NamePool` to ensure that they all use the same integer codes for the same qualified names. However, two Saxon tasks that are unrelated can run under different `Configuration`s.

There are subclasses of `Configuration` containing resources associated with the capabilities of the different Saxon editions: specifically, `com.saxonica.config.ProfessionalConfiguration [Javadoc: com.saxonica.config.ProfessionalConfiguration]` and `com.saxonica.config.EnterpriseConfiguration [Javadoc: com.saxonica.config.EnterpriseConfiguration]`. In many cases the `Configuration` is used as a factory class to deliver services associated with the different capability levels, for example the method `getOptimizer()` returns the query optimizer appropriate to the loaded Saxon edition.

Many configuration options have direct setter and getter methods on the `Configuration` object, for example `setAllowExternalFunctions()` and `isAllowExternalFunctions()`. Some other options have setters and getters on objects reachable from the `Configuration`, for example defaults for XSLT processing can be controlled using methods such as `getDefaultXsltCompilerInfo().setXsltVersion()`, while defaults for XQuery processing can be controlled using methods such as `getDefaultStaticQueryContext().setLanguageVersion()`.

The most general mechanism for getting and setting configuration properties, however, is provided by the methods `getConfigurationProperty(name)` and `setConfigurationProperty(name, value)`. In these methods the name of the configuration property is always a string in the form of a URI (for example, `"http://saxon.sf.net/feature/initialTemplate"`), and the strings available are all defined by constants in the class `net.sf.saxon.lib.FeatureKeys [Javadoc: net.sf.saxon.lib.FeatureKeys]` (for example, `FeatureKeys.INITIAL_TEMPLATE`). The value is of various types depending on the property.

In the vast majority of cases, the property can be supplied as a string, or it has an alternative, equivalent property that can be supplied as a string. For properties that are essentially boolean in nature the value can be supplied either as one of the Java constants `Boolean.TRUE` or `Boolean.FALSE`, or as one of the strings "true", "1", "yes", "on", or "false", "0", "no", "off". These choices are designed to suit the conventions of different APIs in which the configuration options are exposed.

In many APIs for controlling Saxon activities, the `Configuration` object is not exposed directly, but is hidden below some similar object acting as the root object for that particular API. Many of these objects provide a direct way to set the configuration options. These are discussed in the following sections.

- JAXP Factory Interfaces
- Configuration using s9api
- Configuration using the .NET API
- Configuration from the command line
- Configuration using XQJ
- Configuration when running Ant

JAXP Factory Interfaces

Saxon implements a number of JAXP interfaces, notably the APIs for transformation, XPath processing, and validation.

For transformation, the root object of the API is the JAXP `TransformerFactory`. Saxon provides three implementations of this interface: `net.sf.saxon.TransformerFactoryImpl [Javadoc: net.sf.saxon.TransformerFactoryImpl]` for Saxon-HE, and `com.saxonica.config.ProfessionalTransformerFactory [Javadoc: com.saxonica.config.ProfessionalTransformerFactory]` and `com.saxonica.config.EnterpriseTransformerFactory [Javadoc: com.saxonica.config.EnterpriseTransformerFactory]` for Saxon-PE and Saxon-EE respectively. This interface provides methods `getAttribute(name)` and `setAttribute(name, value)` which correspond directly to the methods `getConfigurationProperty(name)` and `setConfigurationProperty(name, value)` on the underlying `Configuration [Javadoc: net.sf.saxon.Configuration]` object. By casting from the JAXP interface to the Saxon implementation class it is also possible to call the `getConfiguration` method which exposes the `Configuration` object directly.

The Saxon-PE and Saxon-EE implementations of the `TransformerFactory` also allow the configuration property `FeatureKeys.CONFIGURATION_FILE [Javadoc: net.sf.saxon.lib.FeatureKeys#CONFIGURATION_FILE]` to be set. The value is a filename containing the name of a configuration file, which must have the format described in Configuration file. This causes any previously-initialized configuration to be discarded, and replaced with a new `Configuration` object built from the settings in the specified configuration file.

The JAXP `XPathFactory` interface has a general-purpose configuration mechanism in the form of the two methods `setFeature()` and `getFeature()`. These can be used to set/get all boolean-valued configuration options in the underlying Saxon `Configuration`, as well as the options defined in the JAXP interface itself. To set configuration options that are not boolean-valued, it is necessary to navigate to the underlying `Configuration` object and use its native interfaces. Saxon's implementation class for the `XPathFactory` is `net.sf.saxon.xpath.XPathFactoryImpl [Javadoc: net.sf.saxon.xpath.XPathFactoryImpl]`, regardless which Saxon edition is in use.

> Note that although Saxon implements the JAXP XPath API, it is rarely possible to use it in a way that has no dependencies on the Saxon implementation, partly because of the requirement for all Saxon tasks to run under the same `Configuration`. When using the XPath API, therefore, the `Configuration` object is usually exposed to the application.

Saxon-EE also implements the JAXP `SchemaFactory` in class `com.saxonica.jaxp.SchemaFactoryImpl [Javadoc: com.saxonica.jaxp.SchemaFactoryImpl]`. The interface offers methods `getProperty(name)` and `setProperty(name, value)` which map to the underlying methods in the Saxon `Configuration`; again, it is also possible to cast to the Saxon implementation class and call configuration-setting methods directly.

Configuration using s9api

In Saxon's s9api interface the root API object is the `net.sf.saxon.s9api.Processor [Javadoc: net.sf.saxon.s9api.Processor]` object. This again is a wrapper around a `Configuration [Javadoc: net.sf.saxon.Configuration]`. All the configuration properties are exposed via the `Processor` methods `getConfigurationProperty(name)` and `setConfigurationProperty(name, value)` which map directly to the same methods on the underlying `Configuration`.

The s9api `Processor` object also has a constructor `new Processor(source)` which allows the underlying `Configuration` to be built from a supplied configuration file. The argument is a `org.xml.sax.Source` object, for example a `StreamSource`, which identifies the configuration file, which must have the format described in Configuration file.

In many cases with s9api it is more appropriate to set options at a finer level of granularity than the `Processor`. For example, options that affect XSLT stylesheet compilation can be set on the `XsltCompiler [Javadoc: net.sf.saxon.s9api.XsltCompiler]` object, and options that affect XQuery compilation on the `XQueryCompiler [Javadoc: net.sf.saxon.s9api.XQueryCompiler]`. Some more specialized configuration options are not exposed directly by these two classes, but can be tailored by accessing the underlying support objects: `CompilerInfo [Javadoc: net.sf.saxon.trans.CompilerInfo]` in the case of XSLT, and `StaticQueryContext [Javadoc: net.sf.saxon.query.StaticQueryContext]` in the case of XQuery.

Configuration using the .NET API

In Saxon's Saxon.API interface on .NET the root API object is the `Saxon.Api.Processor` object. This again is a wrapper around a `Configuration`. All the configuration properties are exposed via the `Processor` methods `getProperty(name)` and `setProperty(name, value)` which map directly to the methods`getConfigurationProperty(name)` and `setConfigurationProperty(name, value)` on the underlying `Configuration`.

The s9api `Processor` object also has a constructor `new Processor(stream)` which allows the underlying `Configuration` to be built from a supplied configuration file, which must have the format described in Configuration file. Configuration files are available only in Saxon-PE and Saxon-EE.

In many cases with the Saxon.Api interface it is more appropriate to set options at a finer level of granularity than the `Processor`. For example, options that affect XSLT stylesheet compilation can be set on the `XsltCompiler` object, and options that affect XQuery compilation on the `XQueryCompiler`.

Configuration from the command line

The main command-line interfaces to Saxon are `net.sf.saxon.Transform [Javadoc: net.sf.saxon.Transform]` for running a transformation, `net.sf.saxon.Query [Javadoc: net.sf.saxon.Query]` for running XQuery, and `com.saxonica.Validate [Javadoc: com.saxonica.Validate]` for validating a document against a schema. These commands allow many configuration options to be specified by command line options: for example if XML Schema 1.1 support is wanted, all three commands allow this to be requested using the option `-xsdversion:1.1`. Many of these options correspond directly to the configuration properties available on the `Configuration` object.

For more specialized options, there is also a fallback mechanism. Each configuration property has a URI, which is always of the form `http://saxon.sf.net/feature/SOME_NAME`. Provided the property allows a string value (as most do), the property can be set from the command line using the syntax `--SOME_NAME:value`. For example the property identified by `FeatureKeys.LICENSE_FILE_LOCATION [Javadoc: net.sf.saxon.lib.FeatureKeys#LICENSE_FILE_LOCATION]` has the URI `http://saxon.sf.net/feature/licenseFileLocation`, and it can therefore be set on the command line using an option such as `--licenseFileLocation:c:/saxon/license.lic`.

For boolean-valued configuration properties, Saxon accepts any of the values "yes", "on", "true", or "1" to switch the option on, or "no", "off", "false", or "0" to switch it off.

Configuration using XQJ

The root object in the XQJ (XQuery for Java) API is `javax.xml.query.XQDataSource`, and the Saxon implementation class is `net.sf.saxon.xqj.SaxonXQDataSource [Javadoc: net.sf.saxon.xqj.SaxonXQDataSource]`. The `XQDataSource` provides methods `getProperty(name)` and `setProperty(name, value)` which at first sight appear to map cleanly to the methods `getConfigurationProperty(name)` and `setConfigurationProperty(name, value)` in the underlying Saxon `Configuration [Javadoc: net.sf.saxon.Configuration]`, and indeed they can be used in this way, using either the URI value of the property or the symbolic constant in class `net.sf.saxon.lib.FeatureKeys [Javadoc: net.sf.saxon.lib.FeatureKeys]`.

There are some glitches, however. Firstly, the XQJ specifications mandate that the properties available through this interface should also have explicit getters and setters: for example if a property named "lineNumbering" is available, then there should be a pair of methods `setLineNumbering()` and `getLineNumbering()`. This does not square well with the use of URIs for property names. Secondly, XQJ requires that the property values should be strings. Saxon therefore:

1. exposes a subset of commonly-used configuration properties using shortened names such as `dtdValidation` and `retainLineNumbers`;
2. provides getters and setters for these properties, as required by the XQJ specification;
3. lists the names of the above properties (only) in the result of the method `getSupportedPropertyNames()`
4. makes all other configuration properties available using URIs as the name, without providing getters and setters, and without listing the names in the result of `getSupportedPropertyNames`, provided that the value can be represented as a string. Boolean values can be represented using any of the strings ("yes", "on", "true", or "1"), or ("no", "off", "false", or "0").

Configuration when running Ant

It is possible to run XSLT transformations from Ant using the `xslt` task, selecting Saxon as the XSLT processor by setting the `factory` child element to the Saxon implementation class of `javax.xml.transform.TransformerFactory`, that is one of `net.sf.saxon.TransformerFactory` [Javadoc: `net.sf.saxon.TransformerFactory`], `com.saxonica.config.ProfessionalTransformerFactory` [Javadoc: `com.saxonica.config.ProfessionalTransformerFactory`], or `com.saxonica.config.EnterpriseTransformerFactory` [Javadoc: `com.saxonica.config.EnterpriseTransformerFactory`] depending on the Saxon edition in use.

Additional configuration options can be specified using the `attribute` child of the `factory` element: for example the following task was used as part of the pipeline for publishing this documentation:

```
<xslt in="${userdoc.dir}/src/function-data.xml"
      style="${userdoc.dir}/style/preprocess-functions.xsl"
      out="${userdoc.dir}/src/functions.xml"
      classpath="e:/saxon/eej/saxon9ee.jar;e:/saxon/eej/saxon-licenses">
      <factory name="com.saxonica.config.EnterpriseTransformerFactory">
          <attribute name="http://saxon.sf.net/feature/xsltSchemaAware" v
          <attribute name="http://saxon.sf.net/feature/schema-validation-
          <attribute name="http://saxon.sf.net/feature/xsd-version" value
      </factory>
</xslt>
```

Many of the options available as configuration parameters (for example `FeatureKeys.XSLT_INITIAL_TEMPLATE)` [Javadoc: `net.sf.saxon.lib.FeatureKeys#XSLT_INITIAL_TEMPLATE`] were provided explicitly with Ant in mind. The provision of these parameters makes the customized version of the Ant XSLT task provided with some earlier Saxon versions redundant, and the customized task is no longer supported.

The Saxon configuration file

Configuration parameters for Saxon can be defined in a configuration file. This file is optional. It can be applied by using the option `-config:filename` on the `Transform`, `Query`, or `Validate` command line, or using the factory method `Configuration.readConfiguration()`.

> A schema for the configuration file is provided as `config.xsd` in the `samples` directory of the `saxon-resources` download.

In both the s9api interface on Java, and the Saxon.Api interface on .NET, there is a constructor on the `Processor` class that takes a configuration file as input.

Here is an example configuration file. It is designed to show as many options as possible; in practice, no option needs to be included if it is to take its default value, and it is probably good practice to only include those parameters that you need to specify explicitly. Some of the example values used in this sample will not work unless you have files with the relevant names at particular locations, or unless you have classes with particular names available on your classpath; if such entries cause problems, you can always delete them.

```
<configuration xmlns="http://saxon.sf.net/ns/configuration"
               edition="EE">
  <global
    allowExternalFunctions="true"
    allowMultiThreading="true"
    allowOldJavaUriFormat="false"
    collationUriResolver="net.sf.saxon.lib.StandardCollationURIResolver"
    collectionUriResolver="net.sf.saxon.lib.StandardCollectionURIResolver"
    compileWithTracing="false"
    defaultCollation="http://www.w3.org/2005/xpath-functions/collation/codepoin
    defaultCollection="file:///e:/temp"
    dtdValidation="false"
    dtdValidationRecoverable="true"
    errorListener="net.sf.saxon.StandardErrorListener"
    expandAttributeDefaults="true"
    lazyConstructionMode="false"
    lineNumbering="true"
    optimizationLevel="10"
    preEvaluateDocFunction="false"
    preferJaxpParser="true"
    recognizeUriQueryParameters="true"
    schemaValidation="strict"
    serializerFactory=""
    sourceParser=""
    sourceResolver=""
    stripWhitespace="all"
    styleParser=""
    timing="false"
    traceExternalFunctions="true"
    traceListener="net.sf.saxon.trace.XSLTTraceListener"
    traceOptimizerDecisions="false"
    treeModel="tinyTreeCondensed"
    uriResolver="net.sf.saxon.StandardURIResolver"
    usePiDisableOutputEscaping="false"
    useTypedValueCache="true"
    validationComments="false"
    validationWarnings="true"
    versionOfXml="1.0"
    xInclude="false"
  />

  <xslt
    initialMode=""
    initialTemplate=""
    messageReceiver=""
    outputUriResolver=""
    recoveryPolicy="recoverWithWarnings"
    schemaAware="false"
    staticErrorListener=""
    staticUriResolver=""
    styleParser=""
    version="2.1"
    versionWarning="false">
    <extensionElement namespace="http://saxon.sf.net/sql"
```

```
        factory="net.sf.saxon.option.sql.SQLElementFactory"/>
  </xslt>

  <xquery
    allowUpdate="true"
    constructionMode="preserve"
    defaultElementNamespace=""
    defaultFunctionNamespace="http://www.w3.org/2005/xpath-functions"
    emptyLeast="true"
    inheritNamespaces="true"
    moduleUriResolver="net.sf.saxon.query.StandardModuleURIResolver"
    preserveBoundarySpace="false"
    preserveNamespaces="true"
    requiredContextItemType="document-node()"
    schemaAware="false"
    staticErrorListener=""
    version="1.1"
    />

  <xsd
    occurrenceLimits="100,250"
    schemaUriResolver="com.saxonica.sdoc.StandardSchemaResolver"
    useXsiSchemaLocation="false"
    version="1.1"
  />

  <serialization
    method="xml"
    indent="yes"
    saxon:indent-spaces="8"
    xmlns:saxon="http://saxon.sf.net/"/>

  <localizations defaultLanguage="en" defaultCountry="US">
    <localization lang="da" class="net.sf.saxon.option.local.Numberer_da"/>
    <localization lang="de" class="net.sf.saxon.option.local.Numberer_de"/>
  </localizations>

  <resources>
    <externalObjectModel>net.sf.saxon.option.xom.XOMObjectModel</externalObject
    <extensionFunction>s9apitest.TestIntegrationFunctions$SqrtFunction</extensi
    <schemaDocument>file:///c:/MyJava/samples/data/books.xsd</schemaDocument>
    <schemaComponentModel/>
  </resources>

  <collations>
    <collation uri="http://www.w3.org/2005/xpath-functions/collation/codepoint"
               class="net.sf.saxon.sort.CodepointCollator"/>
    <collation uri="http://www.microsoft.com/collation/caseblind"
               class="net.sf.saxon.sort.CodepointCollator"/>
    <collation uri="http://example.com/french" lang="fr" ignore-case="yes"/>
  </collations>
</configuration>
```

The `configuration` element takes a single attribute, `edition`, whose value indicates the Saxon edition in use: HE (home edition), PE (professional edition), or EE (enterprise edition). The default value is HE. If PE or EE is specified, then the appropriate license file must be installed.

The children of the `configuration` element may appear in any order.

The contents of the different sections of the configuration file are described in the following subsections.

- The <global> element
- The <xslt> element
- The <xquery> element
- The <xsd> element
- The <resources> element
- The <collations> element
- The <localizations> element

The <global> element

The `global` input element of the configuration file contains properties defining global configuration options.

Table 4.1.

allowExternalFunctions	true|false	True if calls to external Java or .NET functions are allowed
allowMultiThreading	true|false	True if `saxon:threads` attribute on `xsl:for-each` causes multi-threading under Saxon-EE; false to disable multi-threading. Default is true (but multi-threading only happens if explicitly requested using `saxon:threads`.
allowOldJavaUriFormat	true|false	True if reflexive calls to external Java functions are allowed to use the "liberal" syntax (for example "http://my.com/extensions/java.util.Date"). The default is to allow only the "strict" form of URI such as "java:java.util.Date"
collationUriResolver	Name of a class implementing CollationURIResolver	User-supplied class used to intepret collation URIs
collectionUriResolver	Name of a class implementing CollectionURIResolver	User-supplied class used for resolving the URI supplied to the `collection()` function
compileWithTracing	true|false	generates trace code in the expression tree, allowing a TraceListener to be used at run-time
defaultCollation	A collation URI	(Requires Saxon-PE.) The collation URI to be used when no explicit collation is requested.
defaultCollection	A collection URI	The collection URI to be used when no argument is passed to the `collection()` function.

dtdValidation	true|false	Controls whether DTD validation is applied to input files.
dtdValidationRecoverable	true|false	Controls whether DTD validation errors are recoverable or fatal.
errorListener	A Java class that implements javax.xml.transform.ErrorListener	Defines the default ErrorListener for reporting both compile-time and run-time errors
expandAttributeDefaults	true|false	Controls whether attribute default values found in a DTD or schema are expanded or not.
lazyConstructionMode	true|false	If true, causes temporary trees to be constructed lazily
lineNumbering	true|false	Controls whether line and column number information is maintained for input files
optimizationLevel	integer, 0 to 10	Defines the level of code optimization to be applied
preEvaluateDocFunction	true|false	If true, allows calls on `doc()` with a literal argument to be evaluated early, at compile time
preferJaxpParser	true|false	Relevant only on .NET, determines whether the Java Classpath parser is used in preference to the Microsoft .NET parser
recognizeUriQueryParameters	true|false	If true, and the standard URIResolver is in use, query parameters such as `val=strict` will be recognized on URIs supplied to the `doc` or `document()` functions.
schemaValidation	strict|lax|preserve|skip	Controls whether schema validation is applied to input files
serializerFactory	Java class that extends net.sf.saxon.event.SerializerFactory	Allows the serialization pipeline to be customized, for example to handle user-defined serialization parameters
sourceParser	Java class implementing XMLReader	The SAX parser to be used for reading source files.
sourceResolver	Java class name implementing net.sf.saxon.SourceResolver	Name of a user-supplied class that resolves unknown implementations of the JAXP Source class into a known implementation.
stripWhitespace	all|none|ignorable	Controls what whitespace is removed from input documents (all inter-element whitespace,

		no inter-element whitespace, or all inter-element whitespace in elements having element-only content models)
styleParser	Java Class implementing XMLReader	XML parser used for stylesheets and schema documents
timing	true|false	Outputs progress messages to System.err. Equivalent to the -t option on the command line
traceExternalFunctions	true|false	Provides diagnostics when external functions are dynamically loaded
traceListener	Java Class implementing net.sf.saxon.trace.TraceListener	User-defined class to be used for handling run-time trace output
traceOptimizerDecisions	true|false	Causes tracing of decisions made by the optimizer
treeModel	linkedTree|tinyTree|tinyTreeCondensed	Determines the tree model implementation used for input files: TinyTree, LinkedTree, or TinyTree(condensed)
uriResolver	Name of a JAXP URIResolver	The URIResolver to be used for deferencing URIs used in `xsl:include`, `xsl:import`, `doc()`, and `document()`.
usePiDisableOutputEscaping	true|false	When sending output to a user-defined content handler, indicates whether JAXP-defined processing instructions are used to signal the start and end of text in which output escaping is disabled
useTypedValueCache	true|false	If true, typed values of element and attribute nodes are cached in the TinyTree. Uses extra memory, may make execution faster.
validationComments	true|false	Only relevant when validationWarnings=true, indicates that validation error messages should where possible be added as comments in the document instance being validated rather than fatal errors.
validationWarnings	true|false	For result trees subjected to schema validation, indicates whether validation failures should be treated as warnings rather than fatal errors.
versionOfXml	1.0|1.1	Determines whether XML 1.0 or XML 1.1 rules are used for names. (1.0 means the rules before Edition 5)

xInclude	true|false	Controls whether XInclude processing is applied to input files

The <xslt> element

The `xslt` element of the configuration file contains properties specific to XSLT. Remember that these are defaults, they can always be overridden for specific queries or transformations. An attribute whose value is set to a zero-length string is ignored, the effect is the same as omitting the attribute.

Table 4.2.

messageReceiver	Java class that implements `net.sf.saxon.event.Receiver`	Destination of `xsl:message` output
outputUriResolver	Java class that implements the Saxon OutputURIResolver interface	Handles documents written using `xsl:result-document`
recoveryPolicy	recoverWithWarnings|recoverSilently|fail	Indicates how XSLT recoverable errors are handled (for example, ambiguous template rules)
schemaAware	true|false	Indicates whether stylesheet should be compiled to be able to handle schema-typed input, even if they contain no `xsl:import-schema` declaration
staticErrorListener	Java class that implements the JAXP ErrorListener interface	Receives reports of compile-time errors in a stylesheet
staticUriResolver	Java class that implements the JAXP URIResolver interface	User-defined class for dereferencing URIs on `xsl:include` or `xsl:import`
styleParser	Java class that implements XMLReader	XML parser used for stylesheet and schema modules
version	0.0, 2.0, or 2.1	XSLT language version to be supported by the processor. The value 0.0 indicates that the version is taken from the `xsl:stylesheet` element.
versionWarning	true|false	False suppresses the warning produced when the XSLT processor version is not the same as the version in the xsl:stylesheet element.

The `xslt` element may contain one or more `extensionElement` children defining namespaces used for extension instructions. The `extensionElement` element has the following attributes:

Table 4.3.

namespace	A namespace URI	The namespace URI of the extension instructions

factory	The name of Java class that implements the Saxon interface `ExtensionElementFactory`	Links to the implementation of the various extension instructions in the specified namespace.

The <xquery> element

The `xquery` element of the configuration file contains properties specific to XQuery. Remember that these are defaults, they can always be overridden for specific queries. An attribute whose value is set to a zero-length string is ignored, the effect is the same as omitting the attribute.

Table 4.4.

allowUpdate	true|false	Indicates whether XQuery Update syntax is accepted
constructionMode	preserve|strip	Default value for construction mode in the static context
defaultElementNamespace	A namespace URI	Default namespace for elements and types
defaultFunctionNamespace	A namespace URI	Default namespace for functions
emptyLeast	true|false	True if the empty sequence comes first in sort order
inheritNamespaces	true|false	Default value for "inherit namespaces" in the static context
moduleUriResolver	Class that implements the Saxon ModuleURIResolver interface	Used for locating query modules referenced by "import module"
preserveBoundarySpace	true|false	Policy for preserving boundary space within direct element content
preserveNamespaces	true|false	Default value for "preserve namespaces" in the static context
requiredContextItemType	An item type, e.g. document-node()	The required type for the context item
schemaAware	true|false	True if the query makes use of schema information
staticErrorListener	Class that implements JAXP ErrorListener	Receives notification of static errors occurring in a Query
version	1.0|1.1	Indicates whether XQuery 1.1 syntax is accepted

The <xsd> element

The `xsd` element of the configuration file contains properties defining how schema documents are compiled by Saxon.

Table 4.5.

occurrenceLimits	MMM,NNN	Two integers, comma-separated. Controls the limits applied to minOccurs and maxOccurs values in XSD content models.

schemaUriResolver	Class implementing the Saxon SchemaURIResolver interface	Controls the handling of URIs in `xs:include`, `xs:import` etc. declarations, and also in `xsi:schemaLocation`
useXsiSchemaLocation	true|false	Indicates whether the schema processor takes account of `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes appearing in the instance document
version	1.0|1.1	Inidcates whether XSD 1.1 syntax is accepted

The <resources> element

The `resources` element in the configuration file defines a number of resources that can be preloaded into the configuration. It contains child elements as detailed below, in any order; most of them can appear more than once.

Table 4.6.

externalObjectModel	Class that implements the ExternalObjectModel interface	Defines an external object model that can be used to provide input to Saxon (and in some cases receive output): for example DOM, JDOM, XOM, etc.
extensionFunction	Class that implements the IntegratedFunction interface	Defines an "integrated extension function" written to a specific Saxon API
schemaDocument	relative or absolute URI	A schema document to be preloaded into the Saxon schema cache
schemaComponentModel	relative or absolute URI	A schema component model document (previously exported by Saxon) allowing fast loading of a compiled schema

The <collations> element

The `collations` element in the configuration file defines a number of collations that can be preloaded into the configuration. It contains zero or more child `collation` elements as detailed below.

Each `collation` element may have the following attributes:

Table 4.7.

uri	The collation URI (mandatory)	An absolute URI used to identify the collation in queries and stylesheets
class	Java class implementing Collator, StringCollator, or Comparator	Class used to perform string comparisons

lang	Language code, eg. en-US	Language supported by the collation
rules	Rules in Java RuleBasedCollator format	Detailed rules for ordering of characters
strength	primary|secondary|tertiary|identical	The strength of the collation. A stronger collation takes more details of the character into account, e.g. accents and case
ignore-case	yes|no	Yes indicates that upper-case and lower-case are equivalent
ignore-modifiers	yes|no	Yes indicates that accents and other modifiers are ignored
ignore-symbols	yes|no	Yes indicates that punctuation symbols are ignored (.NET only)
ignore-width	yes|mp	Yes indicates that width variations between characters are ignored
decomposition	none|standard|full	Determines whether Unicode normalization should be applied to strings before comparison (Java platform only)
case-order	upper-first|lower-first|#default	Indicates whether upper-case characters should precede or follow their lower-case equivalents
alphanumeric	yes|no	Yes indicates that a sequence of digits within a string is read as a number, for example "test8.xml" precedes "test10.xml"

The <localizations> element

The `localizations` element in the configuration file defines classes used to localize the output of the `xsl:number` instruction in XSLT, and the functions `format-date()`, `format-time()`, and `format-dateTime()`.

It has two attributes, `defaultLanguage` and `defaultCountry` which provide default values for the `lang` attribute/argument and the `country` argument respectively. If no values are supplied, the defaults are taken from the default Locale in the Java VM (which in turn will typically depend on operating system settings).

The element contains zero or more child `localization` elements as detailed below.

Each `localization` element may have the following attributes:

Table 4.8.

lang	An ISO language code, for example "en" or "fr-CA" (mandatory)	The language to which this localization relates
class	The name of a class that implements the interface `net.sf.saxon.number.Numberer`	The class that performs localization for the specified language

additional attributes	Additional attributes are passed on to a user-specified `LocalizationFactory`	The meaning of these attributes depends entirely on the `LocalizationFactory`.

Note that numberers for various European languages (da, de, fr, dr-BE, it, nl, nl-BE, sv) are supplied in package `net.sf.saxon.option.local`. In Saxon-PE and Saxon-EE these are compiled into the standard JAR file, but they are not configured by default. In Saxon-HE they are not built-in to the product, but can be integrated from the supplied source code in the same way as user-written localizations.

Configuration Features

This page provides a complete list of the configuration features available.

The properties are identified by a symbolic name and a URI value defined in the Java module `FeatureKeys [Javadoc: net.sf.saxon.lib.FeatureKeys]`. The table below gives summary information for each property, together with a link to the Javadoc, where more complete information is held.

Chapter 5. Using XSLT 2.0

Using XSLT 2.0 Stylesheets

This section describes how to use Saxon XSLT 2.0 stylesheets, either from the command line, or from the Java API, or from Ant.

It also describes the subset of XSLT 3.0 (previously known as XSLT 2.1) that Saxon currently implements.

Running XSLT from the Command Line

A command is available to apply a given stylesheet to a given source XML document. For simple transformations on the Java platform, use the command:

java net.sf.saxon.Transform -s: -xsl: -o:

where , , and are the source XML file, the XSLT stylesheet, and the output file respectively.

For the .NET platform, the command is simply:

Transform -s: -xsl: -o:

For a schema-aware transformation, specify the option `-sa`, or (on the Java platform only) use the alternate entry point `com.saxonica.Transform`. For more details see Schema-Aware Transformations.

For backwards compatibility with previous releases, the prefixes "-s:" and "-xsl:" can be omitted provided that the source document and the stylesheet are the last two options before any keyword=value parameters.

More generally, the arguments consist of a number of options prefixed with "-", then optionally (for backwards compatibility) the source filename and/or stylesheet filename, then a number of parameters provided as keyword=value pairs. The options must come first, then the file names if present, then the parameters.

For this to work, all the necessary Java components must be available on the classpath. See Installation for details of how to set up the classpath.

If you are are not using any additional Java libraries, you can use the simpler form of command (this example is for the Home Edition):

java -jar /saxon9he.jar [options] [params]

The options are as follows (in any order):

Table 5.1.

-a[:(on|off)]	Use the xml-stylesheet processing instruction in the source document to identify the stylesheet to be used. The stylesheet argument must not be present on the command line.
-catalog:filenames	is either a file name or a list of file names separated by semicolons; the files are OASIS XML catalogs used to define how public identifiers and system identifiers (URIs) used in a source document, stylesheet, or schema are to be redirected, typically to resources

	available locally. For more details see Using XML Catalogs.
-config:filename	Indicates that configuration information should be taken from the supplied configuration file. Any options supplied on the command line override options specified in the configuration file.
-cr:classname	Use the specified CollectionURIResolver to process collection URIs passed to the `collection()` function. The CollectionURIResolver is a user-defined class that implements the `net.sf.saxon.CollectionURIResolver` interface.
-dtd:(on|off|recover)	Setting `-dtd:on` requests DTD-based validation of the source file and of any files read using the document() function. Requires an XML parser that supports validation. The setting `-dtd:off` (which is the default) suppresses DTD validation. The setting `-dtd:recover` performs DTD validation but treats the error as non-fatal if it fails. Note that any external DTD is likely to be read even if not used for validation, because DTDs can contain definitions of entities.
-expand:(on|off)	Normally, if validation using a DTD or Schema is requested, any fixed or default values defined in the DTD or schema will be expanded. Specifying -expand:off suppresses this. (In the case of DTD-defined defaults, this might not work with all XML parsers. It does work with the Xerces parser (default for Java) and the Microsoft parser (default for .NET))
-explain[:filename]	Display an execution plan for the stylesheet. This is a representation of the expression tree after rewriting by the optimizer. It compbines the XSLT instructions and the XPath expressions into a single tree. If no file name is specified the output is sent to the standard error stream. The output is a tree in XML format.
-ext:(on|off)	If `ext:off` is specified, suppress calls on dynamically-loaded external Java functions. This does not affect calls on integrated extension functions, including Saxon and EXSLT extension functions. This option is useful when loading an untrusted stylesheet, perhaps from a remote site using an `http://` URL; it ensures that the stylesheet cannot call arbitrary Java methods and thereby gain privileged access to resources on your machine.
-im:modename	Selects the initial mode for the transformation. If this is namespaced, it can be written as `{uri}localname`
-init:initializer	The value is the name of a user-supplied class that implements the interface `net.sf.saxon.Initializer`; this initializer will be called during the

	initialization process, and may be used to set any options required on the Configuration programmatically. It is particularly useful for such tasks as registering extension functions, collations, or external object models, especially in Saxon-HE where the option does not exist to do this via a configuration file.
-it:template	Selects the initial named template to be executed. If this is namespaced, it can be written as `{uri}localname`. When this option is used, you do not need to supply a source file, but if you do, you must supply it using the `-s` option.
-l[:(on|off)]	If `-l` or `-l:on` is specified, causes line and column numbers to be maintained for source documents. These are accessible using the extension functions `saxon:line-number()` and `saxon:column-number()`. Line numbers are useful when the purpose of the stylesheet is to find errors or anomalies in the source XML file. Without this option, line numbers are available while source documents are being parsed and validated, but they are not retained in the tree representation of the document.
-m:classname	Use the specified Receiver to process the output from xsl:message. The class must implement the `net.sf.saxon.event.Receiver` class. This interface is similar to a SAX ContentHandler, it takes a stream of events to generate output. In general the content of a message is an XML fragment. By default the standard XML emitter is used, configured to write to the standard error stream, and to include no XML declaration. Each message is output as a new document.The sequence of calls to this Receiver is as follows: there is a single `open()` call at the start of the transformation, and a single `close()` call at the end; and each evaluation of an `xsl:message` instruction starts with a `startDocument()` call and ends with `endDocument()`. The `startDocument()` event has a `properties` argument indicating whether `terminate="yes"` was specified, and the `locationId` on calls such as `startElement()` and `characters()` can be used to identify the location in the stylesheet where the message data originated (this is achieved by passing the supplied `locationId` in a call to `getPipelineConfiguration().getLocator().getSy` or to `getLineNumber()` on the same object).Select the class `net.sf.saxon.event.MessageWarner` to have `xsl:message` output notified to the JAXP `ErrorListener`, as described in the JAXP documentation.

-now:yyyy-mm-ddThh:mm:ss+hh:mm	Sets the value of `current-dateTime()` (and `implicit-timezone()`) for the transformation. This is designed for testing, to enable repeatable results to be obtained for comparison with reference results, or to test that stylesheets can handle significant dates and times such as end-of-year processing.
-o:filename	Send output to named file. In the absence of this option, the results go to standard output. If the source argument identifies a directory, this option is mandatory and must also identify a directory; on completion it will contain one output file for each file in the source directory. If the stylesheet writes secondary output files using the `xsl:result-document` instruction; this filename acts as the base URI for the `href` attribute of this instruction. In the absence of this option, secondary output files are written relative to the current working directory. The file is created if it does not already exist; any necessary directories will also be created. If the file does exist, it is overwritten (even if the transformation fails); but not if the transformation produces no principal result tree.
-opt:0...10	Set optimization level. The value is an integer in the range 0 (no optimization) to 10 (full optimization); currently all values other than 0 result in full optimization but this is likely to change in future. The default is full optimization; this feature allows optimization to be suppressed in cases where reducing compile time is important, or where optimization gets in the way of debugging, or causes extension functions with side-effects to behave unpredictably. (Note however, that even with no optimization, lazy evaluation may still cause the evaluation order to be not as expected.)
-or:classname	Use the specified OutputURIResolver to process output URIs appearing in the `href` attribute of `xsl:result-document`. The OutputURIResolver is a user-defined class that implements the `net.sf.saxon.OutputURIResolver` interface.
-outval:(recover|fatal)	Normally, if validation of result documents is requested, a validation error is fatal. Setting the option `-outval:recover` causes such validation failures to be treated as warnings. The validation message is written both to the standard error stream, and (where possible) as a comment in the result document itself.
-p[:(on|off)]	Use the PTreeURIResolver. This option is available in Saxon-PE and Saxon-EE only. It cannot be used in conjunction with the -r option, and it automatically switches on the -u and -sa options. The effect is twofold. Firstly, Saxon-

	specific file extensions are recognized in URIs (including the URI of the source document on the command line). Currently the only Saxon-specific file extension is `.ptree`, which indicates that the source document is supplied in the form of a Saxon PTree. This is a binary representation of an XML document, designed for speed of loading. Secondly, Saxon-specific query parameters are recognized in a URI. Currently the only query parameter that is recognized is `val`. This may take the values `strict`, `lax`, or `strip`. For example, `source.xml?val=strict` loads a document with strict schema validation.
-r:classname	Use the specified URIResolver to process all URIs. The URIResolver is a user-defined class, that extends the net.sf.saxon.URIResolver class, whose function is to take a URI supplied as a string, and return a SAX InputSource. It is invoked to process URIs used in the document() function, in the xsl:include and xsl:import elements, and (if -u is also specified) to process the URIs of the source file and stylesheet file provided on the command line.
-repeat:integer	Performs the transformation N times, where N is the specified integer. This option is useful for performance measurement, since timings for the first transformation are often dominated by Java warm-up time.
-s:filename	Identifies the source file or directory. Mandatory unless the `-it` option is used. The source file is parsed to create a tree, and the document node of this tree acts as the initial context item for the transformation.If the name identifies a directory, all the files in the directory will be processed individually. In this case the `-o` option is mandatory, and must also identify a directory, to contain the corresponding output files. A directory must be specified as a filename, not as a URL.The source-document can be specified as "-" to take the source from standard input.
-sa	Invoke a schema-aware transformation. Requires Saxon-EE to be installed. This options is not needed if either (a) another option implying schema-awareness is present (for example `-val:strict`) or (b) the stylesheet contains an `xsl:import-schema` declaration.
-strip:(all|none|ignorable)	Specifies what whitespace is to be stripped from source documents (applies both to the principal source document and to any documents loaded for example using the `document()` function. The default is `none`: no whitespace stripping. Specifying `all` strips all whitespace text nodes from source documents before any further processing, regardless of any `xsl:strip-space` declarations in the stylesheet, or any `xml:space` attributes in

	the source document.Specifying `ignorable` strips all ignorable whitespace text nodes from source documents before any further processing, regardless of any `xsl:strip-space` declarations in the stylesheet, or any `xml:space` attributes in the source document. Whitespace text nodes are ignorable if they appear in elements defined in the DTD or schema as having element-only content.
-t	Display version and timing information to the standard error output. The output also traces the files that are read and writing, and extension modules that are loaded.
-T[:classname]	Display stylesheet tracing information. This traces execution of each instruction in the stylesheet, so the output can be quite voluminous. Also switches line numbering on for the source document. If a classname is specified, it is a user-defined class, which must implement net.sf.saxon.trace.TraceListener. If the classname is omitted, a system-supplied trace listener is used.For performance profiling, set classname to `net.sf.saxon.trace.TimedTraceListener`. This creates an output file giving timings for each instruction executed. This output file can subsequently be analyzed to give an execution time profile for the stylesheet. See Performance Analysis.
-threads:N	Used only when the `-s` option specifies a directory. Controls the number of threads used to process the files in the directory. Each transformation runs in a single thread.
-TJ	Switches on tracing of the binding of calls to external Java methods. This is useful when analyzing why Saxon fails to find a Java method to match an extension function call in the stylesheet, or why it chooses one method over another when several are available.
-TP:filename	This is equivalent to setting `-T:net.sf.saxon.trace.TimedTraceListener` and `-traceout:filename`; that is, it causes trace profile information to be set to the specified file. This output file can subsequently be analyzed to give an execution time profile for the stylesheet. See Performance Analysis.
-traceout:filename	Indicates that the output of the `trace()` function should be directed to a specified file. Alternatively, specify #out to direct the output to System.out, #err to send it to System.err (the default), or #null to have it discarded. This option is ignored when a trace listener is in use: in that case, trace() output goes to the registered trace listener.
-tree:(linked|tiny|tinyc)	Selects the implementation of the internal tree model. -tree:tiny selects the "tiny tree" model

	(the default). -tree:linked selects the linked tree model. -tree:tinyc selects the "condensed tiny tree" model. See Choosing a tree model.
-u	Indicates that the names of the source document and the stylesheet document are URLs; otherwise they are taken as filenames, unless they start with "http:" or "file:", in which case they are taken as URLs
-val[:(strict|lax)]	Requests schema-based validation of the source file and of any files read using the document() or similar functions. Validation is available only with Saxon-EE, and this flag automatically switches on the -sa option. Specify `-val` or `-val:strict` to request strict validation, or `-val:lax` for lax validation.
-versionmsg:(on|off)	If versionmsg:off is specified, suppress version warnings. This suppresses the warning message that is normally issued (as required by the W3C specification) when running an XSLT 2.0 processor against a stylesheet that specifies `version="1.0"`.
-warnings:(silent|recover|fatal)	Indicates the policy for handling recoverable errors in the stylesheet: `silent` means recover silently, `recover` means recover after writing a warning message to the system error output, `fatal` means signal the error and do not attempt recovery. (Note, this does not currently apply to all errors that the XSLT recommendation describes as recoverable). The default is `recover`.
-x:classname	Use specified SAX parser for source file and any files loaded using the document() function. The parser must be the fully-qualified class name of a Java class that implements the org.xml.sax.Parser or org.xml.sax.XMLReader interfaceOne use of this option is to select an HTML parser such as John Cowan's TagSoup rather than an XML parser. In this case, the TagSoup JAR file must be on the classpath, and the class name to use is `org.ccil.cowan.tagsoup.Parser`.Another common use is to specify `org.apache.xml.resolver.tools.ResolvingXMLRea` This parser is provided by the Apache commons project, and it customizes the default parser by using an `EntityResolver` that resolves external entity references (notable the reference to an external DTD in a DOCTYPE declaration) by reference to an OASIS catalog file. This can be used to avoid repeated calls to external web servers (such as the W3C server) for commonly used DTDs such as the XHTML DTD.
-xi:(on|off)	Apply XInclude processing to all input XML documents (including schema and stylesheet modules as well as source documents). This currently only works when documents are parsed

	using the Xerces parser, which is the default in JDK 1.5 and later.
-xmlversion:(1.0|1.1)	If `-xmlversion:1.1` is specified, allows XML 1.1 and XML Namespaces 1.1 constructs. This option must be set if source documents using XML 1.1 are to be read, or if result documents are to be serialized as XML 1.1. This option also enables use of XML 1.1 constructs within the stylesheet itself.
-xsd:file1;file2;file3...	Loads additional schema documents. The declarations in these schema documents are available when validating source documents (or for use by the `validate{}` expression). This option may also be used to supply the locations of schema documents that are imported into the stylesheet, in the case where the `xsl:import-schema` declaration gives the target namespace of the schema but not its location.
-xsdversion:(1.0|1.1)	If `-xsdversion:1.1` is specified, allows XML Schema 1.1 constructs such as assertions. This option must be set if schema documents using XML Schema 1.1 are to be read.
-xsiloc:(on|off)	If set to "on" (the default) the schema processor attempts to load any schema documents referenced in `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes in the instance document, unless a schema for the specified namespace (or non-namespace) is already available. If set to "off", these attributes are ignored.
-xsl:filename	Specifies the file containing the principal stylesheet module. Mandatory unless the -a option or -c option is used. The value "-" identifies the standard input stream. If the -u option is specified then the value must be a URI rather than a filename.
-xsltversion:(2.0|3.0)	Determines whether an XSLT 2.0 processor or XSLT 3.0 processor is to be used. By default the value is taken from the version attribute of the `xsl:stylesheet` element
-y:classname	Use specified SAX parser for stylesheet file, including any loaded using `xsl:include` or `xsl:import`. The parser must be the fully-qualified class name of a Java class that implements the `org.xml.sax.Parser` or `org.xml.sax.XMLReader` interface
--:value	Set a feature defined in the `Configuration` interface. The names of features are defined in the Javadoc for class `FeatureKeys [Javadoc: net.sf.saxon.lib.FeatureKeys]`: the value used here is the part of the name after the last "/", for example `--allow-external-functions:off`. Only features accepting a string or boolean may be set; for booleans the

	values true/false, on/off, yes/no, and 1/0 are recognized.
-?	Display command syntax

A takes the form `name=value`, being the name of the parameter, and the value of the parameter. These parameters are accessible within the stylesheet as normal variables, using the `$name` syntax, provided they are declared using a top-level `xsl:param` element. If there is no such declaration, the supplied parameter value is silently ignored. If the `xsl:param` element has an `as` attribute indicating the required type, then the string value supplied on the command line is cast to this type: this may result in an error, for example if an integer is required and the supplied value cannot be converted to an integer.

A preceded by a leading question mark (?) is interpreted as an XPath expression. For example, `?time=current-dateTime()` sets the value of the stylesheet parameter `$time` to the value of the current date and time, as an instance of `xs:dateTime`, while `?debug=false()` sets the value of the parameter `$debug` to the boolean value `false`. If the parameter has a required type (for example `<xsl:param name="p" as="xs:date"/>`), then the supplied value must be compatible with this type according to the standard rules for converting variable values and function arguments. The static context for this XPath expression includes only the standard namespaces conventionally bound to the prefixes `xs`, `fn`, `xsi`, and `saxon`. The static base URI (used when calling the `doc()` function) is the current directory. The dynamic context contains no context item, position, or size, and no variables.

A preceded by a leading exclamation mark (!) is interpreted as an output parameter. For example, `!indent=yes` requests indented output. This is equivalent to specifying the attribute `indent="yes"` on an `xsl:output` declaration in the stylesheet. An output parameter specified on the command line overrides one specified within the stylesheet. For parameters `doctype-system`, `doctype-public`, and `saxon:next-in-chain`, a zero-length value is treated as "absent", that is, the effect is to cancel any value that was set within the stylesheet.

> If you are using the `bash` shell, you will need to escape "!" as "\!".

A preceded by a leading plus sign (+) is interpreted as a filename or directory. The content of the file is parsed as XML, and the resulting document node is passed to the stylesheet as the value of the parameter. If the parameter value is a directory, then all the immediately contained files are parsed as XML, and the resulting sequence of document nodes is passed as the value of the parameter. For example, `+lookup=lookup.xml` sets the value of the stylesheet parameter `lookup` to the document node at the root of the tree representing the parsed contents of the file `lookup.xml`.

Under most operating systems it is possible to supply a value containing spaces by enclosing it in double quotes, for example `name="John Smith"`. This is a feature of the operating system shell, not something Saxon does, so it may not work the same way under every operating system or command processor. (In the jEdit console plugin, for example, it has to be written as `"name=John Smith"`)

If the parameter name is in a non-null namespace, the parameter can be given a value using the syntax `{uri}localname=value`. Here `uri` is the namespace URI of the parameter's name, and `localname` is the local part of the name.

This applies also to output parameters. For example, you can set the indentation level to 4 by using the parameter `!{http://saxon.sf.net/}indent-spaces=4`. In this case, however, lexical QNames using the prefix "saxon" are also recognized, for example `!saxon:indent-spaces=4`. See also Additional serialization parameters.

If the `-a` option is used, the name of the stylesheet is omitted. The source document must contain a `<?xml-stylesheet?>` processing instruction before the first element start tag; this processing instruction must have a pseudo-attribute `href` that identifies the relative or absolute URL of the stylsheet document, and a pseudo-attribute type whose value is `text/xml`, `application/xml`, or `text/xsl`. For example:

<?xml-stylesheet type="text/xsl" href="../style3.xsl" ?>

It is also possible to refer to a stylesheet embedded within the source document, provided it has an id attribute and the id attribute is declared in the DTD as being of type ID. For example:

```
<?xml-stylesheet type="text/xsl" href="#style1" ?>
<!DOCTYPE BOOKLIST SYSTEM "books.dtd"
  <!ATTLIST xsl:transform id ID #IMPLIED>
<
<BOOKLIST>
  ...
  <xsl:transform id="style1" version="1.0" xmlns:xsl="...">
  ...
  </xsl:transform>
</BOOKLIST>
```

Compiling a Stylesheet

The facility to "compile" a stylesheet to a binary disk representation (the `CompileStylesheet` command and the -c option of `net.sf.saxon.Transform`) has been dropped with effect from Saxon 9.4.

Saxon now generates Java bytecode automatically when stylesheets are processed, but the bytecode cannot be saved to disk, because it is integrated with interpreted data structures representing the executable stylesheet.

The performance boost achieved by bytecode generation is variable; 25% is typical. The constructs that benefit the most are those where the expression tree contains many constructs that are relatively cheap in themselves, such as type conversion, comparisons, and arithmetic. This is because the saving from bytecode generation is mainly not in the cost of performing primitive operations, but in the cost of deciding which operations to perform: so the saving is greater where the number of operations is high relative to their average cost.

There are configuration options to suppress bytecode generation (`FeatureKeys.GENERATE_BYTE_CODE`), to insert debugging logic into the generated bytecode (`FeatureKeys.DEBUG_BYTE_CODE`), and to display the generated bytecode (`FeatureKeys.DISPLAY_BYTE_CODE`)

A new facility for distributing an obfuscated stylesheet is available (this was the main use case for compiled stylesheets, since there were negligible performance benefits). This is described in the next section.

Packaged Stylesheets

A new facility is available in Saxon 9.4 allowing an XSLT stylesheet to be distributed to users in a format such that the source code of the stylesheet is not readily visible. Creating a packaged stylesheet requires Saxon-PE or Saxon-EE; running the stylesheet only requires Saxon-HE, provided that it only uses features that are available with Saxon-HE.

This feature is not currently available on .NET

The stylesheet is distributed in the form of a ZIP file; the format of the content is not strongly encrypted, but requires non-trivial effort to decompile. The ZIP package contains all the stylesheet modules that are needed.

To create the package for distribution, use the following command:

java com.saxonica.ptree.StylesheetPackager stylesheet.xsl package.zip

where `stylesheet.xsl` is the principal stylesheet module, and `package.zip` is the output file containing the packaged stylesheet.

Running the stylesheet requires both the normal Saxon executable JAR file (HE or higher), and also the JAR file `saxon9-unpack.jar` which is included as standard in each Saxon edition. This contains Saxonica-proprietary code issued in compiled form only (not open source), but it does not require a license key to run. To run the stylesheet, use a command such as the following (on a single line):

java -cp saxon9he.jar;saxon9-unpack.jar net.sf.saxon.Transform -xsl:package.zip -s:source.xml -r:com.saxonica.ptree.PackageURIResolver -u:on

In this command, the `-r` option sets a URIResolver that has the task of retrieving obfuscated stylesheet modules from the ZIP file, and the `-u` option ensures that this URIResolver is used for the main stylesheet module (here `package.zip`) as well as for resolving `xsl:include` and `xsl:import` references between modules.

Limitations

Some care is needed with operations that depend on the static base URI. This is interpreted as the location of the packaged stylesheet at the time it is executed, not the original location at the time it was compiled. However, there is no limitation on the URIs that can be used in `xsl:include` and `xsl:import` declarations - the original URIs appearing in the source code will be replaced with references to the obfuscated modules within the generated ZIP file.

Use of the `document('')` function to access the source stylesheet is best avoided.

There is no special handling of schema documents referenced by the stylesheet. The packaged stylesheet does not include any imported schema; this must be distributed separately. The version of the schema used for validating documents at run-time must be consistent with the version used for compiling the stylesheet.

Running Saxon XSLT Transformations from Ant

It is possible to run a Saxon transformation from Ant using the standard xslt task [http://ant.apache.org/manual/CoreTasks/style.html], by using the `trax` processor and with an appropriate `classpath` that forces Saxon to be selected. This can now be used to provide the ability to take advantage of the full range of capabilities offered by XSLT 2.0 in general and Saxon in particular (for example, schema aware processing and multiple output files).

The custom Ant task developed for earlier Saxon releases is not being further developed, although it remains available. It is no longer issued as an intrinsic part of the Saxon product, but can be downloaded as a separate package from SourceForge: see https://sourceforge.net/project/showfiles.php?group_id=29872. For information, see the documentation accompanying Saxon 9.2 or earlier releases.

Saxon-specific configuration options can be specified using the `attribute` child of the `factory` element. For example, the following switches on Saxon tracing (the equivalent of the -T option on the command line):

```
<factory name="net.sf.saxon.TransformerFactoryImpl">
   <attribute name="http://saxon.sf.net/feature/traceListenerClass"
              value="net.sf.saxon.trace.XSLTTraceListener"/>
```

```
  </factory>
```

For a full list of feature names, see the Javadoc documentation of class `net.sf.saxon.FeatureKeys`.

With Saxon-PE and Saxon-EE it is possible to get detailed control of the configuration by specifying the name of a Saxon configuration file using this mechanism, for example:

```
  <factory name="com.saxonica.config.EnterpriseTransformerFactory">
    <attribute name="http://saxon.sf.net/feature/configuration-file"
               value="config-de.xml"/>
  </factory>
```

In this case this must be the first attribute.

The initial template and initial mode for the transformation can be specified using the attribute names `http://saxon.sf.net/feature/initialTemplate` and `http://saxon.sf.net/feature/initialMode` respectively. The value is a QName in Clark notation (that is `{uri}local`).

Note that names in Clark notation may also be used for the qualified names of stylesheet parameters and serialization options.

Note that an Ant transformation always has an input file. If necessary, a dummy file can be specified.

There is a history of bugs in successive releases of Ant that mean not all these features work in every Ant version. In particular, the `classpath` attribute of the `xslt` task element has been unreliable: the safest approach is to ensure that the Jar files needed to run Saxon are present on the externally-specified classpath (the classpath at the point where Ant is invoked), rather than relying on the task-specific classpath.

Invoking XSLT from an application

Rather than using the XSLT interpreter from the command line, you may want to include it in your own application, perhaps one that enables it to be used within an applet or servlet. If you run the interpreter repeatedly, this will always be much faster than running it each time from a command line.

There are several APIs you can use to invoke Saxon's XSLT processor from an application.

The JAXP interface (Java)

On the Java platform, Saxon incorporates support for the standard JAXP transformation API [http://jaxp.java.net/] (originally known as TrAX). This is compatible with the API for invoking other XSLT processors such as Xalan and Oracle.

This API is described in the documentation provided with JDK 1.5 and later. It is available online at http://download.oracle.com/javase/6/docs/api/ Look for the javax.xml.transform [http://download.oracle.com/javase/6/docs/api/javax/xml/transform/package-summary.html] package.

The s9api interface (Java)

Alternatively, you can use Saxon's own interface. This is designed to provide an integrated approach to XML processing across the different range of languages supported by Saxon; unlike JAXP, it includes support for XSLT 2.0 capabilities, and it also takes advantage of generics in Java 5.

The Saxon.Api interface (.NET)

Links

- Using s9api for Transformations
- Using JAXP for Transformations

Using s9api for Transformations

You can perform a transformation using the s9api interface as follows:

1. Create a Processor (`net.sf.saxon.s9api.Processor`) and set any global configuration options on the Processor.
2. Call `newXsltCompiler()` to create an XSLT Compiler, and set any options that are local to a specific compilation (for example, the destination of error messages).
3. Call the `compile()` method to compile a stylesheet. The result is an `XsltExecutable`, which can be used as often as you like in the same thread or in different threads.
4. To run a transformation, call the `load()` method on the `XsltExecutable`. This creates an `XsltTransformer`. The `XsltTransformer` can be serially reused, but it must not be shared across multiple threads. Set any options required for the specific transformation (for example, the initial context node, the stylesheet parameters, and the destination for the results of the transformation), and then call the `transform()` method to run the transformation.

The output of the transformation is specified as a `Destination` object, which allows a wide range of possibilities: you can send the output to a serializer, or to a SAX ContentHandler. You can build a tree either in Saxon's native format (represented by the s9api class `XdmNode`) or as a DOM. You can send the output to be validated against a schema by nominating a `SchemaValidator` as the destination, or you can pipe it through another transformation, because `XsltTransformer` itself implements the `Destination` interface.

Examples of s9api transformations are included in the Saxon resources file, see module .

Using JAXP for Transformations

This API is described in the documentation provided with JDK 1.5 and later. It is available online at http://download.oracle.com/javase/6/docs/api/ Look for the javax.xml.transform [http://download.oracle.com/javase/6/docs/api/javax/xml/transform/package-summary.html] package.

More information and examples relating to the JAXP transformation API can be found in the example application found in the samples directory.

The class name for the JAXP `TransformerFactory` depends on which Saxon edition you are using:

- `net.sf.saxon.TransformerFactoryImpl [Javadoc: net.sf.saxon.TransformerFactoryImpl]`
- `com.saxonica.config.ProfessionalTransformerFactory [Javadoc: com.saxonica.config.ProfessionalTransformerFactory]`
- `com.saxonica.config.EnterpriseTransformerFactory [Javadoc: com.saxonica.config.EnterpriseTransformerFactory]`

Note that as an alternative to using `TransformerFactory.newInstance()` to find Saxon dynamically on the class path, it is possible (and much faster, and more robust) to instantiate the Saxon `TransformerFactory` directly, by a call such as `TransformerFactory factory = new com.saxonica.config.ProfessionalTransformerFactory()`

The types of object that can be supplied as stylesheet parameters are not defined in the JAXP specification: they are implementation-dependent. Saxon takes the Java object supplied, and converts it to an XPath value using the same rules as it applies for the return value from a Java extension function: for these rules, see Saxon Extensibility. If the resulting value is an atomic value, it is cast to the required type of the parameter as specified in the `xsl:param` declaration, using the XPath casting rules. If the value is non-atomic (for example, if it is a node, or a sequence of integers), then no conversion is attempted, instead, the value must match the required type as stated.

The JAXP `TransformerFactory` interface provides a configuration method `setAttribute()` for setting implementation-defined configuration parameters. The parameters supported by Saxon have names defined by constants in the class `net.sf.saxon.FeatureKeys`. The names of these properties and their meanings, are described in Configuration Features.

Where the required value of a property is a Boolean, the supplied value may be either a `java.lang.Boolean`, or a String holding the values "true" or "false" (also accepted are "on"|"off", "1"|"0", or "yes"|"no"). The returned value of the property, however, will be a `Boolean`.

Saxon's implementation of the JAXP `Transformer` interface is the class `net.sf.saxon.Controller [Javadoc: net.sf.saxon.Controller]`. This provides a number of options beyond those available in the standard JAXP interface, for example the ability to set an output URI resolver for secondary output documents, and a method to set the initial mode before the transformation starts. You can access these methods by casting the `Transformer` to a `Controller`. The methods are described in the JavaDoc documentation supplied with the product.

When using the JAXP interface, you can set serialization properties using a `java.util.Properties` object. The names of the core XSLT 1.0 properties, such as `method`, `encoding`, and `indent`, are defined in the JAXP class `javax.xml.transform.OutputKeys`. Additional properties, including Saxon extensions and XSLT 2.0 extensions, have names defined by constants in the class `net.sf.saxon.lib.SaxonOutputKeys [Javadoc: net.sf.saxon.lib.SaxonOutputKeys]`. The values of the properties are exactly as you would specify them in the `xsl:output` declaration, except that QNames are written in Clark notation (`{uri}local`).

Performance Analysis

Saxon comes with a simple tool allowing profiling of the execution time in a stylesheet.

To run this tool, first execute the transformation with the `-TP:filename` option, which will gather timed tracing information and create a profile report to the specified file (or to the standard error output if no filename is given)

java -jar /saxon9he.jar -TP:profile.html

Then view the resulting `profile.html` file in your browser.

The output identifies templates and functions in the original stylesheet by their name or match pattern, line number, and the last few characters of the URI of their module. For each instruction it gives the number of times the instruction was executed, the average time in milliseconds of each execution, and the total time. Timings are given both gross (the time for a template including all the templates it calls recursively), and net (the time in a template excluding time in its called templates). The table is sorted according to a weighting function that attempts to put the dominant functions and templates at the top. These will not necessarily be those with the greatest time, which tend to be instructions that were only executed once but remained active for the duration of the transformation.

XSLT 3.0 Support

Saxon 9.4 adds further support for a number of features defined in the draft XSLT 3.0 specification (previously known as XSLT 2.1) that were introduced in Saxon 9.3. It must be noted that this is an early working draft, and everything is subject to change. If the W3C specification changes, Saxon will change to match, without regards to backwards compatibility. Use it at your own risk.

All these features require at least Saxon-PE. Streaming requires Saxon-EE.

XSLT 3.0 support must be explicitly enabled, for example by specifying `version="3.0"` in the stylesheet or by using the option `-xsltversion:3.0` on the command line. It can also be enabled from the configuration file or using methods in the API (for example, on the s9api `XsltCompiler` object).

For details of the features implemented in the current Saxon release, see XSLT 3.0 Conformance. Full details of these features are in the W3C XSLT and XPath specifications; but summary information about some of them can be found here:

- Functions
- XSLT Elements
- XPath 3.0 Expressions
- Maps in XPath 3.0

Chapter 6. Using XQuery

Introduction

This section describes how to use Saxon as an XQuery processor, either from the command line, or from the Java API.

For details of the .NET API, see Saxon API for .NET

For information about the conformance of Saxon to the XQuery 1.0 and XQuery 3.0 specifications, and about the handling of implementation-defined features of the specifications, see Conformance.

Saxon uses the same run-time engine to support both XQuery and XSLT, reflecting the fact that the two languages have very similar semantics. Most of the compile-time code (in particular, the type checking logic and the optimizer) is also common. The XQuery support in Saxon consists essentially of an XQuery parser (which is itself an extension of the XPath parser); the parser generates the same internal interpretable code as the XSLT processor. There are also some constructs in the internal expression tree that will only be generated from XQuery source rather than XSLT source; examples are the XQuery `order by` and `group by` clauses, which have no direct XSLT equivalent.

The XQuery processor may be invoked either from the operating system command line, or via an API from a user-written application. There is no graphical user interface provided.

Saxon is an in-memory processor. Unless you can take advantage of streaming, Saxon is designed to process source documents that fit in memory. Saxon has been used successfully to process source documents of 100Mbytes or more without streaming, but if you attempt anything this large, you need to be aware (a) that you will need to allocate sufficient memory to the Java VM (at least 5 times the size of the source document), and (b) that complex FLWOR expressions may be very time-consuming to execute. (In this scenario, Saxon-EE is recommended, because it has a more powerful optimizer for complex joins).

The memory available in the Java heap is controlled using the -Xmx option on the command line, for example `java -Xmx1024m net.sf.saxon.Query ...` allocates 1Gb.

Running XQuery from the Command Line

A command is available to run a query contained in a file. The form of command on the Java platform is:

java net.sf.saxon.Query [options] -q:queryfile [ ]

On the .NET platform, the command is simply:

Query [options] -q:queryfile [ ]

The options must come first, then the params. If the last option before the params has no leading hyphen and option letter then it is recognized as the -q option.

The options are as follows (in any order). Square brackets indicate an optional value.

Table 6.1.

-backup:(on|off)	Only relevant when -update:on is specified. Default is on. When backup is enabled, any file that is updated by the query will be preserved in its original state by renaming it, adding ".bak" to the

	original filename. If backup is disabled, updated files will be silently overwritten.
-catalog:filenames	is either a file name or a list of file names separated by semicolons; the files are OASIS XML catalogs used to define how public identifiers and system identifiers (URIs) used in a source document, query, or schema are to be redirected, typically to resources available locally. For more details see Using XML Catalogs.
-config:filename	Indicates that configuration information should be taken from the supplied configuration file. Any options supplied on the command line override options specified in the configuration file.
-cr:classname	Use the specified CollectionURIResolver to process collection URIs passed to the `collection()` function. The CollectionURIResolver is a user-defined class that implements the `CollectionURIResolver [Javadoc: >net.sf.saxon.lib.CollectionURIResolver]` interface.
-dtd:(on|off|recover)	Setting `-dtd:on` requests DTD-based validation of the source file and of any files read using the document() function. Requires an XML parser that supports validation. The setting `-dtd:off` (which is the default) suppresses DTD validation. The setting `-dtd:recover` performs DTD validation but treats the error as non-fatal if it fails. Note that any external DTD is likely to be read even if not used for validation, because DTDs can contain definitions of entities.
-expand:(on|off)	Normally, if validation using a DTD or Schema is requested, any fixed or default values defined in the DTD or schema will be expanded. Specifying -expand:off suppresses this. (In the case of DTD-defined defaults, this might not work with all XML parsers. It does work with the Xerces parser (default for Java) and the Microsoft parser (default for .NET))
-explain[:filename]	Display a query execution plan. This is a representation of the expression tree after rewriting by the optimizer. If no file name is specified the output is sent to the standard error stream. The output is a tree in XML format.
-ext:(on|off)	If `ext:off` is specified, suppress calls on dynamically-loaded external Java functions. This does not affect calls on integrated extension functions, including Saxon and EXSLT extension functions. This option is useful when loading an untrusted query, perhaps from a remote site using an `http://` URL; it ensures that the query cannot call arbitrary Java methods and thereby gain privileged access to resources on your machine.

-init:initializer	The value is the name of a user-supplied class that implements the interface `Initializer` [Javadoc: `net.sf.saxon.lib.Initializer`]; this initializer will be called during the initialization process, and may be used to set any options required on the Configuration programmatically. It is particularly useful for such tasks as registering extension functions, collations, or external object models, especially in Saxon-HE where the option does not exist to do this via a configuration file.
-l[:(on|off)]	If `-l` or `-l:on` is specified, causes line and column numbers to be maintained for source documents. These are accessible using the extension functions `saxon:line-number()` and `saxon:column-number()`. Line numbers are useful when the purpose of the query is to find errors or anomalies in the source XML file. Without this option, line numbers are available while source documents are being parsed and validated, but they are not retained in the tree representation of the document.
-mr:classname	Use the specified ModuleURIResolver to process all query module URIs. The ModuleURIResolver is a user-defined class that implements the `ModuleURIResolver` [Javadoc: `net.sf.saxon.lib.ModuleURIResolver`] interface. It is invoked to process URIs used in the `import module` declaration in the query prolog, and (if -u is also specified, or if the file name begins with "http:" or "file:") to process the URI of the query source file provided on the command line.
-o:filename	Send output to named file. In the absence of this option, the results go to standard output. The output format depends on whether the -wrap option is present. The file is created if it does not already exist; any necessary directories will also be created. If the file does exist, it is overwritten (even if the query fails).
-opt:0...10	Set optimization level. The value is an integer in the range 0 (no optimization) to 10 (full optimization); currently all values other than 0 result in full optimization but this is likely to change in future. The default is full optimization; this feature allows optimization to be suppressed in cases where reducing compile time is important, or where optimization gets in the way of debugging, or causes extension functions with side-effects to behave unpredictably. (Note however, that even with no optimization, lazy evaluation may still cause the evaluation order to be not as expected.)
-outval:(recover|fatal)	Normally, if validation of result documents is requested, a validation error is fatal. Setting

	the option `-outval:recover` causes such validation failures to be treated as warnings. The validation message is written both to the standard error stream, and (where possible) as a comment in the result document itself.
-p[:(on|off)]	Use the PTreeURIResolver. This option is available in Saxon-EE only. It cannot be used in conjunction with the -r option, and it automatically switches on the -u option. The effect is twofold. Firstly, Saxon-specific file extensions are recognized in URIs (including the URI of the source document on the command line). Currently the only Saxon-specific file extension is `.ptree`, which indicates that the source document is supplied in the form of a Saxon PTree. This is a binary representation of an XML document, designed for speed of loading. Secondly, Saxon-specific query parameters are recognized in a URI. Currently the only query parameter that is recognized is `val`. This may take the values `strict`, `lax`, or `strip`. For example, `source.xml?validation=strict` loads a document with strict schema validation.
-pipe:(push|pull)	Execute query internally in push or pull mode. Default is push. This may give performance advantages for certain kinds of query, especially queries that construct intermediate trees in memory. In practice when the output is serialized there is usually little difference, and the option is there mainly for testing.
-projection:(on|off)	Use (or don't use) document projection. Document Projection is a mechanism that analyzes a query to determine what parts of a document it can potentially access, and then while building a tree to represent the document, leaves out those parts of the tree that cannot make any difference to the result of the query. Requires Saxon-EE.
-q:queryfile	Identifies the file containing the query. The file can be specified as "-" to read the query from standard input. If this is the last option then the "-q:" prefix may be omitted.
-qs:querystring	Allows the query to be specified inline (if it contains spaces, you will need quotes around the expression to keep the command line processor happy). For example `java net.sf.saxon.Query -qs:doc('a.xml')//p[1]` selects elements within the file a.xml in the current directory.
-qversion:(1.0|3.0)	Specifies the XQuery language version supported. Default is "1.0". The value "3.0" is currently only available with Saxon-EE. It supports a fairly small subset of the features from the draft XQuery 3.0

	specification (originally published as XQuery 1.1 but due to be renamed).
-r:classname	Use the specified URIResolver to process all URIs. The URIResolver is a user-defined class, that implements the URIResolver interface defined in JAXP, whose function is to take a URI supplied as a string, and return a SAX InputSource. It is invoked to process URIs used in the doc() function, and (if -u is also specified) to process the URI of the source file provided on the command line.
-repeat:integer	Performs the transformation N times, where N is the specified integer. This option is useful for performance measurement, since timings for the first few runs of the query are often dominated by Java warm-up time.
-s:filename-or-URI	Take input from the specified file. If the -u option is specified, or if the name begins with "file:" or "http:", then the name is assumed to be a URI rather than a filename. This file must contain an XML document. The document node of the document is made available to the query as the context item. The source document can be specified as "-" to take the source from standard input.
-sa	Invoke a schema-aware query. Requires Saxon-EE to be installed.
-strip:(all|none|ignorable)	Specifies what whitespace is to be stripped from source documents (applies both to the principal source document and to any documents loaded for example using the `doc()` function. The default is `none`: no whitespace stripping.
-t	Display version and timing information to the standard error output. The output also traces the files that are read and written, and extension modules that are loaded.
-T[:classname]	Notify query tracing information. Also switches line numbering on for the source document. If a classname is specified, it is a user-defined class, which must implement `TraceListener [Javadoc: net.sf.saxon.lib.TraceListener]`. If the classname is omitted, a system-supplied trace listener is used. This traces execution of function calls to the standard error output.
-TJ	Switches on tracing of the binding of calls to external Java methods. This is useful when analyzing why Saxon fails to find a Java method to match an extension function call in the stylesheet, or why it chooses one method over another when several are available.
-traceout:filename	Indicates that the output of the `trace()` function should be directed to a specified file. Alternatively, specify #out to direct the output

	to System.out, #err to send it to System.err (the default), or #null to have it discarded. This option is ignored when a trace listener is in use: in that case, trace() output goes to the registered trace listener.
-tree:(linked|tiny|tinyc)	Selects the implementation of the internal tree model. -tree:tiny selects the "tiny tree" model (the default). -tree:linked selects the linked tree model. -tree:tinyc selects the "condensed tiny tree" model. See Choosing a tree model.
-u	Indicates that the name of the source document is a URI; otherwise it is taken as a filename, unless it starts with "http:" or "file:", in which case they it is taken as a URL.
-update:(on|off|discard)	Indicates whether XQuery Update syntax is accepted. This option requires Saxon-EE. The value "on" enables XQuery update; any elegible files updated by the query are written back to filestore. A file is eligible for updating if it was read using the `doc()` or `collection()` functions using a URI that represents an updateable location. The context document supplied using the `-s` option is eligible for updating. The default value "off" disables update (any use of XQuery update syntax is an error). The value "discard" allows XQuery Update syntax, but modifications made to files are not saved in filestore. If the document supplied in the -s option is updated, the updated document is serialized as the result of the query (writing it to the -o destination); updates to any other documents are simply discarded.Use of the `-t` option is recommended: it gives feedback on which files have been updated by the query.
-val[:(strict|lax)]	Requests schema-based validation of the source file and of any files read using the doc() function. This option is available only with Saxon-EE, and it automatically switches on the -sa option. Specify `-val` or `-val:strict` to request strict validation, or `-val:lax` for lax validation.
-wrap	Wraps the result sequence in an XML element structure that indicates the type of each node or atomic value in the query result. This format can handle any type of query result. In the absence of this option, the command effectively wraps a `document{}` constructor around the supplied query, so that the result is a single XML document, which is then serialized. This will fail if the query result includes constructs that cannot be added to a document node in this way, notably free-standing attribute nodes.
-x:classname	Use specified SAX parser for source file and any files loaded using the document() function. The parser must be the fully-qualified class name of a

	Java class that implements the org.xml.sax.Parser or org.xml.sax.XMLReader interface
-xi:(on|off)	Apply XInclude processing to all input XML documents (including schema documents as well as source documents). This currently only works when documents are parsed using the Xerces parser, which is the default in JDK 1.5 and later.
-xmlversion:(1.0|1.1)	If `-xmlversion:1.1` is specified, allows XML 1.1 and XML Namespaces 1.1 constructs. This option must be set if source documents using XML 1.1 are to be read, or if result documents are to be serialized as XML 1.1. This option also enables use of XML 1.1 constructs within the stylesheet itself.
-xsd:file1;file2;file3...	Loads additional schema documents. The declarations in these schema documents are available when validating source documents (or for use by the `validate{}` expression). This option may also be used to supply the locations of schema documents that are imported into the query, in the case where the `import schema` declaration gives the target namespace of the schema but not its location.
-xsdversion:(1.0|1.1)	If `-xsdversion:1.1` is specified, allows XML Schema 1.1 constructs such as assertions. This option must be set if schema documents using XML Schema 1.1 are to be read.
-xsiloc:(on|off)	If set to "on" (the default) the schema processor attempts to load any schema documents referenced in `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes in the instance document, unless a schema for the specified namespace (or non-namespace) is already available. If set to "off", these attributes are ignored.
--:value	Set a feature defined in the `Configuration` interface. The names of features are defined in Configuration Features: the value used here is the part of the name after the last "/", for example `--allow-external-functions:off`. Only features accepting a string or boolean may be set; for booleans the values true/false, on/off, yes/no, and 1/0 are recognized.
-?	Display command syntax

A takes the form `name=value`, being the name of the parameter, and the value of the parameter. These parameters are accessible within the query as external variables, using the `$name` syntax, provided they are declared in the query prolog. If there is no such declaration, the supplied parameter value is silently ignored.

A preceded by a leading question mark (?) is interpreted as an XPath expression. For example, `?time=current-dateTime()` sets the value of the external variable `$time` to the value of the current date and time, as an instance of `xs:dateTime`, while `?debug=false()` sets the value of the variable `$debug` to the boolean value `false`. If the parameter has a required type (for example `declare variable $p as xs:date external;`), then the supplied value must be compatible with this type according to the standard rules for converting function arguments (it doesn't need to satisfy the stricter rules that apply to variable initialization). The static context for the XPath expression includes only the standard namespaces conventionally bound to the prefixes `xs`, `fn`, `xsi`, and `saxon`. The static base URI (used when calling the `doc()` function) is the current directory. The dynamic context contains no context item, position, or size, and no variables.

A preceded by a leading plus sign (+) is interpreted as a filename or directory. The content of the file is parsed as XML, and the resulting document node is passed to the stylesheet as the value of the parameter. If the parameter value is a directory, then all the immediately contained files are parsed as XML, and the resulting sequence of document nodes is passed as the value of the parameter. For example, `+lookup=lookup.xml` sets the value of the external variable `lookup` to the document node at the root of the tree representing the parsed contents of the file `lookup.xml`.

A preceded by a leading exclamation mark (!) is interpreted as a serialization parameter. For example, `!indent=yes` requests indented output, and `!encoding=iso-8859-1` requests that the serialized output be in ISO 8859/1 encoding. This is equivalent to specifying the option declaration `declare option saxon:output "indent=yes";` or `declare option saxon:output "encoding=iso-8859-1";` in the query prolog.

If you are using the `bash` shell, you will need to escape "!" as "\!".

Under Windows, and some other operating systems, it is possible to supply a value containing spaces by enclosing it in double quotes, for example `name="John Smith"`. This is a feature of the operating system shell, not something Saxon does, so it may not work the same way under every operating system.

If the parameter name is in a non-null namespace, the parameter can be given a value using the syntax `{uri}localname=value`. Here `uri` is the namespace URI of the parameter's name, and `localname` is the local part of the name.

This applies also to output parameters. For example, you can set the indentation level to 4 by using the parameter `!{http://saxon.sf.net/}indent-spaces=4`. In this case, however, lexical QNames using the prefix "saxon" are also recognized, for example `!saxon:indent-spaces=4`. For the extended set of output parameters supported by Saxon, see Additional serialization parameters.

Running Queries from a Java Application

Saxon offers three different APIs allowing queries to be run from Java:

- The XQJ interface is an implementation of the XQuery API for Java (XQJ) interface defined in JSR-225
- The s9api interface is a Saxon-specific interface allowing integrated access to all Saxon's XML processing capabilities in a uniform way, taking advantage of the type safety offered by generics in Java 5
- There is also a legacy interface retained from previous Saxon releases, which may be appropriate if you need access to lower-level Saxon internals. This is described only in the JavaDoc: start at `StaticQueryContext [Javadoc: net.sf.saxon.query.StaticQueryContext]`.

Links

- Using s9api for XQuery
- Invoking XQuery using the XQJ API

Using s9api for XQuery

You can perform a query using the s9api interface as follows:

1. Create a `Processor [Javadoc: net.sf.saxon.s9api.Processor]`) and set any global configuration options on the Processor.
2. Optionally, build the source document by calling `newDocumentBuilder()` to create a document builder, setting appropriate options, and then calling the `build()` method. This returns an `XdmNode [Javadoc: net.sf.saxon.s9api.XdmNode]` which can be supplied as input to the query either as the context item, or as the value of an external variable.
3. Call `newXQueryCompiler()` to create an XQuery Compiler. Then set any options that are local to a specific compilation (for example, the destination of error messages, the base URI, or the character encoding of the query text).
4. Call one of the `compile()` methods to compile a query. The result is an `XQueryExecutable`, which can be used as often as you like in the same thread or in different threads.
5. To run a query, call the `load()` method on the `XQueryExecutable`. This creates an `XQueryEvaluator [Javadoc: net.sf.saxon.s9api.XQueryEvaluator]`. The `XQueryEvaluator` can be serially reused, but it must not be shared across multiple threads. Set any options required for the specific query execution (for example, the initial context node, the values of external variables, and the destination for the query result), and then call either the `iterator()` or the `run()` method to execute the query.
6. Because the `XQueryEvaluator` is an `Iterable`, it is possible to iterate over the results directly using the Java 5 "for-each" construct.

The output of the query may be retrieved as an iterator over a sequence of items, or it may is specified as a `Destination [Javadoc: net.sf.saxon.s9api.Destination]` object, which allows a wide range of possibilities: you can send the output to a serializer, or to a SAX ContentHandler. You can build a tree either in Saxon's native format (represented by the s9api class `XdmNode`) or as a DOM. You can send the output to be validated against a schema by nominating a `SchemaValidator [Javadoc: net.sf.saxon.s9api.SchemaValidator]` as the destination, or you can pipe it through an XSLT transformation, because `XsltTransformer [Javadoc: net.sf.saxon.s9api.XsltTransformer]` also implements the `Destination` interface.

Examples of s9api queries are included in the Saxon resources file, see module S9APIExamples.java.

Separate compilation of library modules

Under Saxon-EE, it is possible to compile library modules separately from the main module. This reduces the compilation time and memory usage when the same library module is imported by many main modules for different queries. A method `compileLibrary()` (with a number of overloaded variants supplying the input in different ways) is provided in the `XQueryCompiler` class; any library module compiled using this method will be available to all subsequent compilations using the same `XQueryCompiler`. To import the module, simply use `import module` specifying the module URI in the normal way. It is not necessary to supply a module location hint (`at "URI"`), and if any is supplied, it will be ignored.

Invoking XQuery using the XQJ API

XQJ (XQuery API for Java, also known as JSR 225) is a vendor-neutral API for invoking XQuery from Java applications. The Final Release (1.0) is published at http://jcp.org/en/jsr/detail?id=225. Saxon includes a complete and conformant implementation of this API.

For information on how to use the API, please see the JSR 225 documentation.

XQJ has many similarities with JDBC, and its general style is that of a client-server API in which the application opens a "connection" to a database. This of course does not fit the Saxon in-process model particularly well; on the other hand, apart from the terminology and the use of some methods (such as the ability to set a connection timeout) that make little sense in a Saxon context, the API works equally well in an environment like Saxon where the XQuery processor is invoked directly and runs within the same Java VM as the client application.

The samples directory in the issued saxon-resources download file includes a Java test application, , which illustrates some of the possible ways of invoking Saxon using the XQJ interface.

Note that Saxon will generally only recognize its own implementation of XQJ interfaces. For example, the interface `XQDynamicContext` includes a method `bindAtomicValue` that allows the value of a variable or the context item to be supplied. The type of the argument is `XQItem`: however, Saxon will only accept an `XQItem` that was created by its own implementations of the factory methods in `XQDataFactory`.

Unlike JAXP interfaces, XQJ does not include an implementation-independent factory class. Instead, you start the process by calling:

new SaxonXQDataSource()

This constructor will create a new Configuration, which will be an `EnterpriseConfiguration` or `ProfessionalConfiguration` if Saxon-EE or Saxon-PE is in use. As an alternative, there is also a constructor that allows a specific pre-exising configuration to be used.

From the `XQDataSource` you can call `getConnection()` to get a connection, and from the connection you can call `prepareExpression()` to compile a query. The resulting `XQPreparedExpression` object has a method `executeQuery()` allowing the query to be evaluated. The result of the query evaluation is an `XQSequence`, which acts as a cursor or iterator: it has a `next()` method allowing you to change the current position, and a `getItem()` method allowing you to retrieve the item at the current position. The result of `getItem()` is an `XQItem` object, and this has methods allowing you to determine the item type, and to convert the item into a suitable Java object or value.

Using XQuery Update

Saxon-EE supports use of the update extensions to XQuery defined in http://www.w3.org/TR/xquery-update-10/. The current version supported is the Candidate Recommendation.

Update is available only in Saxon-EE, and is supported only if explicitly requested. The command line has an option `update:on` for this purpose, and all the XQuery APIs have an option to enable updating, which must be set before compiling the query. If this option is not set, the parser will not recognize update syntax, and any use of updating expressions will trigger a syntax error.

It is possible for an update to modify the document supplied as the context item, or a document read using the `doc()` or `collection()` function, or even a document constructed dynamically by the query itself. When using the various APIs, the general policy is that updated documents are never written automatically to disk. Instead, the list of updated documents is available to the application on completion of the query, and the application can decide whether to save the documents to their original location, or to some other location. Documents that were originally read from disk will have a document URI property which can be used to decide where to write them back.

When using XQuery Update from the command line, updated documents will be written back to disk if they have a known document URI, and if that URI is an updatable location (which in practice means it must be a URI that uses the `file://` scheme). For testing purposes, the write-back can be suppressed by using `-update:discard`. There is also a `-backup` option to control whether the old file is saved under a different name before being overwritten.

> Saxon does no locking to prevent multiple threads attempting to update the same document. This is entirely a user responsibility.

Most errors that can arise during updating operations (for example, inserting two conflicting attributes) will cause an exception, with the supplied input document in memory being left in its original state. However, errors detected during the validation phase (that is, when the updated document is invalid against the schema, assuming revalidation is requested) are non-recoverable; after such a failure, the state of the document is unpredictable. Generally the (invalid) updates will have been made, and some of the updates done during schema validation (setting type annotations and default values) may also have been made.

Note that updates to a document will fail unless it is implemented using the model. This can be selected from the command line using `-tree:linked`, or via configuration settings in the API. It is not at present possible to update the Tiny Tree, nor external object models such as DOM, JDOM, or XOM.

Calling XQuery Functions from Java

Although the usual way to invoke XQuery from a Java application is to compile and execute a query as described above, it is also possible to invoke individual XQuery functions directly from Java if required. This interface is very efficient but performs less validation of parameters than the standard interface described above. To achieve this, first compile the query as described above, specifying a query that includes one or more `declare function` declarations in the query prolog. It is then possible to retrieve the `UserFunction` objects representing the compiled code of these functions, by calling the `getUserDefinedFunction` method on the `StaticQueryContext` object. As discussed above, the information is not held in the original `StaticQueryContext` object (which Saxon does not alter), but in a modified copy which is obtainable from the `XQueryExpression` object. Once found, such a function can be called using its `call` method. The first argument to this is an array containing the values of the arguments. These must be supplied using Saxon's native classes, for example an integer argument is supplied as an instance of `IntegerValue [Javadoc: net.sf.saxon.value.IntegerValue]`. These values must be of the type expected by the function; no conversion or type checking takes place (which means that a `ClassCastException` will probably occur if the wrong type of value is supplied). The second argument to the `call` method is a `Controller`. A Controller can be obtained by calling the `getController()` method on the `XQueryExpression` object. The same Controller can be used for a series of separate function calls.

For example:

```
Configuration config = new Configuration();
StaticQueryContext sqc = config.newStaticQueryContext();

XQueryExpression exp1 = sqc.compileQuery(
        "declare namespace f='f.ns';" +
        "declare function f:t1($p as xs:integer) { $p * $p };" +
        "declare function f:t2($p as xs:integer) { $p + $p };" +
        "1"
);

StaticQueryContext sqc2 = exp1.getStaticContext();
UserFunction fn1 = sqc2.getUserDefinedFunction("f.ns", "t1", 1);
UserFunction fn2 = sqc2.getUserDefinedFunction("f.ns", "t2", 1);
Controller controller = exp1.getController();

IntegerValue[] arglist = new IntegerValue[1];
for (int x=1; x<1000000; x++) {
    arglist[0] = new IntegerValue(x);
    Value v1 = fn1.call(arglist, controller);
    Value v2 = fn2.call(arglist, controller);
    System.err.println("Returned product " + v1 + "; sum =" + v2);
}
```

Result Format

The result of a query is a sequence of nodes and atomic values - which means it is not, in general, an XML document. This raises the question as to how the results should be output.

The Saxon command line processor for XQuery by default produces the output in raw format. This converts the result sequence to a document following the rules of the XQuery `document{}` constructor, and then serializes this document.

The alternative is wrapped format, requested using the `-wrap` argument. This wraps the result sequence as an XML document, and then serializes the resulting document. Each item in the result sequence is wrapped in an element (such as `result:element` or `result:atomic-value`) according to its type. The sequence as a whole is wrapped in a `result:sequence` element.

Compiling Queries

In Saxon 9.4, Saxon-EE automatically (and selectively) compiles queries to Java bytecode. When running on .NET, the bytecode is then automatically converted to IL code for execution. The bytecode exists only in memory, and would not be useful otherwise because it contains many references to the data structures generated by the Saxon parser and optimizer.

This facility replaces the ability in previous Saxon-EE releases to generate Java source code from a query.

The performance boost achieved by bytecode generation is variable; 25% is typical. The constructs that benefit the most are those where the expression tree contains many constructs that are relatively cheap in themselves, such as type conversion, comparisons, and arithmetic. This is because the saving from bytecode generation is mainly not in the cost of performing primitive operations, but in the cost of deciding which operations to perform: so the saving is greater where the number of operations is high relative to their average cost.

There are configuration options to suppress bytecode generation (`FeatureKeys.GENERATE_BYTE_CODE`), to insert debugging logic into the generated bytecode (`FeatureKeys.DEBUG_BYTE_CODE`), and to display the generated bytecode (`FeatureKeys.DISPLAY_BYTE_CODE`).

Extensibility

The Saxon XQuery implementation allows you to call Java methods as external functions. The function does not need to be declared. Use a namespace declaration such as `declare namespace math="java:java.lang.Math"`, and invoke the method as `math:sqrt(2)`.

More details of this mechanism are found in Writing Extension Functions.

Saxon recognizes the XQuery pragma syntax, but it currently defines only one pragma of its own, the `saxon:validate-type` pragma (see Extensions), and this is now redundant since the equivalent facility is standard in XQuery 3.0. Saxon will adopt the correct fallback behavior if presented with a query that uses another vendor's extensions, provided these are designed in conformance with the W3C pragma specification.

Saxon also recognizes the XQuery option declaration syntax. Several specific option declarations are provided: `declare option saxon:default` declares a default value for external variables (query parameters); `declare option saxon:output` declares a serialization parameter; and `declare option saxon:memo-function` defines whether the following function declaration is to be implemented as a memo function. These are described under Extensions. Any other option declaration in the Saxon namespace is ignored with a warning; an option declaration in any other namespace is ignored silently.

Extensions

The full library of Saxon and EXSLT functions described in Extensions is available, except for those (for example, some forms of `saxon:serialize`) that have an intrinsic dependency on an XSLT stylesheet.

declare option saxon:default

An XQuery option declaration is defined allowing a default value to be specified for a query parameter (external variable). The syntax is illustrated below:

```
declare namespace saxon="http://saxon.sf.net/";
declare option saxon:default "20";
declare variable $x external;
```

The default value is written as an XPath expression. The surrounding quotes are part of the "declare option" syntax, not part of the expression: therefore, if the default value is to be supplied as a string literal, two sets of quotes are needed. In the above example, the default value is the integer 20, not a string. Perhaps it would be clearer to show this by writing `saxon:default "(+20)"`

declare option saxon:output

Saxon provides an option declaration to set serialization parameters. This takes the form shown in the following example:

```
declare namespace saxon="http://saxon.sf.net/";
declare option saxon:output "method=html";
declare option saxon:output "saxon:indent-spaces=1";
```

The standard serialization parameters described in The W3C Serialization specification [http://www.w3.org/TR/xslt-xquery-serialization/] are all available, namely:

- byte-order-mark
- cdata-section-elements
- doctype-public
- doctype-system
- encoding
- escape-uri-attributes
- include-content-type
- indent
- media-type
- method
- normalization-form
- omit-xml-declaration
- standalone
- undeclare-prefixes
- use-character-maps (only useful in XSLT)
- version

In addition some Saxon-specific serialization parameters are available: see Additional serialization parameters.

declare option saxon:memo-function

Saxon provides an option declaration to treat the immediately following function as a memo-function. This takes the form shown in the following example:

```
declare namespace saxon="http://saxon.sf.net/";
declare option saxon:memo-function "true";
declare function local:factorial($n as xs:integer) as xs:integer {
  local:factorial($n - 1) * n
};
```

A memo function remembers the results of previous calls, so if it is called twice with the same arguments, it will not repeat the computation, but return the previous result.

The allowed values of the option are "true" and "false" (the default is false), and any other value is ignored with a warning.

declare option saxon:allow-cycles

Saxon checks for the error XQST0093 which was introduced in the Proposed Recommendation. This error makes it illegal for a function or variable in module A to reference a function or variable in module B if there is a function or variable in module B that references one in A. Because this restriction is quite unnecessary and makes it very difficult to write modular applications, Saxon provides an option `declare option saxon:allow-cycles "true"` to disable this check. This option also disables error XQST0073, which otherwise occurs when two modules in different namespaces import each other

The allowed values of the option are "true" and "false" (the default is false), and any other value is ignored with a warning.

This option does not disable the check for cycles that would actually cause execution to fail, for example a global variable $V1 whose initializer uses $V2, when $V2 similarly depends on $V1.

Note that this option declaration must be written the module imports, but before any variable or function declarations.

The saxon:validate-type pragma

Saxon-EE provides a pragma (a language extension) to allow constructed elements to be validated against a schema-defined global type definition. The standard `validate` expression allows validation only against a global element declaration, but some schemas (an example is FpML) provide very few global elements, and instead rely heavily on locally-declared elements having a global type. This makes it impossible to construct fragments of an FpML document in a way that takes advantage of static and dynamic type checking.

The extension takes the form:

```
(# saxon:validate-type my:personType #) { expr }
```

Conceptually, it makes a copy of the result of evaluating `expr` and validates it against the named schema type, causing the copied nodes to acquire type annotations based on the validation process. The effect is the same as that of the `type` attribute in XSLT instructions such as `xsl:element` and `xsl:copy-of`. The schema type (shown in the above example as `myPersonType`) may be a simple type or a complex type defined in an imported schema, or a built-in type; it is written as a QName, using the default namespace for elements and types if it is unprefixed.

Note that XQuery processors other than Saxon will typically ignore this pragma, and return the value of `expr` unchanged. Such processors will report type-checking failures if the value is used in a context where the required type is `element(*, type-name)`.

You can use a different namespace prefix in place of "saxon", but it must be bound using a namespace declaration to the namespace "http://saxon.sf.net/".

Here is a complete example:

```
module namespace tim="http://www.example.com/module/hour-minute-time";
declare namespace saxon="http://saxon.sf.net/";
import schema namespace fpml = "http://www.fpml.org/2005/FpML-4-2" at "....";

declare function time:getCurrentHourMinuteTime() as element(*, fpml:HourMinuteT
    let $time = string(current-time())
    return
        (# saxon:validate-type fpml:HourMinuteTime #) {
            <time>{substring($time, 1, 5)}:00</time>
        }
};
```

Saxon ignores any pragmas in a namespace other than the Saxon namespace; it rejects any pragmas whose QName is ill-formed, as well as pragmas in the Saxon namespace whose local name is not recognized.

The construct also allows validation of attributes against a simple type definition, for example:

```
module namespace tim="http://www.example.com/module/hour-minute-time";
declare namespace saxon="http://saxon.sf.net/";
import schema namespace fpml = "http://www.fpml.org/2005/FpML-4-2" at "....";

declare function time:getCurrentHourMinuteTime() as attribute(*, fpml:HourMinut
    let $time = string(current-time())
    return (# saxon:validate-type fpml:HourMinuteTime #) {
       attribute time {concat(substring($time, 1, 5), ':00')}
    }
};
```

Use Cases

Saxon runs all the XQuery Use Cases [http://www.w3.org/xquery-use-cases].

The relevant queries (some of which have been corrected from those published by W3C) are included in the Saxon distribution (folder `use-cases`) together with batch scripts for running them. A few additional use cases have been added to show features that would otherwise not be exercised. A separate script is available for running the STRONG use cases since these require Saxon-EE.

Also included in the distribution is a query `samples/query/tour.xq`. This is a query that generates a knight's tour of the chessboard. It is written as a demonstration of recursive functional programming in XQuery. It requires no input document. You need to supply a parameter on the command line indicating the square where the knight should start, for example `start=h8`. The output is an HTML document.

Chapter 7. Handling Source Documents

Handling Source Documents

This section discusses the various options in Saxon for handling source documents that form the input to a query or stylesheet.

See the topics below for further information:

- Source Documents on the Command Line
- Collections
- Building a Source Document from an application
- Preloading shared reference documents
- Using XML Catalogs
- Writing input filters
- XInclude processing
- Controlling Parsing of Source Documents
- Saxon and XML 1.1
- JAXP Source Types
- Third-party Object Models: DOM, JDOM, XOM, and DOM4J
- Choosing a Tree Model
- The PTree File Format
- Validation of Source Documents
- Whitespace Stripping in Source Documents
- Streaming of Large Documents
- Document Projection
- References to W3C DTDs

Source Documents on the Command Line

When Saxon (either XSLT or XQuery) is invoked from the command line, the source document will normally be an XML 1.0 document. Supplying an XML 1.1 document will also work, provided that (a) the selected parser is an XML 1.1 parser, and (b) the command line option `-1.1` is set.

If a custom parser is specified using the `-x` option on the command line, then the source document can be in any format accepted by this custom parser. The only constraint is that the parser must behave as a SAX2 parser, delivering a stream of events that define a virtual XML document. For example, the TagSoup [http://www.tagsoup.info/] parser from John Cowan can be used to feed an HTML document as input to Saxon.

Non-standard input formats can also be handled by specifying a user-written `URIResolver`. If the `-u` option is used on the command line, or if the source file name begins with `http://` or `file://`, then the source file name is resolved to a JAXP Source object using the `URIResolver`; if a user-written `URIResolver` is nominated (using the `-r` option) then this may translate the file name into a Source object any way that it wishes.

Collections

Saxon implements the `collection()` function by passing the given URI (or null, if the default collection is requested) to a user-provided `CollectionURIResolver [Javadoc: net.sf.saxon.lib.CollectionURIResolver]`. This section describes how the standard collection resolver behaves, if no user-written collection resolver is supplied.

The default collection resolver returns the empty sequence as the default collection. The only way of specifying a default collection it to provide your own `CollectionURIResolver`.

If a collection URI is provided, Saxon attempts to dereference it. What happens next depends on whether the URI identifies a file or a directory.

Using catalog files

If the collection URI identifies a file, Saxon treats this as a catalog file. This is a file in XML format that lists the documents comprising the collection. Here is an example of such a catalog file:

```
<collection stable="true">
  <doc href="dir/chap1.xml"/>
  <doc href="dir/chap2.xml"/>
  <doc href="dir/chap3.xml"/>
  <doc href="dir/chap4.xml"/>
</collection>
```

The `stable` attribute indicates whether the collection is stable or not. The default value is `true`. If a collection is stable, then the URIs listed in the `doc` elements are treated like URIs passed to the `doc()` function. Each URI is first looked up in the document pool to see if it is already loaded; if it is, then the document node is returned. Otherwise the URI is passed to the registered `URIResolver`, and the resulting document is added to the document pool. The effect of this process is firstly, that two calls on the `collection()` function passing the same collection URI will return the same nodes each time, and secondly, that these results are consistent with the results of the `doc()` function: if the `document-uri()` of a node returned by the `collection()` function is passed to the `doc()` function, the original node will be returned. If `stable="false"` is specified, however, the URI is dereferenced directly, and the document is not added to the document pool, which means that a subsequent retrieval of the same document will not return the same node.

Processing directories

If the URI passed to the `collection()` function (still assuming a default `CollectionURIResolver`) identifies a directory, then the contents of the directory are returned. Such a URI may have a number of query parameters, written in the form `file:///a/b/c/d?keyword=value;keyword=value;....` The recognized keywords and their values are as follows:

Table 7.1.

The pattern used in the `select` parameter can take the conventional form, for example `*.xml` selects all files with extension "xml". More generally, the pattern is converted to a regular expression by prepending "^", appending "$", replacing "." by "\.", and replacing "*" by ".*", and it is then used to match the file names appearing in the directory using the Java regular expression rules. So, for example, you can write `?select=*.(xml|xhtml)` to match files with either of these two file extensions. Note however, that special characters used in the URL (that is, characters with a special meaning in regular expressions) may need to be escaped using the %HH convention. For example, vertical bar needs to be written as `%7C`. This escaping can be achieved using the iri-to-uri() function.

Registered Collections

On the .NET product there is a third way to use a collection URI (provided that you use the API rather than the command line): you can register a collection using the `Processor.RegisterCollection` method on the `Saxon.Api.Processor` class.

Building a Source Document from an application

With the Java s9api interface, a source document can be built using the `DocumentBuilder [Javadoc: net.sf.saxon.s9api.DocumentBuilder]` class, which is created using the factory method `newDocumentBuilder` on the `Processor [Javadoc: net.sf.saxon.s9api.Processor]` object. Various options for document building are available as methods on the `DocumentBuilder`, for example options to perform schema or DTD validation, to strip whitespace, to expand XInclude directives, and also to choose the tree implementation model to be used.

Similarly in the .NET API, there is a `DocumentBuilder` object that can be created from the processor. This allows options to be set controlling the way documents are built, and provides an overloaded `Build` method allowing a tree to be built from various kinds of source.

It is also possible to build a Saxon tree in memory by using the `buildDocument` method of the `Configuration [Javadoc: net.sf.saxon.Configuration]` object. (When using the JAXP Transformation API, the `Configuration` can be obtained from the `TransformerFactory` as the value of the attribute named `FeatureKeys.CONFIGURATION [Javadoc: net.sf.saxon.lib.FeatureKeys#CONFIGURATION]`.)

The `buildDocument() [Javadoc: net.sf.saxon.Configuration#buildDocument]` method takes a single argument, a JAXP `Source`. This can be any of the standard kinds of JAXP `Source`. See JAXP Sources for more information.

All the documents processed in a single transformation or query must be loaded using the same `Configuration [Javadoc: net.sf.saxon.Configuration]`. However, it is possible to copy a document from one `Configuration` into another by supplying the `DocumentInfo [Javadoc: net.sf.saxon.om.DocumentInfo]` at the root of the existing document as the `Source` supplied to the `buildDocument()` method of the new Configuration.

Preloading shared reference documents

An option is available in the `Configuration [Javadoc: net.sf.saxon.Configuration]` to indicate that calls to the `doc()` or `document()` functions with constant string arguments should be evaluated when a query or stylesheet is compiled, rather than at run-time. This option is intended for use when a reference or lookup document is used by all queries and transformations. Using this option has a number of effects:

1. The URI is resolved using the compile-time URIResolver rather than the run-time URIResolver

2. The document is loaded into a document pool held by the `Configuration`, whose memory is released only when the `Configuration` itself ceases to exist;

3. all queries and transformations using this document share the same copy;

4. any updates to the document that occur between compile-time and run-time have no effect.

The option is selected by using `Configuration.setConfigurationProperty()`\ or `TransformerFactory.setAttribute()` with the property name `FeatureKeys.PRE_EVALUATE_DOC_FUNCTION [Javadoc: net.sf.saxon.lib.FeatureKeys#PRE_EVALUATE_DOC_FUNCTION]`. This option is not available from the command line because it has no useful effect with a single-shot compile-and-run interface.

This option has no effect if the URI supplied to the `doc()` or `document()` function includes a fragment identifier.

It is also possible to preload a specific document into the shared document pool from the Java application by using the call `config.getGlobalDocumentPool().add(doc, uri)`. When the `doc()` or `document()` function is called, the shared document pool is first checked to see if the requested document is already present. The `DocumentPool [Javadoc: net.sf.saxon.om.DocumentPool]` object also has a `discard()` method which causes the document to be released from the pool.

Using XML Catalogs

XML Catalogs (defined by OASIS [http://xml.apache.org/commons/components/resolver/resolver-article.html]) provide a way to avoid hard-coding the locations of XML documents and other resources in your application. Instead, the applicaton refers to the resource using a conventional system identifier (URI) or public identifier, and a local catalog is used to map the system and public identifiers to an actual location.

When using Saxon from the command line, it is possible to specify a catalog to be used using the option `-catalog:`. Here is the catalog file to be searched, or a list of filenames separated by semicolons. This catalog will be used to locate DTDs and external entities required by the XML parser, XSLT stylesheet modules requested using `xsl:import` and `xsl:include`, documents requested using the `document()` and `doc()` functions, and also schema documents, however they are referenced.

With Saxon on the Java platform, if the `-catalog` option is used on the command line, then the open-source Apache library `resolver.jar` must be present on the classpath. With Saxon on .NET, this module (cross-compiled to IL) is included within the Saxon DLL.

Setting the `-catalog` option is equivalent to setting the following options:

Table 7.2.

-r	org.apache.xml.resolver.tools.CatalogResolver
-x	org.apache.xml.resolver.tools.ResolvingXMLReader
-y	org.apache.xml.resolver.tools.ResolvingXMLReader

In addition, the system property `xml.catalog.files` is set to the value of the supplied value. And if the `-t` option is also set, Saxon sets the verbosity level of the catalog manager to 2, causing it to report messages for each resolved URI. Saxon customizes the Apache resolver library to integrate these messages with the other output from the `-t` option: that is, by default it is sent to the standard error output.

When the `-catalog` option is used on the command line, this overrides the internal resolver used in Saxon (from 9.4) to redirect well-known W3C references (such as the XHTML DTD) to

Saxon's local copies of these resources. Because both these features rely on setting the XML parser's `EntityResolver`, it is not possible to use them in conjunction.

This support for OASIS catalogs is implemented only in the Saxon command line. To use catalogs from a Saxon application, it is necessary to configure the various options individually. For example:

- To use catalogs to resolve references to DTDs and external entities, choose `ResolvingXMLReader` as your XML parser, or set `org.apache.xml.resolver.tools.CatalogResolver` as the `EntityResolver` used by your chosen XML parser.
- To use catalogs to resolve `xsl:include` and `xsl:import` references, choose `org.apache.xml.resolver.tools.CatalogResolver` as the `URIResolver` used by Saxon when compiling the stylesheet.
- To use catalogs to resolve calls on `doc()` or `document()` references, choose `org.apache.xml.resolver.tools.CatalogResolver` as the `URIResolver` used by Saxon when running the stylesheet (for example, using `Transformer.setURIResolver()`).

Here is an example of a very simple catalog file. The `publicId` and `systemID` attributes give the public or system identifier as used in the source document; the `uri` attribute gives the location (in this case a relative location) where the actual resource will be found.

```
 <?xml version="1.0"?>
<catalog  xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
  <group  prefer="public"  xml:base="file:///usr/share/xml/" >

    <public
       publicId="-//OASIS//DTD DocBook XML V4.5//EN"
       uri="docbook45/docbookx.dtd"/>

    <system
       systemId="http://www.oasis-open.org/docbook/xml/4.5/docbookx.dtd"
       uri="docbook45/docbookx.dtd"/>

  </group>
</catalog>
```

There are many tutorials for XML catalogs available on the web, including some that have information specific to Saxon, though this may well relate to earlier releases.

Writing input filters

Saxon can take its input from a JAXP `SAXSource` object, which essentially represents a sequence of SAX events representing the output of an XML parser. A very useful technique is to interpose a between the parser and Saxon. The filter will typically be an instance of the SAX2 class.

There are a number of ways of using a Saxon XSLT transformation as part of a pipeline of filters. Some of these techniques also work with XQuery. The techniques include:

- Generate the transformation as an `XMLFilter` using the `newXMLFilter()` method of the `TransformerFactory`. This works with XSLT only. A drawback of this approach is that it is not possible to supply parameters to the transformation using standard JAXP facilities. It is possible, however, by casting the `XMLFilter` to a `net.sf.saxon.Filter [Javadoc: net.sf.saxon.Filter]`, and calling its `getTransformer()` method, which returns a `Transformer` object offering the usual `addParameter()` method.

- Generate the transformation as a SAX `ContentHandler` using the `newTransformerHandler()` method. The pipelines stages after the transformation can be added by giving the transformation a `SAXResult` as its destination. This again is XSLT only.
- Implement the pipeline step before the transformation or query as an `XMLFilter`, and use this as the `XMLReader` part of a `SAXSource`, pretending to be an XML parser. This technique works with both XSLT and XQuery, and it can even be used from the command line, by nominating the `XMLFilter` as the source parser using the `-x` option on the command line.

The `-x` option on the Saxon command line specifies the parser that Saxon will use to process the source files. This class must implement the SAX2 XMLReader interface, but it is not required to be a real XML parser; it can take the input from any kind of source file, so long as it presents it in the form of a stream of SAX events. When using the JAXP API, the equivalent to the `-x` option is to call `transformerFactory.setAttribute(net.sf.saxon.lib.FeatureKeys.SOURCE_PARSER_CLASS, 'com.example.package.Parser')`

XInclude processing

If you are using Xerces as your XML parser, you can have Xerces expand any XInclude directives.

The `-xi` option on the command line causes XInclude processing to be applied to all input XML documents. This includes source documents, stylesheets, and schema documents listed on the command line, and also those loaded indirectly for example by calls on the `doc()` function or by mechanisms such as `xsl:include` and `xs:include`.

From the Java API, the equivalent is to call `setXInclude()` on the `Configuration` object, or to set the attribute denoted by `FeatureKeys.XINCLUDE [Javadoc: net.sf.saxon.lib.FeatureKeys#XINCLUDE]` to `Boolean.TRUE` on the `TransformerFactory`.

XInclude processing can be requested at a per-document level by creating an `AugmentedSource [Javadoc: net.sf.saxon.lib.AugmentedSource]` and calling its `setXIncludeAware()` method. The corresponding method is also recognized on Saxon's implementation of the JAXP `DocumentBuilderFactory`. When the `doc()` or `document()` or `collection()` function is called from an XPath expression, XInclude processing can be enabled by including `xinclude=yes` among the query parameters in the URI.

It is also possible to switch on XInclude processing (for all documents) by setting the system property:

```
-Dorg.apache.xerces.xni.parser.XMLParserConfiguration=
    org.apache.xerces.parsers.XIncludeParserConfiguration
```

An alternative approach is to incorporate an XInclude processor as a SAX filter in the input pipeline. You can find a suitable SAX filter at http://xincluder.sourceforge.net/, and you can incorporate it into your application as described at Writing Input Filters.

On the .NET platform, there is a customized `XmlReader` that performs XInclude processing available at http://www.xmlmvp.org/xinclude/index.html. You can supply this as an argument to the method `Build(XmlReader parser)` in the `DocumentBuilder` class of the .NET Saxon API.

For further information on using XInclude, see http://www.sagehill.net/docbookxsl/Xinclude.html

Controlling Parsing of Source Documents

Saxon does not include its own XML parser. By default:

- On the Java platform, the default SAX parser provided as part of the JDK is used. With the Sun/ Oracle JDK, this is a variant of the Apache Xerces parser customized by Sun.
- On the .NET platform, Saxon includes a copy of the Apache Xerces parser cross-compiled to run on .NET

An error reported by the XML parser is generally fatal. It is not possible to process ill-formed XML.

There are several ways you can cause a different XML parser to be used:

- The -x and -y options on the command line can be used to specify the class name of a SAX parser, which Saxon will load in preference to the default SAX parser. The -x option is used for source XML documents, the -y option for schemas and stylesheets. The equivalent options can be set programmatically or by using the configuration file.
- By default Saxon uses the `SAXParserFactory` mechanism to load a parser. This can be configured by setting the system property `javax.xml.parsers.SAXParserFactory`, by means of the file `lib/jaxp.properties` in the JRE directory, or by adding another parser to the `lib/endorsed` directory.
- The source for parsing can be supplied in the form of a `SAXSource` object, which has an `XMLReader` property containing the parser instance to be used.
- On .NET, the configuration option `PREFER_JAXP_PARSER` can be set to false, in which case Saxon will use the Microsoft XML parser instead of the Apache parser. (This parser is not used by default because it does not notify ID attributes to the application, which means the XPath `id()` and `idref()` functions do not work.)

Saxonica recommends use of the Xerces parser from Apache in preference to the version bundled in the JDK, which is known to have some serious bugs.

By default, Saxon invokes the parser in non-validating mode (that is, without requested DTD validation). Note however, that the parser still needs to read the DTD if one is present, because it may contain entity definitions that need to be expanded. DTD validation can be requested using `-dtd:on` on the command line, or equivalent API or configuration options.

Saxon is issued with local copies of commonly-used W3C DTDs such as the XHTML, SVG, and MathML DTDs. When Saxon itself instantiates the XML parser, it will use an EntityResolver that causes these local copies of DTDs to be used rather than fetching public copies from the web (the W3C servers are increasingly failing to serve these requests as the volume of traffic is too high.) It is possible to override this using the configuration setting `ENTITY_RESOLVER_CLASS`, which can be set to the name of a user-supplied EntityResolver, or to the empty string to indicate that no EntityResolver should be used. Saxon will not add this EntityResolver in cases where the XML parser instance is supplied by the caller as part of a `SAXSource` object. It will add it to a parser obtained as an instance of the class specified using the -x and -y command line options, unless either the use of the EntityResolver is suppressed using the `ENTITY_RESOLVER_CLASS` configuration option, or the instantiated parser already has an EntityResolver registered.

Saxon never asks the XML parser to perform schema validation. If schema validation is required it should be requested using the command line options `-val:strict` or `-val:lax`, or their API equivalents. Saxon will then use its own schema processor to validate the document as it emerges from the XML parser. Schema processing is done in parallel with parsing, by use of a SAX-like pipeline.

Saxon and XML 1.1

XML 1.1 (with XML Namespaces 1.1) originally extended XML 1.0 in three ways:

- The set of valid characters is increased
- The set of characters allowed in XML Names is increased

- Namespace undeclarations are permitted.

The second change has subsequently been retrofitted to XML 1.0 Fifth Edition (XML 1.0e5). Saxon now uses the XML 1.1 and XML 1.0e5 rules unconditionally for all validation of XML names.

Saxon is capable of working with XML 1.1 input documents. If you want to use Saxon with XML 1.1, you should set the option "-xmlversion:1.1" on the Saxon command line, or call the method `configuration.setXMLVersion(Configuration.XML11)` `[Javadoc: net.sf.saxon.Configuration#setXMLVersion]` or, in the case of XSLT, `transformerFactory.setAttribute(FeaturesKeys.XML_VERSION, "1.1")`

This configuration setting affects:

- the characters considered valid in the source of an XQuery query
- the characters considered valid in the result of the functions `codepoints-to-string()` and `unparsed-text()`
- the characters considered valid in the result of certain Saxon extension functions
- the way in which line endings in XQuery queries are normalized
- the default version used by the serializer (with output method XML)

Since Saxon 9.4, the configuration setting no longer affects:

- validation of names used in XQuery and XPath expressions, including names of elements, attributes, functions, variables, and types
- validation of names of constructed elements, attributes, and processing instructions in XQuery and XSLT
- schema validation of values of type `xs:NCName`, `xs:QName`, `xs:NOTATION`, and `xs:ID`
- the permitted names of stylesheet objects such as keys, templates, decimal-formats, output declarations, and output methods

Note that if you use the default setting of "1.0", then supplying an XML 1.1 source document as input may cause undefined errors.

It is advisable to use an XML parser that supports XML 1.1 when the configuration is set to "1.1", and an XML parser that does not support XML 1.1 when the configuration is set to "1.0". However, Saxon does not enforce this.

You can set the configuration to allow XML 1.1, but still serialize result documents as XML 1.0 by specifying the output property `version="1.0"`. In this case Saxon will check while serializing the document that it conforms to the XML 1.0 constraints (note that this check can be expensive). These checks are not performed if the configuration default is set to XML 1.0.

If you want the serializer to output namespace undeclarations, use the output property `undeclare-namespaces="yes"` as well as `version="1.1"`.

JAXP Source Types

When a user application invokes Saxon via the Java API, then a source document is supplied as an instance of the JAXP `Source` class. This is true whether invoking an XSLT transformation, an XQuery query, or a free-standing XPath expression. The `Source` class is essentially a marker interface. The `Source` that is supplied must be a kind of `Source` that Saxon recognizes.

Saxon recognizes the three kinds of `Source` defined in JAXP: a `StreamSource`, a `SAXSource`, and a `DOMSource`.

Note that the Xerces DOM implementation is not thread-safe, even for read-only access. Never use a `DOMSource` in several threads concurrently, unless you have checked that the DOM implementation you are using is thread-safe.

Saxon also accepts input from an `XMLStreamReader` (`javax.xml.stream.XMLStreamReader`), that is a StAX pull parser as defined in JSR 173. This is achieved by creating an instance of `net.sf.saxon.pull.StaxBridge` `[Javadoc: net.sf.saxon.pull.StaxBridge]`, supplying the `XMLStreamReader` using the `setXMLStreamReader()` method, and wrapping the `StaxBridge` object in an instance of `net.sf.saxon.pull.PullSource` `[Javadoc: net.sf.saxon.pull.PullSource]`, which implements the JAXP `Source` interface and can be used in any Saxon method that expects a `Source`. Saxon has been validated with two StAX parsers: the Zephyr parser from Sun (which is supplied as standard with JDK 1.6), and the open-source Woodstox parser from Tatu Saloranta. In my experience, Woodstox is the more reliable of the two. However, there is no immediate benefit in using a pull parser to supply Saxon input rather than a push parser; the main use case for using an `XMLStreamReader` is when the data is supplied from some source other than parsing of lexical XML.

Nodes in Saxon's implementation of the XPath data model are represented by the interface `NodeInfo` `[Javadoc: net.sf.saxon.om.NodeInfo]`. A `NodeInfo` is itself a `Source`, which means that any method in the API that requires a source object will accept any implementation of `NodeInfo`. As discussed in the next section, implementations of `NodeInfo` are available to wrap DOM, DOM4J, JDOM, or XOM nodes, and in all cases these wrapper objects can be used wherever a `Source` is required.

Saxon also provides a class `net.sf.saxon.lib.AugmentedSource` `[Javadoc: net.sf.saxon.lib.AugmentedSource]` which implements the `Source` interface. This class encapsulates one of the standard `Source` objects, and allows additional processing options to be specified. These options include whitespace handling, schema and DTD validation, XInclude processing, error handling, choice of XML parser, and choice of Saxon tree model.

Saxon allows additional `Source` types to be supported by registering a `SourceResolver` `[Javadoc: net.sf.saxon.lib.SourceResolver]` with the `Configuration` `[Javadoc: net.sf.saxon.Configuration]` object. The task of a `SourceResolver` is to convert a `Source` that Saxon does not recognize into a `Source` that it does recognize. For example, this may be done by building the document tree in memory and returning the `NodeInfo` `[Javadoc: net.sf.saxon.om.NodeInfo]` object representing the root of the tree.

Third-party Object Models: DOM, JDOM, XOM, and DOM4J

In the case of DOM, all Saxon editions support DOM access "out of the box", and no special configuration action is necessary.

Support for JDOM, XOM, and DOM4J is not available "out of the box" with Saxon-HE, but the source code is open source (in sub-packages of `net.sf.saxon.option`) and can be compiled for use with Saxon-HE if required.

In general, use of a third party tree implementation is much less efficient than using Saxon's native `TinyTree`. These models should only be used if your application needs to construct them for other reasons. Transforming a DOM can take up to 10 times longer than transforming the equivalent `TinyTree`.

A support module for JDOM2 has been created, but is not released with Saxon 9.4 because JDOM2 at the time of release was not yet sufficiently stable.

In addition, Saxon allows various third-party object models to be used to supply the input to a transformation or query. Specifically, it supports JDOM, XOM, and DOM4J in addition to DOM. Since Saxon 9.2 the support code for these three models is integrated into the main JAR files for Saxon-PE and Saxon-EE, but (unlike the case of DOM) it is not activated unless the object model is registered with the `Configuration [Javadoc: net.sf.saxon.Configuration]`, which can be done either by including it in the relevant section of the configuration file, or by nominating it using the method `registerExternalObjectModel() [Javadoc: net.sf.saxon.Configuration#registerExternalObjectModel]`.

For DOM input, the source can be supplied by wrapping a `DOMSource` around the DOM Document node. For JDOM, XOM, and DOM4J the approach is similar, except that the wrapper classes are supplied by Saxon itself: they are `net.sf.saxon.option.jdom.DocumentWrapper [Javadoc: net.sf.saxon.option.jdom.DocumentWrapper]`, `net.sf.saxon.option.xom.DocumentWrapper [Javadoc: net.sf.saxon.option.xom.DocumentWrapper]`, and `net.sf.saxon.option.dom4j.DocumentWrapper [Javadoc: net.sf.saxon.option.dom4j.DocumentWrapper]`, and respectively. These wrapper classes implement the Saxon `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` interface (which means that they also implement `Source`).

> Note that the Xerces DOM implementation is not thread-safe, even for read-only access. Never use a `DOMSource` in several threads concurrently, unless you have checked that the DOM implementation you are using is thread-safe.

Saxon supports these models by wrapping each DOM, JDOM, XOM, or DOM4J node in a wrapper that implements the Saxon `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` interface. When nodes are returned by the XQuery or XPath API, these wrappers are removed and the original node is returned. Similarly, the wrappers are generally removed when extension functions expecting a node are called.

In the case of DOM only, Saxon also supports a wrapping the other way around: an object implementing the DOM interface may be wrapped around a Saxon `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]`. This is done when Java methods expecting a DOM Node are called as extension functions, if the `NodeInfo` is not itself a wrapper for a DOM `Node`.

You can also send output to a DOM by using a `DOMResult`, or to a JDOM tree by using a `JDOMResult`, or to a XOM document by using a `XOMWriter`. In such cases it is a good idea to set `saxon:require-well-formed="yes"` on `xsl:output` to ensure that the transformation or query result is a well-formed document (for example, that it does not contain several elements at the top level).

Choosing a Tree Model

Saxon provides several implementations of the internal tree data structure (or tree model). The tree model can be chosen by an option on the command line (-tree:tiny for the tiny tree, -tree:linked for the linked tree [previously known as the "standard tree"]). There is also a variant of the tiny tree called a "condensed tiny tree" which saves space (at the expense of build time) by recognizing text nodes and attribute nodes whose values appear more than once in the input document. The tree model can also be selected from the Java API. The default is to use the tiny tree model. The choice should make no difference to the results of a transformation (except the order of attributes and namespace declarations) but only affects performance.

Generally speaking, the tiny tree model is both faster to build and faster to navigate. It also uses less space.

The tiny tree model gives most benefit when you are processing a large document. It uses a lot less memory, so it can prevent thrashing when the size of document is such that the linked tree doesn't fit in real memory. Use the "condensed" variant if you need to save memory, and if your source data contains many text or attribute nodes with repeated values.

The linked tree is used internally to represent stylesheet and schema modules because of the programming convenience it offers: it allows element nodes on the tree to be represented by custom classes for each kind of element. The linked tree is also needed when you want to use XQuery Update, because unlike the TinyTree, it is mutable.

The PTree File Format

Saxon-PE and Saxon-EE support a file format called the PTree (persistent tree). This is a binary representation of an XML document. The PTree file is generally about the same size as the original document (perhaps 10% smaller), but it typically loads in about half the time. Storing a document as a PTree can therefore give a useful performance improvement when the same source document is used repeatedly as the input to many queries or transformations. Another benefit of the PTree is that it retains any type information that is present, which means that the document does not need to be validated against its schema each time it is loaded. (The schema, however, must be loaded whenever the document is loaded.)

Two commands are available for converting XML documents into PTree files and vice versa. To create a PTree, use:

java com.saxonica.ptree.PTreeWriter source.xml result.ptree

The option `-strip` causes all whitespace-only text nodes to be stripped in the process, which will often give a useful saving in space and therefore in loading time.

To convert a PTree back to an XML document, use:

java com.saxonica.ptree.PTreeReader source.ptree result.xml

It is possible to apply a query or transformation directly to a PTree by specifying the `-p` option on the command line for `com.saxonica.Transform` or `com.saxonica.Query`. This option actually causes a different URIResolver, the `PTreeURIResolver [Javadoc: com.saxonica.ptree.PTreeURIResolver]`, to be used in place of the standard URIResolver. The `PTreeURIResolver` recognizes any URI ending in the extension `.ptree` as an identifier for a file in PTree format. This extends to files loaded using the `doc()` or `document()` functions: if the file extension is `.ptree`, the file will be assumed to be in PTree format.

The result of a query or transformation can be serialized as a PTree file by specifying `saxon:ptree` as the output method, where the namespace prefix `saxon` represents the URI `http://saxon.sf.net/`.

The PTree format is designed to allow future Saxon releases to read files created using older releases. The converse may not always be true: it might sometimes be impossible for release N to read a PTree file created using release N+1.

In releases up to and including Saxon 9.3, the PTree files were always at version 0. Saxon 9.4 introduces a new version, version 1. The new version differs in retaining DTD-derived attribute types (ID, IDREF, IDREFS). The `PTreeReader [Javadoc: com.saxonica.ptree.PTreeReader]` in Saxon 9.4 (onwards) will read both versions. The `PTreeWriter [Javadoc: com.saxonica.ptree.PTreeWriter]` in Saxon 9.4 writes version 1 output by default (which cannot be read by earlier releases), but it can still write original version 0 output if requested. If called from the command line, use the option `-version:1`.

The PTree format does not retain the base URI of the original file: when a PTree is loaded, the base URI is taken as the URI of that file, not the original XML file. The PTree is a serialization of the XPath data model, so information that isn't present in the data model will not be present in the PTree: for example, it will have no DTD and no entity references or CDATA sections.

References to unparsed entities are not currently retained in a PTree.

Validation of Source Documents

With Saxon-EE, source documents may be validated against a schema. Not only does this perform a check that the document is valid, it also adds type information to each element and attribute node in the document to identify the schema type against which it was validated. It may also expand the source document by adding default values of elements and attributes.

If the option `-val:strict` is specified on the command line for `com.saxonica.Query` or `com.saxonica.Transform`, then the principal source document to the query or transformation is schema-validated, as is every document loaded using the `doc()` or `document()` function. Saxon will look among all the loaded schemas for an element declaration that matches the outermost element of the document, and will then check that the document is valid against that element declaration, reporting a fatal error if it is not. The loaded schemas include schemas imported statically into the query or stylesheet using `import schema` or `xsl:import-schema`, schemas referenced in the `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation` attributes of the source document itself, and schemas loaded by the application using the `addSchema` method of the `Configuration [Javadoc: net.sf.saxon.Configuration]` object.

As an alternative to `-val:strict`, the option `-val:lax` may be specified. This validates the document if and only if an element declaration can be found. If there is no declaration of the outermost element in any loaded schema, then it is left as an untyped document.

When invoking transformations or queries from the Java API, the equivalent of the `-val:strict` option is to call the method `setSchemaValidation(Validation.STRICT)` on the `Configuration [Javadoc: net.sf.saxon.Configuration]` object. The equivalent of `-val:lax` is `setSchemaValidation(Validation.LAX)`.

When documents are built using the `DocumentBuilder [Javadoc: net.sf.saxon.s9api.DocumentBuilder]` in the s9api interface, or in the Saxon.Api interface on .NET, validation may be controlled by setting the appropriate options on the `DocumentBuilder`.

On Java interfaces that expect a JAXP `Source` object it is possible to request validation by supplying an `AugmentedSource [Javadoc: net.sf.saxon.lib.AugmentedSource]`. This consists of a `Source` and a set of options, including validation options; since `AugmentedSource` implements the JAXP `Source` interface it is possible to use it anywhere that a `Source` is expected, including as the object returned by a user-written `URIResolver`.

Saxon's standard `URIResolver` uses this technique if it has been enabled (for example by using `-p` on the command line). With this option, any URI containing the query parameter `?val=strict` (for example, `doc('source.xml?val=strict')`) causes strict validation to be requested for that document, while `?val=lax` requests lax validation, and `?val=strip` requests no validation.

Whitespace Stripping in Source Documents

A number of factors combine to determine whether whitespace-only text nodes in the source document are visible to the user-written XSLT or XQuery code.

By default, if there is a DTD or schema, then is stripped from any source document loaded from a `StreamSource` or `SAXSource`. Ignorable whitespace is defined as the whitespace that appears separating the child elements in element declared to have element-only content. This whitespace is removed regardless of any `xml:space` attributes in the source document.

It is possible to change this default behavior in several ways.

- From the Transform or Query command line, options are available: `-strip:all` strips all whitespace text nodes, `-strip:none` strips no whitespace text nodes, and `-strip:ignorable` strips ignorable whitespace text nodes only (this is the default).
- If the `-p` option is used on the command line, then query parameters are recognized in the URI passed to the `document()` or `doc()` function. The parameter `strip-space=yes` strips all whitespace text nodes, `strip-space=no` strips no whitespace text nodes, and `strip-space=ignorable` strips ignorable whitespace text nodes only. This overrides anything specified on the command line.
- Options corresponding to the above can also be set on the `TransformerFactory` object or on the `Configuration [Javadoc: net.sf.saxon.Configuration]`. These settings are global.

Whitespace stripping that is specified in any of the above ways does not occur only if the source document is parsed under Saxon's control: that is, if it supplied as a JAXP `StreamSource` or `SAXSource`. It also applies where the input is supplied in the form of a tree (for example, a DOM). In this case Saxon wraps the supplied tree in a virtual tree that provides a view of the original tree with whitespace text nodes omitted.

This whitespace stripping is additional (and prior) to any stripping carried out as a result of the `xsl:strip-space` declaration in the stylesheet.

Streaming of Large Documents

Sometimes source documents are too large to hold in memory. Saxon-EE provides a range of facilities for processing such documents in : that is, processing data as it is read by the XML parser, without building a complete tree representation of the document in memory.

Some of these facilities implement new features in the draft XSLT 3.0 standard (also known as XSLT 2.1). Some are specific to Saxon, and a few facilities are also available in XQuery.

Inevitably there are things that cannot be done in streaming mode - sorting is an obvious example. Sometimes, achieving a streaming transformation means rethinking the design of how it works - for example, splitting it into multiple phases. So streaming is rarely a case of simply taking your existing code and setting a simple switch to request streamed implementation.

There are basically two ways of doing streaming in Saxon:

- Burst-mode streaming: with this approach, the transformation of a large file is broken up into a sequence of transformations of small pieces of the file. Each piece in turn is read from the input, turned into a small tree in memory, transformed, and written to the output file.

  This approach works well for files that are fairly flat in structure, for example a log file holding millions of log records, where the processing of each log record is independent of the ones that went before.

  A variant of this technique uses the new XSLT 3.0 `xsl:iterate` instruction to iterate over the records, in place of `xsl:for-each`. This allows working data to be maintained as the records are processed: this makes it possible, for example, to output totals or averages at the end of the run, or to make the processing of one record dependent on what came before it in the file. The `xsl:iterate` instruction also allows early exit from the loop, which makes it possible for a transformation to process data from the beginning of a large file without actually reading the whole file.

  Burst-mode streaming is available in both XSLT and XQuery, but there is no equivalent in XQuery to the `xsl:iterate` construct.

- Streaming templates: this approach follows the traditional XSLT processing pattern of performing a recursive descent of the input XML hierarchy by matching template rules to the nodes at each level, but does so one element at a time, without building the tree in memory.

  Every template belongs to a `mode` (perhaps the default, unnamed mode), and streaming is a property of the mode that can be specified using the new `xsl:mode` declaration. If the mode is declared to be streamable, then every template rule within that mode must obey the rules for streamable processing.

  The rules for what is allowed in streamed processing are quite complicated, but the essential principle is that the template rule for a given node can only read the descendants of that node once, in order. There are further rules imposed by limitations in the current Saxon implementation: for example, although grouping using `<xsl:for-each-group group-adjacent="xxx">` is theoretically consistent with a streamed implementation, it is not currently implemented in Saxon.

  The streamed template mechanism applies to XSLT only.

Both these facilities are available in Saxon-EE only. Streamed templates also require XSLT 3.0 to be enabled by setting the relevant configuration parameters or command line options.

- Burst-mode streaming
- Processing the nodes returned by saxon:stream()
- Reading source documents partially
- Streamable path expressions
- How burst-mode streaming works
- Using saxon:stream() with saxon:iterate
- Streaming Templates

Burst-mode streaming

The `saxon:stream` extension function enables burst-mode streaming by reading a source document and delivering a sequence of element nodes representing selected elements within that document. For example:

```
saxon:stream(doc('employees.xml')/*/employee)
```

This example returns a sequence of `employee` elements. These elements are parentless, so it is not possible to navigate from one employee element to others in the file; in fact, only one of them actually exists in memory at any one time.

The function `saxon:stream` may be regarded as a pseudo-function. Conceptually, it takes the set of nodes supplied in its argument, and makes a deep copy of each one (the copy operation is needed to make the `employee` elements parentless). The resulting sequence of nodes will usually be processed by an instruction such as `xsl:for-each` or `xsl:iterate`, or by a FLWOR expression in XQuery, which handles the nodes one at a time. The actual implementation of `saxon:stream`, however, is rather different, in that it changes the way in which its argument is evaluated: instead of the `doc()` function building a tree in the normal way, the path expression `doc('employees.xml')/*/employee)` is evaluated in streamed mode - which means that it must conform to a subset of the XPath syntax which Saxon can evaluate in streamed mode. For details of this subset, see Streamable path expressions

The facility should not be used if the source document is read more than once in the course of the query/transformation. There are two reasons for this: firstly, performance will be better in this case if the document is read into memory; and secondly, when this optimization is used, there is no guarantee that the `doc()` function will be stable, that is, that it will return the same results when called repeatedly with the same URI.

If the path expression cannot be evaluated in streaming mode, execution does not fail; rather it is evaluated with an unoptimized copy-of instruction. This will give the same results provided enough memory is available for this mode of evaluation. To check whether streamed processing is actually being used, set the -t option from the command line or the `FeatureKeys.TIMING` option from the configuration API; the output will indicate whether a particular source document has been processed by building a tree, or by streaming.

In XSLT an alternative way of invoking the facility is by using an `<xsl:copy-of>` instruction with the special attribute `saxon:read-once="yes"`. Typically the `xsl:copy-of` instruction will form the body of a stylesheet function, which can then be called in the same way as `saxon:stream` to deliver the stream of records. This approach has the advantage that the code is portable to other XSLT processors (`saxon:read-once="yes"` is an extension attribute, a processing hint that other XSLT processors are required to ignore.)

In XQuery the same effect can be achieved using a pragma `(# saxon:read-once #)`. Again, processors other than Saxon are required to ignore this pragma.

Example: selective copying

A very simple way of using this technique is when making a selective copy of parts of a document. For example, the following code creates an output document containing all the `footnote` elements from the source document that have the attribute `@type='endnote'`:

```
<xsl:template name="main">
  <footnotes>
    <xsl:sequence select="saxon:stream(doc('thesis.xml')//footnote[@type='endno
                  xmlns:saxon="http://saxon.sf.net/"/>
  </footnotes>
</xsl:template>
```

```
  <footnotes>{
     saxon:stream(doc('thesis.xml')//footnote[@type='endnote'])
  }</footnotes>
```

XSLT example using xsl:copy-of

To allow code to be written in a way that will still work with processors other than Saxon, the facility can also be invoked using extension attributes in XSLT. Using this syntax, the previous example can be written as:

```
<xsl:template name="main">
  <footnotes>
    <xsl:copy-of select="doc('thesis.xml')//footnote[@type='endnote']"
                   saxon:read-once="yes" xmlns:saxon="http://saxon.sf.net/"/>
  </footnotes>
</xsl:template>
```

XQuery example using the saxon:stream pragma

In XQuery the pragma `saxon:stream` is available as an alternative to the function of the same name, allowing the code to be kept portable. The above example can be written:

```
  <footnotes>{
    (# saxon:stream #) {
       doc('thesis.xml')//footnote[@type='endnote']
    }
  }</footnotes>
```

Note the restrictions below on the kind of predicate that may be used.

Processing the nodes returned by saxon:stream()

The nodes selected by the streamed expression may be further processed. For example:

```
<xsl:template name="main">
  <xsl:apply-templates select="saxon:stream(doc('customers.xml')/*/customer)"
                       xmlns:saxon="http://saxon.sf.net/"/>
</xsl:template>

<xsl:template match="customer">
  <xsl:value-of select="code, name, location" separator="|"/>
  <xsl:text>&#xa;</xsl:text>
</xsl:template>
```

```
declare function f:customers() {
   saxon:stream(doc('customers.xml')/*/customer)
};

for $c in f:customers()
return concat(string-join(($c/code, $c/name, $c/location), '|'), '&#xa;')
```

Conceptually, `saxon:stream()` evaluates the sequence supplied in its first argument as a sequence of nodes, and then makes copies of these nodes as described in the rules of the `xsl:copy-of` instruction. The significance of the (notional) copy operation is that the returned nodes have no ancestors or siblings; each is the root of its own tree.

The document that is processed in streaming mode must be read using the `doc()` function (or in XSLT, the `document()` function). The query or stylesheet may also process other documents (for example a document named on the command line) but this is not necessary. In XSLT it is often useful to activate the stylesheet at a named template using the `-it` option on the command line, which allows activation without a primary input document.

When streaming copy is used, the relevant calls on the `doc()` or `document()` functions are not : that is, there is no guarantee that if the same document is read more than once, its contents will be unchanged. This is because the whole point of the facility is to ensure that Saxon does not need to keep the content of the document in memory. This limitation explains the choice of the keyword `read-once`: the facility should not be used to process a document if it needs to be read more than once during the query or transformation.

Reading source documents partially

As well as allowing a source document to be processed in a single sequential pass, the streaming facility in many cases allows the source document to be read only partially. For example, the following query will return true as soon as it finds a transaction with a negative value, and will then immediately stop processing the input file:

```
some $t in saxon:stream(doc('big-transaction-file.xml')//transaction)
satisfies number($t/@value) lt 0
```

This facility is particularly useful for extracting data that appears near the start of a large file. It does mean, however, that well-formedness or validity errors appearing later in the file will not necessarily be detected.

Streamable path expressions

The expression used as an argument to the `saxon:stream` function must consist of:

1. A call to the `document()` or `doc()` function, followed by
2. A streamable pattern

Streamable patterns use a subset of XPath expression corresponding roughly to the rules for match patterns in XSLT (the reason for this is that both subsets are designed to make it efficient to test an individual node for membership of the selected set of nodes). There are some extensions and some restrictions.

- Unlike XSLT match patterns, streamable patterns are not allowed to perform arbitrary navigation within a predicate. For example, `employee[id = preceding-sibling::employee/id]` is not allowed.

  More specifically, the predicate must not be positional (that is, it must not evaluate to a number, and must not call `position()` or `last()`), and it must only use downward selection from the context node (the self, child, attribute, descendant, descendant-or-self, or namespace axes)

- The streamable pattern that follows `doc()/` in the argument to `saxon:stream` must be a relative path: unlike XSLT match patterns, it may not start with "/" or "//" or with a call to the `key()` or `id()` function.

- Some of the restrictions in XSLT match patterns are relaxed, however: for example, the descendant axis can be used.

How burst-mode streaming works

Where necessary, the implementation of burst-mode streaming will use multithreading. One thread (which operates as a push pipeline) is used to read the source document and filter out the nodes selected by the path expression. The nodes are then handed over to the main processing thread, which iterates over the selected nodes using an XPath pull pipeline. Because multithreading is used, this facility is not used when tracing is enabled. It should also be disabled when using a debugger (there is a method in the Configuration object to achieve this.)

In cases where the entire stylesheet or query can be evaluated in "push" mode (as in the first example above), there is no need for multithreading: the selected nodes are written directly to the current output destination.

Note that a tree is built for each selected node, and its subtree. Trees are also built for all nodes selected by the path expression, whether or not the satisfy the filter (if they do not satisfy the filter, they will be immediately discarded from memory). The saving in memory comes when these nodes are processed one at a time, because each subtree can then be discarded as soon as it has been processed. There is no benefit if the stylesheet needs to perform non-serial processing, such as sorting. There is also no benefit if the path expression selects a node that contains most or all of the source document, for example its outermost element.

Saxon can handle expressions that select nested nodes, for example `//section` where one section contains another. However, the need to deliver nodes in document order makes the pipeline somewhat turbulent in such cases, increasing memory usage.

Streamed processing in this way is not actually faster than conventional processing (in fact, when multithreading is required, it may only run at half the speed). Its big advantage is that it saves memory, thus making it possible to process documents that would otherwise be too large for XSLT to handle. There may also be environments where the multithreading enables greater use of the processor capacity available. To run without this optimization, either change the `xsl:copy-of` instruction to `xsl:sequence`, or set `saxon:read-once` to "no".

Using saxon:stream() with saxon:iterate

In the examples given above, `saxon:stream()` is used to select a sequence of element nodes from the source document, and each of these nodes is then processed independently. In cases where the processing of one node depends in some way on previous nodes, it is possible to use `saxon:stream()` in conjunction with the `saxon:iterate` extension element in XSLT. (For details see saxon:iterate.)

The following example takes a sequence of `<transaction>` elements in an input document, each one containing the value of a debit or credit from an account. As output it copies the transaction elements, adding a current balance.

```
<saxon:iterate select="saxon:stream(doc('transactions.xml')/account/trans
  <xsl:param name="balance" as="xs:decimal" select="0.00"/>
  <xsl:variable name="new-balance" as="xs:decimal" select="$balance + xs:
  <transaction balance="{$new-balance}">
     <xsl:copy-of select="@*"/>
  </transaction>
  <saxon:continue>
    <xsl:with-param name="balance" select="$new-balance"/>
  </saxon:continue>
</saxon:iterate>
```

The following example is similar: this time it copies the account number (contained in a separate element at the start of the file) into each transaction element:

```
<saxon:iterate select="saxon:stream(doc('transactions.xml')/account/(acco
  <xsl:param name="accountNr"/>
  <xsl:choose>
     <xsl:when test="self::account-number">
       <saxon:continue>
          <xsl:with-param name="accountNr" select="string(.)"/>
       </saxon:continue>
     </xsl:when>
     <xsl:otherwise>
       <transaction account-number="{$accountNr}">
         <xsl:copy-of select="@*"/>
       </transaction>
     </xsl:otherwise>
  </xsl:choose>
```

```
</saxon:iterate>
```

Here is a more complex example, one that groups adjacent transaction elements having the same date attribute. The two loop parameters are the current grouping key and the current date. The contents of a group are accumulated in a variable until the date changes.

```
<saxon:iterate select="saxon:stream(doc('transactions.xml')/account/trans
<xsl:param name="group" as="element(transaction)*" select="()"/>
<xsl:param name="currentDate" as="xs:date?" select="()"/>
  <xsl:choose>
    <xsl:when test="xs:date(@date) eq $currentDate or empty($group)">
      <saxon:continue>
        <xsl:with-param name="currentDate" select="@date"/>
        <xsl:with-param name="group" select="($group, .)"/>
      </saxon:continue>
    </xsl:when>
    <xsl:otherwise>
      <daily-transactions date="{$currentDate}">
        <xsl:copy-of select="$group"/>
      </daily-transactions>
      <saxon:continue>
        <xsl:with-param name="group" select="."/>
        <xsl:with-param name="currentDate" select="@date"/>
      </saxon:continue>
    </xsl:otherwise>
  </xsl:choose>
  <saxon:finally>
    <final-daily-transactions date="{$currentDate}">
      <xsl:copy-of select="$group"/>
    </final-daily-transactions>
  </saxon:finally>
</saxon:iterate>
```

Note that when a `saxon:iterate` loop is terminated using `saxon:break`, parsing of the source document will be abandoned. This provides a convenient way to read data near the start of a large file without incurring the cost of reading the entire file.

Streaming Templates

Streaming templates allow a document to be processed hierarchically in the classical XSLT style, applying template rules to each element (or other nodes) in a top-down manner, while scanning the source document in a pure streaming fashion, without building the source tree in memory. Saxon-EE allows streamed processing of a document using template rules, provided the templates conform to a set of strict guidelines. The facility was introduced in a very simple form in Saxon 9.2, and is greatly enhanced in Saxon 9.3.

Streaming is a property of a ; a mode can be declared to be streamable, and if it is so declared, then all template rules using that mode must obey the rules for streamability. A mode is declared to be streamable using the top-level stylesheet declaration:

<xsl:mode name="s" streamable="yes"/>

The `name` attribute is optional; if omitted, the declaration applies to the default (unnamed) mode.

Streamed processing of a source document can be applied either to the principal source document of the transformation, or to a secondary source document read using the `doc()` or `document()` function.

To use streaming on the principal source document, the input to the transformation must be supplied in the form of a `StreamSource` or `SAXSource`, and the initial mode selected on entry to the transformation must be a streamable mode. In this case there must be no references to the context item in the initializer of any global variable.

Streamed processing of a secondary document is initiated using the instruction:

<xsl:apply-templates select="doc('abc.xml')" mode="s"/>

Here the `select` attribute must contain a simple call on the `doc()` or `document()` function, and the mode (explicit or implicit) must be declared as streamable. The call on `doc()` or `document()` can be extended with a streamable selection path, for example `select="doc('employee.xml')/*/employee"`

If a mode is declared as streamable, then it must ONLY be used in streaming mode; it is not possible to apply templates using a streaming mode if the selected nodes are ordinary non-streamed nodes.

Every template rule within a streamable mode must follow strict rules to ensure it can be processed in a streaming manner. The essence of these rules is:

1. The match pattern for the template rule must be a simple pattern that can be evaluated when positioned at the start tag of an element, without repositioning the stream (but information about the ancestors of the element and their attribute is available). Examples of acceptable patterns are `*`, `para`, or `para/*`

2. The body of the template rule must contain at most one expression or instruction that reads the contents below the matched element (that is, children or descendants), and it must process the contents in document order. This expression or instruction will often be one of the following:

   - `<xsl:apply-templates/>`
   - `<xsl:value-of select="."/>`
   - `<xsl:copy-of select="."/>`
   - `string(.)`
   - `data(.)` (explicitly or implicitly)

   but this list is not exhaustive. It is possible to process the contents selectively by using a streamable path expression, for example:

   - `<xsl:apply-templates select="foo"/>`
   - `<xsl:value-of select="a/b/c"/>`
   - `<xsl:copy-of select="x/y"/>`

   but this effectively means that the content not selected by this path is skipped entirely; the transformation ignores it.

   The template can access attributes of the context item without restriction, as well as properties such as its `name()`, `local-name()`, and `base-uri()`. It can also access the ancestors of the context item, the attributes of the ancestors, and properties such as the name of an ancestor; but having navigated to an ancestor, it cannot then navigate downwards or sideways, since the siblings and the other descendants of the ancestor are not available while streaming.

   The restriction that only one downwards access is allowed makes it an error to use an expression such as `price - discount` in a streamable template. This problem can often be circumvented by making a copy of the context item. This can be done using an `xsl:variable` containing an `xsl:copy-of` instruction, or for convenience it can also be done using the `copy-of()` function: for example `<xsl:value-of select="copy-of(.)/(price - discount)"/>`. Taking a copy of the context node requires memory, of course, and should be avoided unless the contents of the node are small.

The following rules gives further advice on what is allowed and disallowed within the body of a streaming template.

Non-context-sensitive instructions

Instructions and expressions that do not access the context node are allowed without restriction.

This includes:

- Instructions that create new nodes, for example literal result elements, `xsl:element` and `xsl:attribute` are allowed without restriction.
- Instructions that declare variables, including temporary trees, if the value of the variable does not depend on the context.
- Instructions that process documents other than the streamed document, for example by calling the `doc()` or `document()` functions. Provided such processing is not streamed, the full capabilities of the XSLT language can be used.

Access to attributes and ancestors

Access to attributes: there are no restrictions on accessing attributes of the context node, or attributes of its ancestors.

Properties of the context node: there are no restrictions on using functions such as `name()`, `node-name()`, or `base-uri()` to access properties of the context node, or properties of its ancestors, its attributes, or attributes of its ancestors. It is also possible to use the `is` operator to test the identity of the node, the `<<` and `>>` operators to test its position in document order, or the `instance of` operator to test its type. For attribute nodes it is possible to use (explicitly or implicitly) the `string()` function to get its string value and the `data()` function to get its typed value.

It is not possible to perform navigation from the attributes of the node or from its ancestors, only to access the values of attributes and properties such as the name of the node.

It is not possible to bind a variable (or pass a parameter, or return a result) to a node in the streamed document, because Saxon does not currently include the logic to analyse that the way in which the variable is subsequently used is consistent with streaming.

Conditional instructions

This includes `xsl:if`, `xsl:choose`, and the XPath `if` expression. All of these are regarded as special cases of a construct of the form `if (condition-1) then action-1 else if (condition-2) then action2 else ...`

The rule is that the conditional must fit one of the following descriptions:

- The first condition makes a downward selection, in which case none of the actions and none of the subsequent conditions may make a downward selection
- The first condition makes no downward selection, in which case each of the actions is allowed to make a downward selection (but subsequent conditions must not do so).

So examples of permitted conditionals are:

- `if (@a = 3) then b else c`
- `if (a = 3) then @b else @c`

while the following are not permitted:

- `if (a = 3) then b else c`

- `<xsl:choose> <xsl:when test="a=3">foo</xsl:when> <xsl:when test="a=4">bar</xsl:when> </xsl:choose>`

Looping instructions

This applies primarily to `xsl:for-each` and `xsl:iterate`. In addition, an XPath expression `for $x in SEQ return E` is translated to an equivalent `xsl:for-each` instruction, provided that `E` does not depend on the context item, position, or size.

The common case is where the `select` expression and the loop body each make a downward selection, for example:

```
<xsl:for-each select="employee">
  <salary><xsl:value-of select="salary"/></salary>
</xsl:for-each>
```

The body of the loop may only make a single downwards selection of this kind.

No sorting is allowed.

If the `select` expression does not make a downward selection, then the loop body must not perform any navigation from the context node. This is because the same navigation would have to take place more than once, which is inconsistent with streaming.

Saxon handles the case where some reordering of the output is required. This arises when the `select` expression uses the descandant axis, for example:

```
<xsl:for-each select=".//section">
  <size><xsl:value-of select="string-length(.)"/></size>
</xsl:for-each>
```

In this example, given nested sections, the downward selections for each section needed to evaluate `string-length()` overlap with each other, and the string-length of section 2.1 (say) must be output before that of its children (sections 2.1.1 and 2.1.2, say), even though the computation for the children completes earlier. Saxon achieves this by buffering output results where necessary to achieve the correct ordering.

It is of course quite permissible to call `xsl:apply-templates` within the body of the `xsl:for-each`; this will count as the one permitted downward selection.

It is permitted to call `position()` within the loop, but not `last()`.

Sorting, grouping and numbering

Sorting (`xsl:sort`), grouping (`xsl:for-each-group`), and numbering (`xsl:number`) are not supported in streaming mode.

Document Projection

Document Projection is a mechanism that analyzes a query to determine what parts of a document it can potentially access, and then while building a tree to represent the document, leaves out those parts of the tree that cannot make any difference to the result of the query.

Document projection can be enabled as an option on the XQuery command line interface: set `-projection:on`. It is only used if requested. The command line option affects both the primary source document supplied on the command line, and any calls on the `doc()` function within the body of the query that use a literal string argument for the document URI.

For feedback on the impact of document projection in terms of reducing the size of the source document in memory, use the `-t` option on the command line, which shows for each document loaded how many nodes from the input document were retained and how many discarded.

From the s9api API, document projection can be invoked as an option on the `DocumentBuilder`. The call `setDocumentProjectionQuery()` supplies as its argument a compiled query (an `XQueryExecutable`), and the document built by the document builder is then projected to retain only the parts of the document that are accessed by this query, when it operates on this document as the initial context item. For example, if the supplied query is `count(//ITEM)`, then only the `ITEM` elements will be retained.

It is also possible to request that a query should perform document projection on documents that it reads using the `doc()` function, provided this has a string-literal argument. This can be requested using the option `setAllowDocumentProjection(true)` on the `XQueryExpression` object. This is not available directly in the s9api interface, but the `XQueryExpression` is reachable from the `XQueryExecutable` using the accessor method `getUnderlyingCompiledQuery()`.

> It is best to avoid supplying a query that actually returns nodes from the document supplied as the context item, since the analysis cannot know what the invoker of the query will want to do with these nodes. For example, the query `<out>{//ITEM}</out>` works better than `//ITEM`, since it is clear that all descendants of the `ITEM` elements must be retained, but not their ancestors. If the supplied query selects nodes from the input document, then Saxon assumes that the application will need access to the entire subtree rooted at these nodes, but that it will not attempt to navigate upwards or outwards from these nodes. On the other hand, nodes that are atomized (for example in a filter) will be retained without their descendants, except as needed to compute the filter.

The more complex the query, the less likely it is that Saxon will be able to analyze it to determine the subset of the document required. If precise analysis is not possible, document projection has no effect. Currently Saxon makes no attempt to analyze accesses made within user-defined functions. Also, of course, Saxon cannot analyze the expectations of external (Java) functions called from the query.

Currently document projection is supported only for XQuery, and it works only when a document is parsed and loaded for the purpose of executing a single query. It is possible, however, to use the mechanism to create a manual filter for source documents if the required subset of the document is known. To achieve this, create a query that selects the required parts of the document supplied as the context item, and compile it to a s9api `XQueryExecutable`. The query does not have to do anything useful: the only requirement is that the result of the query on the subset document must be the same as the result on the original document. Then supply this `XQueryExecutable` to the s9api `DocumentBuilder` used to build the document.

Of course, when document projection is used manually like this then it entirely a user responsibility to ensure that the selected part of the document contains all the nodes required.

References to W3C DTDs

During 2010-11, W3C took steps to reduce the burden of meeting requests for commonly-referenced documents such as the DTD for XHTML. The W3C web server is routinely rejecting such requests, causing parsing failures. In response to this, Saxon now includes copies of these documents within the issued JAR file, and recognizes requests for these documents, satisfying the request using the local copy.

This is done only in cases where Saxon itself instantiates the XML parser. In cases where the user application instantiates an XML parser, the same effect can be achieved by setting the EntityResolver `StandardEntityResolver [Javadoc: net.sf.saxon.lib.StandardEntityResolver]` as a property of the `XMLReader` (parser).

The documents recognized by the `StandardEntityResolver` are:

Table 7.3.

Public ID	System ID	Saxon resource name
-//W3C//ENTITIES Latin 1 for XHTML//EN	http://www.w3.org/TR/xhtml1/DTD/xhtml-lat1.ent	w3c/xhtml-lat1.ent
-//W3C//ENTITIES Symbols for XHTML//EN	http://www.w3.org/TR/xhtml1/DTD/xhtml-symbol.ent	w3c/xhtml-symbol.ent
-//W3C//ENTITIES Special for XHTML//EN	http://www.w3.org/TR/xhtml1/DTD/xhtml-special.ent	w3c/xhtml-special.ent
-//W3C//DTD XHTML 1.0 Transitional//EN	http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd	w3c/xhtml10/xhtml1-transitional.dtd
-//W3C//DTD XHTML 1.0 Strict//EN	http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd	w3c/xhtml10/xhtml1-strict.dtd
-//W3C//DTD XHTML 1.0 Frameset//EN	http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd	w3c/xhtml10/xhtml1-frameset.dtd
-//W3C//DTD XHTML Basic 1.0//EN	http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd	w3c/xhtml10/xhtml-basic10.dtd
-//W3C//DTD XHTML 1.1//EN	http://www.w3.org/MarkUp/DTD/xhtml11.dtd	w3c/xhtml11/xhtml11.dtd
-//W3C//DTD XHTML Basic 1.1//EN	http://www.w3.org/MarkUp/DTD/xhtml-basic11.dtd	w3c/xhtml11/xhtml-basic11.dtd
-//W3C//ELEMENTS XHTML Access Element 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-access-1.mod	w3c/xhtml11/xhtml-access-1.mod
-//W3C//ENTITIES XHTML Access Attribute Qnames 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-access-qname-1.mod	w3c/xhtml11/xhtml-access-qname-1.mod
-//W3C//ELEMENTS XHTML Java Applets 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-applet-1.mod	w3c/xhtml11/xhtml-applet-1.mod
-//W3C//ELEMENTS XHTML Base Architecture 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-arch-1.mod	w3c/xhtml11/xhtml-arch-1.mod
-//W3C//ENTITIES XHTML Common Attributes 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-attribs-1.mod	w3c/xhtml11/xhtml-attribs-1.mod
-//W3C//ELEMENTS XHTML Base Element 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-base-1.mod	w3c/xhtml11/xhtml-base-1.mod
-//W3C//ELEMENTS XHTML Basic Forms 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-basic-form-1.mod	w3c/xhtml11/xhtml-basic-form-1.mod
-//W3C//ELEMENTS XHTML Basic Tables 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-basic-table-1.mod	w3c/xhtml11/xhtml-basic-table-1.mod
-//W3C//ENTITIES XHTML Basic 1.0 Document Model 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-basic10-model-1.mod	w3c/xhtml11/xhtml-basic10-model-1.mod
-//W3C//ENTITIES XHTML Basic 1.1 Document Model 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-basic11-model-1.mod	w3c/xhtml11/xhtml-basic11-model-1.mod
-//W3C//ELEMENTS XHTML BDO Element 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-bdo-1.mod	w3c/xhtml11/xhtml-bdo-1.mod
-//W3C//ELEMENTS XHTML Block Phrasal 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-blkphras-1.mod	w3c/xhtml11/xhtml-blkphras-1.mod

Public ID	System ID	Saxon resource name
-//W3C//ELEMENTS XHTML Block Presentation 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-blkpres-1.mod	w3c/xhtml11/xhtml-blkpres-1.mod
-//W3C//ELEMENTS XHTML Block Structural 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-blkstruct-1.mod	w3c/xhtml11/xhtml-blkstruct-1.mod
-//W3C//ENTITIES XHTML Character Entities 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-charent-1.mod	w3c/xhtml11/xhtml-charent-1.mod
-//W3C//ELEMENTS XHTML Client-side Image Maps 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-csismap-1.mod	w3c/xhtml11/xhtml-csismap-1.mod
-//W3C//ENTITIES XHTML Datatypes 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-datatypes-1.mod	w3c/xhtml11/xhtml-datatypes-1.mod
-//W3C//ELEMENTS XHTML Editing Markup 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-edit-1.mod	w3c/xhtml11/xhtml-edit-1.mod
-//W3C//ENTITIES XHTML Intrinsic Events 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-events-1.mod	w3c/xhtml11/xhtml-events-1.mod
-//W3C//ELEMENTS XHTML Forms 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-form-1.mod	w3c/xhtml11/xhtml-form-1.mod
-//W3C//ELEMENTS XHTML Frames 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-frames-1.mod	w3c/xhtml11/xhtml-frames-1.mod
-//W3C//ENTITIES XHTML Modular Framework 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-framework-1.mod	w3c/xhtml11/xhtml-framework-1.mod
-//W3C//ENTITIES XHTML HyperAttributes 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-hyperAttributes-1.mod	w3c/xhtml11/xhtml-hyperAttributes-1.mod
-//W3C//ELEMENTS XHTML Hypertext 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-hypertext-1.mod	w3c/xhtml11/xhtml-hypertext-1.mod
-//W3C//ELEMENTS XHTML Inline Frame Element 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-iframe-1.mod	w3c/xhtml11/xhtml-iframe-1.mod
-//W3C//ELEMENTS XHTML Images 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-image-1.mod	w3c/xhtml11/xhtml-image-1.mod
-//W3C//ELEMENTS XHTML Inline Phrasal 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-inlphras-1.mod	w3c/xhtml11/xhtml-inlphras-1.mod
-//W3C//ELEMENTS XHTML Inline Presentation 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-inlpres-1.mod	xhtml11/xhtml-inlpres-1.mod
-//W3C//ELEMENTS XHTML Inline Structural 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-inlstruct-1.mod	w3c/xhtml11/xhtml-inlstruct-1.mod
-//W3C//ENTITIES XHTML Inline Style 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-inlstyle-1.mod	w3c/xhtml11/xhtml-inlstyle-1.mod
-//W3C//ELEMENTS XHTML Inputmode 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-inputmode-1.mod	w3c/xhtml11/xhtml-inputmode-1.mod
-//W3C//ELEMENTS XHTML Legacy Markup 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-legacy-1.mod	w3c/xhtml11/xhtml-legacy-1.mod
-//W3C//ELEMENTS XHTML Legacy Redeclarations 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-legacy-redecl-1.mod	w3c/xhtml11/xhtml-legacy-redecl-1.mod
-//W3C//ELEMENTS XHTML Link Element 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-link-1.mod	w3c/xhtml11/xhtml-link-1.mod
-//W3C//ELEMENTS XHTML Lists 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-list-1.mod	w3c/xhtml11/xhtml-list-1.mod

Public ID	System ID	Saxon resource name
-//W3C//ELEMENTS XHTML Metainformation 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-meta-1.mod	w3c/xhtml11/xhtml-meta-1.mod
-//W3C//ELEMENTS XHTML Metainformation 2.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-meta-2.mod	w3c/xhtml11/xhtml-meta-2.mod
-//W3C//ENTITIES XHTML MetaAttributes 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-metaAttributes-1.mod	w3c/xhtml11/xhtml-metaAttributes-1.mod
-//W3C//ELEMENTS XHTML Name Identifier 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-nameident-1.mod	w3c/xhtml11/xhtml-nameident-1.mod
-//W3C//NOTATIONS XHTML Notations 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-notations-1.mod	w3c/xhtml11/xhtml-notations-1.mod
-//W3C//ELEMENTS XHTML Embedded Object 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-object-1.mod	w3c/xhtml11/xhtml-object-1.mod
-//W3C//ELEMENTS XHTML Param Element 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-param-1.mod	w3c/xhtml11/xhtml-param-1.mod
-//W3C//ELEMENTS XHTML Presentation 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-pres-1.mod	w3c/xhtml11/xhtml-pres-1.mod
-//W3C//ENTITIES XHTML-Print 1.0 Document Model 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-print10-model-1.mod	w3c/xhtml11/xhtml-print10-model-1.mod
-//W3C//ENTITIES XHTML Qualified Names 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-qname-1.mod	w3c/xhtml11/xhtml-qname-1.mod
-//W3C//ENTITIES XHTML +RDFa Document Model 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-rdfa-model-1.mod	w3c/xhtml11/xhtml-rdfa-model-1.mod
-//W3C//ENTITIES XHTML RDFa Attribute Qnames 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-rdfa-qname-1.mod	w3c/xhtml11/xhtml-rdfa-qname-1.mod
-//W3C//ENTITIES XHTML Role Attribute 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-role-1.mod	w3c/xhtml11/xhtml-role-1.mod
-//W3C//ENTITIES XHTML Role Attribute Qnames 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-role-qname-1.mod	w3c/xhtml11/xhtml-role-qname-1.mod
-//W3C//ELEMENTS XHTML Ruby 1.0//EN	http://www.w3.org/TR/ruby/xhtml-ruby-1.mod	w3c/xhtml11/xhtml-ruby-1.mod
-//W3C//ELEMENTS XHTML Scripting 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-script-1.mod	w3c/xhtml11/xhtml-script-1.mod
-//W3C//ELEMENTS XHTML Server-side Image Maps 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-ssismap-1.mod	w3c/xhtml11/xhtml-ssismap-1.mod
-//W3C//ELEMENTS XHTML Document Structure 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-struct-1.mod	w3c/xhtml11/xhtml-struct-1.mod
-//W3C//DTD XHTML Style Sheets 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-style-1.mod	w3c/xhtml11/xhtml-style-1.mod
-//W3C//ELEMENTS XHTML Tables 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-table-1.mod	w3c/xhtml11/xhtml-table-1.mod
-//W3C//ELEMENTS XHTML Target 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-target-1.mod	w3c/xhtml11/xhtml-target-1.mod
-//W3C//ELEMENTS XHTML Text 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml-text-1.mod	w3c/xhtml11/xhtml-text-1.mod

Public ID	**System ID**	**Saxon resource name**
-//W3C//ENTITIES XHTML 1.1 Document Model 1.0//EN	http://www.w3.org/MarkUp/DTD/xhtml11-model-1.mod	w3c/xhtml11/xhtml11-model-1.mod
-//W3C//MathML 1.0//EN	http://www.w3.org/Math/DTD/mathml1/mathml.dtd	w3c/mathml/mathml1/mathml.dtd
-//W3C//DTD MathML 2.0//EN	http://www.w3.org/Math/DTD/mathml2/mathml2.dtd	w3c/mathml/mathml2/mathml2.dtd
-//W3C//DTD MathML 3.0//EN	http://www.w3.org/Math/DTD/mathml3/mathml3.dtd	w3c/mathml/mathml3/mathml3.dtd
-//W3C//DTD SVG 1.0//EN	http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd	w3c/svg10/svg10.dtd
-//W3C//DTD SVG 1.1//EN	http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd	w3c/svg11/svg11.dtd
-//W3C//DTD SVG 1.1 Tiny//EN	http://www.w3.org/Graphics/SVG/1.1/DTD/svg11-tiny.dtd	w3c/svg11/svg11-tiny.dtd
-//W3C//DTD SVG 1.1 Basic//EN	http://www.w3.org/Graphics/SVG/1.1/DTD/svg11-basic.dtd	w3c/svg11/svg11-basic.dtd
-//XML-DEV//ENTITIES RDDL Document Model 1.0//EN	http://www.rddl.org/xhtml-rddl-model-1.mod	w3c/rddl/xhtml-rddl-model-1.mod
-//XML-DEV//DTD XHTML RDDL 1.0//EN	http://www.rddl.org/rddl-xhtml.dtd	w3c/rddl/rddl-xhtml.dtd
-//XML-DEV//ENTITIES RDDL QName Module 1.0//EN	http://www.rddl.org/rddl-qname-1.mod	rddl/rddl-qname-1.mod
-//XML-DEV//ENTITIES RDDL Resource Module 1.0//EN	http://www.rddl.org/rddl-resource-1.mod	rddl/rddl-resource-1.mod
-//W3C//DTD Specification V2.10//EN	http://www.w3.org/2002/xmlspec/dtd/2.10/xmlspec.dtd	w3c/xmlspec/xmlspec.dtd
-//W3C//DTD XMLSCHEMA 200102//EN	http://www.w3.org/2001/XMLSchema.dtd	w3c/xmlschema/XMLSchema.dtd

This Saxon feature can be disabled by setting the configuration property `FeatureKeys.ENTITY_RESOLVER_CLASS` `[Javadoc: net.sf.saxon.lib.FeatureKeys]` to null; it is also possible to set it to a different `EntityResolver` class (perhaps a subclass of Saxon's `StandardEntityResolver`) that varies the behavior. If an `EntityResolver` is set in the relevant `ParseOptions` or in an `AugmentedSource` then this will override any `EntityResolver` set at the configuration level.

Chapter 8. XML Schema Processing

Introduction

Saxon can be used as a free-standing schema processor in its own right, either from the command line or from a Java application. In addition, Saxon can be used as a schema-aware XSLT processor or as a schema-aware XQuery processor.

Saxon-EE supports the schema validation APIs in JAXP 1.3, as well as its own native APIs.

Running Validation from the Command Line

The Java class `com.saxonica.Validate` allows you to validate a source XML document against a given schema, or simply to check a schema for internal correctness.

To validate one or more source documents, using the Java platform, write:

java com.saxonica.Validate [options] source.xml...

The equivalent on the .NET platform is:

Validate [options] source.xml...

It is possible to use glob syntax to process multiple files, for example `Validate *.xml`.

In the above form, the command relies on the use of `xsi:schemaLocation` attributes within the instance document to identify the schema to be loaded. As an alternative, the schema can be specified on the command line:

[java com.saxonica.Validate | Validate] -xsd:schema.xsd -s:instance.xml

In this form of the command, it is possible to specify multiple schema documents and/or multiple instance documents, in both cases as a semicolon-separated list. Glob syntax (such as `*.xml`) is available only if the "-s:" prefix is omitted, because the shell has to recognize the argument as a filename.

Thus, source files to be validated can be listed either using the `-s` option, or in any argument that is not prefixed with "-". This allows the standard wildcard expansion facilities of the shell interpreter to be used, for example `*.xml` validates all files in the current directory with extension "xml".

If no instance documents are supplied, the effect of the command is simply to check a schema for internal correctness. So a schema can be verified using the command:

[java com.saxonica.Validate | Validate] -xsd:schema.xsd

More generally the syntax of the command is:

[java com.saxonica.Validate | Validate] [options] [params] [filenames]

where options generally take the form `-code:value` and params take the form `keyword=value`.

The options are as follows (in any order):

Table 8.1.

-catalog:filenames	is either a file name or a list of file names separated by semicolons; the files are OASIS XML catalogs used to define how public identifiers and system identifiers (URIs) used in a source document or schema are to be redirected,

	typically to resources available locally. For more details see Using XML Catalogs.
-config:filename	Loads options from a configuration file. This must describe a schema-aware configuration.
-init:initializer	The value is the name of a user-supplied class that implements the interface `net.sf.saxon.lib.Initializer [Javadoc: net.sf.saxon.lib.Initializer]`; this initializer will be called during the initialization process, and may be used to set any options required on the `Configuration [Javadoc: net.sf.saxon.Configuration]` programmatically.
-limits:min,max	Sets upper limits on the values of minOccurs and maxOccurs allowed in a schema content model, in cases where Saxon is not able to implement the rules using a finite state machine with counters. For further details see Handling minOccurs and maxOccurs
-r:classname	Use the specified `URIResolver` to process the URIs of all schema documents and source documents. The URIResolver is a user-defined class, that implements the `URIResolver` interface defined in JAXP, whose function is to take a URI supplied as a string, and return a SAX `InputSource`. It is invoked to process URIs found in `xs:include` and `xs:import schemaLocation` attributes of schema documents, the URIs found in `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes in the source document, and (if -u is also specified) to process the URI of the source file provided on the command line.Specifying `-r:org.apache.xml.resolver.tools.CatalogResolv` selects the Apache XML resolver (part of the Apache Commons project, which must be on the classpath) and enables URIs to be resolved via a catalog, allowing references to external web sites to be redirected to local copies.
-s:file;file...	Supplies a list of source documents to be validated. Each document is validated using the same options. The value is a list of filenames separated by semicolons. It is also possible to specify the names of source documents as arguments without any preceding option flag; in this case shell wildcards can be used. A filename can be specified as "-" to read the source document from standard input.
-scmin:filename	Loads a precompiled schema component model from the given file. The file should be generated in a previous run using the -scmout option. When this option is used, the -xsd option should not be

	present. Schemas loaded from an SCM file are assumed to be valid, without checking.
-scmout:filename	Makes a copy of the compiled schema (providing it is valid) as a schema component model to the specified XML file. This file will contain schema components corresponding to all the loaded schema documents. This option may be combined with other options: the SCM file is written after all document instance validation has been carried out.
-stats:filename	Requests creation of an XML document containing statistics showing which schema components were used during the validation episode, and how often (coverage data). This data can be used as input to further processes to produce user-readable reports; for example the data could be combined with the output of `-scmout` to show which components were not used at all during the validation.
-t	Requests display of version and timing information to the standard error output. This also shows all the schema documents that have been loaded.
-top:element-name	Requires that the outermost element of the instance being validated has the required name. This is written in Clark notation format "{uri}local".
-u	Indicates that the name of the source document and schema document are supplied as URIs; otherwise they are taken as filenames, unless they start with "http:" or "file:", in which case they they are taken as URLs.
-val:strict|lax	Invokes strict or lax validation (default is strict). Lax validation validates elements only if there is an element declaration to validate them against, or if they have an `xsi:type` attribute.
-x:classname	Requests use of the specified SAX parser for parsing the source file. The classname must be the fully-qualified name of a Java class that implements the `org.xml.sax.XMLReader` interface. In the absence of this argument, the standard JAXP facilities are used to locate an XML parser. Note that the XML parser performs the raw XML parsing only; Saxon always does the schema validation itself.Selecting `-x:org.apache.xml.resolver.tools.ResolvingXMLR` selects a parser configured to use the Apache entity resolver, so that DTD and other external references in source documents are resolved via a catalog. The parser (part of the Apache Commons project) must be on the classpath.
-xi:on|off	Apply XInclude processing to all input XML documents (both schema documents and instance documents). This currently only works when

	documents are parsed using the Xerces parser, which is the default in JDK 1.5 and later.
-xmlversion:1.0|1.1	If set to 1.1, allows XML 1.1 and XML Namespaces 1.1 constructs. This option must be set if source documents using XML 1.1 are to be validated, or if the schema itself is an XML 1.1 document. This option causes types such as xs:Name, xs:QName, and xs:ID to use the XML 1.1 definitions of these constructs.
-xsd:file;file...	Supplies a list of schema documents to be used for validation. The value is a list of filenames separated by semicolons. If no source documents are supplied, the schema documents will be processed and any errors in the schema will be notified. This option must not be used when `-scmin` is specified. The option may be omitted, in which case the schema to be used for validation will be located using the `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes in the source document. A filename can be specified as "-" to read the schema from standard input.
-xsiloc:on|off	If set to "on" (the default) the schema processor attempts to load any schema documents referenced in `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes in the instance document, unless a schema for the specified namespace (or non-namespace) is already available. If set to "off", these attributes are ignored.
--:value	Set a feature defined in the `Configuration [Javadoc: net.sf.saxon.Configuration]` interface. The names of features are defined in the Javadoc for class `FeatureKeys [Javadoc: net.sf.saxon.lib.FeatureKeys]`: the value used here is the part of the name after the last "/", for example `--allow-external-functions:off`. Only features accepting a string or boolean may be set; for booleans the values true/false or on/off are recognized.
-?	Display command syntax

The results of processing the schema, and of validating the source document against the schema, are written to the standard error output. Unless the `-t` option is used, successful processing of the source document and schema results in no output.

Controlling Validation from Java

Schema validation can be controlled either using the standard JAXP Java interface, or using Saxon's own interface. The two approaches are described in the following sections. The main advantage of using JAXP is that it is portable; the main advantage of s9api is that it is better integrated across the range of Saxon XML processing interfaces.

- Schema Processing using s9api

• Schema Processing using JAXP

Schema Processing using s9api

The s9api interface allows schemas to be loaded into a `Processor [Javadoc: net.sf.saxon.s9api.Processor]`, and then to be used for validating instances, or for schema-aware XSLT and XQuery processing.

The main steps are:

1. Create a Processor (`net.sf.saxon.s9api.Processor [Javadoc: net.sf.saxon.s9api.Processor]`) and call its `getSchemaManager()` method to get a `SchemaManager [Javadoc: net.sf.saxon.s9api.SchemaManager]`.
2. If required, set options on the `SchemaManager [Javadoc: net.sf.saxon.s9api.SchemaManager]` to control the way in which schema documents will be loaded.
3. Load a schema document by calling the `load()` method, which takes a JAXP Source object as its argument. The resulting schema document is available to all applications run within the containing `Processor [Javadoc: net.sf.saxon.s9api.Processor]`.
4. To validate an instance document, call the `newSchemaValidator` method on the `SchemaManager [Javadoc: net.sf.saxon.s9api.SchemaManager]` object.
5. Set options on the `SchemaValidator [Javadoc: net.sf.saxon.s9api.SchemaValidator]` to control the way in which a particular validation episode is performed, and then invoke its `validate()` method to validate an instance document.

Note that additional schemas referenced from the `xsi:schemaLocation` attributes within the source documents will be loaded as necessary. A target namespace is ignored if there is already a loaded schema for that namespace; Saxon makes no attempt to load multiple schemas for the same namespace and check them for consistency.

Although the API is defined in such a way that a `SchemaValidator [Javadoc: net.sf.saxon.s9api.SchemaValidator]` is created for a particular `SchemaManager [Javadoc: net.sf.saxon.s9api.SchemaManager]`, in the Saxon implementation the schema components that are available to the validator are not only the components within that schema, but all the components that form part of any schema registered with the `Processor` (or indeed, with the underlying `Configuration`)

The `SchemaValidator` implements the `Destination [Javadoc: net.sf.saxon.s9api.Destination]` interface, which means it can be used to receive input from any process that writes to a `Destination`, for example an XSLT transformation or an XQuery query. The result of validation can also be sent to any `Destination`, for example an XSLT transformer.

Schema Processing using JAXP

Applications can invoke schema processing using the APIs provided in JAXP 1.3. This makes Saxon interchangeable with other schema processors implementing this interface. There is full information on these APIs in the Java documentation. The two main mechanisms are the `Validator` class, and the `ValidatorHandler` class. Sample applications using these interfaces are provided in the `samples/java` directory. Saxon also supplies the class `com.saxonica.jaxp.ValidatingReader`, which implements the SAX2 `XMLReader` interface, allowing it to be used as a schema-validating XML parser.

The main steps are:

1. Create a `SchemaFactory`, by calling `SchemaFactory.getInstance()` with the argument `"http://www.w3.org/2001/XMLSchema"`, and with the Java system properties set up to ensure that Saxon is loaded as the chosen schema processor. Saxon will normally be loaded as the default schema processor if Saxon-EE is present on the classpath, but to make absolutely sure, set the system property `javax.xml.validation.SchemaFactory:http://www.w3.org/2001/XMLSchema` to the value `com.saxonica.jaxp.SchemaFactoryImpl [Javadoc: com.saxonica.jaxp.SchemaFactoryImpl]`. Note that if you set this property using a property file, colons in the property name must be escaped as "\:".
2. Process a schema document by calling one of the several `newSchema` methods on the returned `SchemaFactory`.
3. Create either a `Validator` or a `ValidatorHandler` from this returned `Schema`.
4. Use the `Validator` or `ValidatorHandler` to process one or more source documents.

Note that additional schemas referenced from the `xsi:schemaLocation` attributes within the source documents will be loaded as necessary. A target namespace is ignored if there is already a loaded schema for that namespace; Saxon makes no attempt to load multiple schemas for the same namespace and check them for consistency.

Although the API is defined in such a way that a `Validator` or `ValidatorHandler` is created for a particular `Schema`, in the Saxon implementation the schema components that are available to the validator are not only the components within that schema, but all the components that form part of any schema registered with the `Configuration [Javadoc: net.sf.saxon.Configuration]`.

Another way to control validation from a Java application is to run a JAXP identity transformation, having first set the option to perform schema validation. The following code (from the sample application `QuickValidator.java`) illustrates this:

```
try {
    System.setProperty(
            "javax.xml.transform.TransformerFactory",
            "com.saxonica.SchemaAwareTransformerFactory");
    TransformerFactory factory =
            TransformerFactory.newInstance();
    factory.setAttribute(FeatureKeys.SCHEMA_VALIDATION,
            new Integer(Validation.STRICT));
    Transformer trans = factory.newTransformer();
    StreamSource source =
            new StreamSource(new File(args[0]).toURI().toString());
    SAXResult sink =
            new SAXResult(new DefaultHandler());
    trans.transform(source, sink);
} catch (TransformerException err) {
    System.err.println("Validation failed");
}
```

If you set an `ErrorListener` on the `TransformerFactory`, then you can control the way that error messages are output.

If you want to validate against a schema without hard-coding the URI of the schema into the source document, you can do this by pre-loading the schema into the `TransformerFactory`. This extended example (again from the sample application `QuickValidator.java`) illustrates this:

```
try {
    System.setProperty(
            "javax.xml.transform.TransformerFactory",
            "com.saxonica.SchemaAwareTransformerFactory");
    TransformerFactory factory =
            TransformerFactory.newInstance();
    factory.setAttribute(FeatureKeys.SCHEMA_VALIDATION,
            new Integer(Validation.STRICT));
    if (args.length > 1) {
        StreamSource schema =
                new StreamSource(new File(args[1]).toURI().toString());
        ((SchemaAwareTransformerFactory)factory).addSchema(schema);
    }
    Transformer trans = factory.newTransformer();
    StreamSource source =
            new StreamSource(new File(args[0]).toURI().toString());
    SAXResult sink =
            new SAXResult(new DefaultHandler());
    trans.transform(source, sink);
} catch (TransformerException err) {
    System.err.println("Validation failed");
}
```

You can preload as many schemas as you like using the `addSchema` method. Such schemas are parsed, validated, and compiled once, and can be used as often as you like for validating multiple source documents. You cannot unload a schema once it has been loaded. If you want to remove or replace a schema, start afresh with a new `TransformerFactory`.

Behind the scenes, the `TransformerFactory` uses a `Configuration` object to hold all the configuration information. The basic Saxon product uses the class `net.sf.saxon.TransformerFactoryImpl` for the `TransformerFactory`, and `net.sf.saxon.Configuration` for the underlying configuration information. The schema-aware product subclasses these with `com.saxonica.config.SchemaAwareTransformerFactory` `[Javadoc: com.saxonica.config.SchemaAwareTransformerFactory]` and `com.saxonica.config.EnterpriseConfiguration` `[Javadoc: com.saxonica.config.EnterpriseConfiguration]` respectively. You can get hold of the `Configuration` object by casting the `TransformerFactory` to a Saxon `TransformerFactorImpl` and calling the `getConfiguration()` method. This gives you more precise control, for example it allows you to retrieve the `Schema` object containing the schema components for a given target namespace, and to inspect the compiled schema to establish its properties. See the JavaDoc documentation for further details.

The programming approach outlined above, of using an identity transformer, is suitable for a wide class of applications. For example, it enables you to insert a validation step into a SAX-based pipeline. However, for finer control, there are lower-level interfaces available in Saxon that you can also use. See for example the JavaDoc for the `EnterpriseConfiguration` `[Javadoc: com.saxonica.config.EnterpriseConfiguration]` class, which includes methods such as `getElementValidator`. This constructs a `Receiver [Javadoc: net.sf.saxon.event.Receiver]` which acts as a validating XML event filter. This can be inserted into a pipeline of `Receiver`s. Saxon also provides classes to bridge between SAX events and `Receiver` events: `ReceivingContentHandler [Javadoc: net.sf.saxon.event.ReceivingContentHandler]` and `ContentHandlerProxy [Javadoc: net.sf.saxon.event.ContentHandlerProxy]` respectively.

Running validation from Ant

It is possible to use the Saxon schema validator using the standard Ant tasks `xmlvalidate` and `schemavalidate`. To use Saxon rather than Xerces as the validation engine, specify the attribute `classname="com.saxonica.jaxp.ValidatingReader"`, and make sure Saxon-EE is on the classpath.

The schema to be used for validation can be specified using the `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes in the instance document, or (in the case of the `schemavalidate` task) using the `schemavalidate/schema` child element or the `schemavalidate/@noNamespaceFile` or `schemavalidate/@noNamespaceURL` attributes.

The attributes `lenient` and `fullchecking` have no effect.

The child element `schemavalidate/attribute` can be used to set options. Any option defined by the constants in class `net.sf.saxon.FeatureKeys` can be specified, provided the required value is expressible as a string (for boolean values, use "true" and "false"). Saxon also recognizes some property names defined by the Apache Xerces product, for compatibility.

Properties of particular interest include the following:

Table 8.2.

Name	Value
http://saxon.sf.net/feature/licenseFileLocation	The filename where the Saxon-EE license file is found
http://saxon.sf.net/feature/schemaURIResolverClass	Class used to resolve URIs of schema documents
http://saxon.sf.net/feature/schema-validation-mode	`strict` or `lax`: determines whether validation fails if no element declaration can be found for the top-level element.
http://saxon.sf.net/feature/standardErrorOutputFile	Log file to capture validation errors
http://saxon.sf.net/feature/xsd-version	`1.0` or `1.1` depending on the version of the XML Schema (XSD) Recommendation to be supported. Default is `1.0`

Schema-Aware XSLT from the Command Line

To run a schema-aware transformation from the command line, use the `com.saxonica.Transform [Javadoc: com.saxonica.Transform]` command instead of the usual `net.sf.saxon.Transform`. This has an additional option `-val:strict` to request strict validation of the source document, or `-val:lax` for lax validation. This applies not only to the principal source document loaded from the command line, but to all documents loaded via the `doc()` and `document()` functions.

The schemas to be used to validate these source documents can be specified either by using the `xsl:import-schema` declaration in the stylesheet, or using `xsi:schemaLocation` (or `xsi:noNamespaceSchemaLocation`) attributes within the source documents themselves, or by using the `-xsd` option on the command line.

Validating the source document has several effects. Most obviously, it will cause the transformation to fail if the document is invalid. It will also cause default values for attributes and elements to be expanded, so they will appear to the stylesheet as if they were present on the source document. In addition, element and attribute nodes that have been validated will be annotated with a type. This enables operations to be performed in a type-safe way. This may cause error messages, for example if you try to use an `xs:decimal` value as an argument to a function that expects a string. It may also cause some operations to produce different results: for example when using elements or attribute that have been given a list type in the schema, the typed value of the node will appear in the stylesheet as a sequence rather than as a single string value.

Saxon-EE also allows you to validate result documents (both final result documents and temporary trees), using the `validation` and `type` attributes. For details of these, refer to the XSLT 2.0 specification. Validation of result documents is done on-the-fly, so if the stylesheet attempts to produce invalid output, you will usually get an error message that identifies the offending instruction in the stylesheet. Type annotations on final result documents are lost if you send the output to a standard JAXP `Result` object (whether it's a `StreamResult`, `SAXResult`, or `DOMResult`, but they remain available if you capture the output in a Saxon `Receiver` or in a `DOMResult` that encapsulates a Saxon `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]`. For details of the way in which type annotations are represented in the Saxon implementation of the data model, see the JavaDoc documentation. The `getTypeAnnotation()` method on a `NodeInfo` object returns an integer fingerprint, which can be used to locate the name of the relevant type in the `NamePool [Javadoc: net.sf.saxon.om.NamePool]`. The `NamePool` also provides the ability to locate the actual type definitions in the schema model, starting from these integer fingerprints.

The `-vw` option on the command line causes validation errors encountered in processing a final result tree to be treated as warnings, allowing processing to continue. This allows more than one error to be reported in a single run. The result document is serialized as if validation were successful, but with XML comments inserted to show where the validation errors were found. This option does not necessarily recover from all validation errors, for example at present it does not recover from errors in uniqueness or referential constraints. It applies only to result trees validated using the `validation` attribute of `xsl:result-document`.

With the schema-aware version of Saxon, type declarations (the `as` attribute on elements such as `xsl:function`, `xsl:variable`, and `xsl:param`) can refer to schema-defined types, for example you can write `<xsl:variable name="a" as="schema-element(ipo:invoice)"/>`. You can also use the `element()` and `attribute()` tests to select nodes by their schema type in path expressions and match patterns.

Saxon does a certain amount of static analysis of the XSLT and XPath code based on schema information. For example, if a template rule is defined with a match pattern such as `match="schema-element(invoice)"`, then it will check any path expressions used in the template rule to ensure that they are valid against the schema when starting from `invoice` as the context node. Similarly, if the result type of a template rule or function is declared using an `as` attribute, then Saxon will check any literal result elements in the body of the template or function to ensure that they are consistent with this declared type. This analysis can reveal many simple user errors at compile time that would otherwise result in run-time errors or simply in incorrect output. But this is only possible if the source code explicitly declares the types of parameters, template and function results, and match patterns.

Schema-Aware XSLT from Java

When transformations are controlled using the Java JAXP interfaces, the equivalent to the `-val` option is to set the attribute "http://saxon.sf.net/feature/schema-validation" on the TransformerFactory to the value `net.sf.saxon.lib.Validation.STRICT [Javadoc: net.sf.saxon.lib.Validation#STRICT]`. Alternatively, you can set the value to `Validation.LAX [Javadoc: net.sf.saxon.lib.Validation#STRICT]`. This attribute name is available as the constant `FeatureKeys.SCHEMA_VALIDATION [Javadoc: net.sf.saxon.lib.FeatureKeys#SCHEMA_VALIDATION]`.

This option switches validation on for all source documents used by any transformation under the control of this `TransformerFactory`. If you want finer control, so that some documents are validated and others are not, you can achieve this by using the `AugmentedSource [Javadoc: net.sf.saxon.lib.AugmentedSource]` object. An `AugmentedSource` is a wrapper around a normal JAXP `Source` object, in which additional properties can be set: for example, a property to request validation of the document. The `AugmentedSource` itself implements the JAXP

Source interface, so it can be used anywhere that an ordinary Source object can be used, notably as the first argument to the transform method of the Transformer, and as the return value from a user-written URIResolver.

If the PTreeURIResolver [Javadoc: com.saxonica.ptree.PTreeURIResolver] is used, it is also possible to control validation for each source document by means of query parameters in the document URI. For example, document('source.xml?val=strict') requests the loading of the file source.xml with strict validation.

The attribute FeatureKeys.VALIDATION_WARNINGS [Javadoc: net.sf.saxon.lib.FeatureKeys#VALIDATION_WARNINGS] has the same effect as the -vw option on the command line: validation errors encountered when processing the final result tree are reported to the ErrorListener as warnings, not as fatal errors.

Schemas can be loaded using either of the techniques used with the command-line interface: that is, by specifying them in the xsl:import-schema directive in the stylesheet, or by including them in an xsi:schemaLocation attribute in a source document. In addition, they can be loaded using the addSchema() method on the SchemaAwareTransformerFactory [Javadoc: com.saxonica.config.SchemaAwareTransformerFactory] class.

All schemas that are loaded are cached as part of the TransformerFactory (or more specifically, as part of the Configuration [Javadoc: net.sf.saxon.Configuration] object owned by the TransformerFactory). This is true whether the schema is loaded explicitly using the Java API, whether it is loaded as a result of xsl:import-schema, or whether it is referenced in an xsi:schemaLocation attribute in a source document. There can only be one schema document loaded for each namespace: any further attempts to load a schema for a given target namespace will return the existing loaded schema, rather than loading a new one. Note in particular that this means there can only be one loaded no-namespace schema document. If you want to force loading of a different schema document for an existing namespace, the only way to do it is to create a new TransformerFactory.

If you are validating the result tree, and you want your application to have access to the type annotations in the validated tree, then you should specify as the result of the transformation either a user-written Receiver, or a DOMResult that wraps a Saxon DocumentInfo object. Note that type annotations are supported only with the TinyTree implementation.

Schema-Aware XQuery from the Command Line

To run a schema-aware query from the command line, use the usual command net.sf.saxon.Query [Javadoc: net.sf.saxon.Query]. This has an option -val:strict to request strict validation of the source document, or -val:lax to request lax validation. This applies not only to the principal source document loaded using the -s option on the command line, but to all documents loaded via the doc() functions, or supplied as additional command line parameters in the form +param=doc.xml.

The schemas to be used to validate these source documents can be specified either by using the import schema declaration in the query prolog, or using xsi:schemaLocation (or xsi:noNamespaceSchemaLocation) attributes within the source documents themselves, or by using the -xsd option on the command line.

Validating the source document has several effects. Most obviously, it will cause the query to fail if the document is invalid. It will also cause default values for attributes and elements to be expanded, so they will appear to the query as if they were present on the source document. In addition, element and attribute nodes that have been validated will be annotated with a type. This enables operations to be performed in a type-safe way. This may cause error messages, for example if you try to use an xs:decimal value as an argument to a function that expects a string. It may also cause some operations to produce different results: for example when using elements or attributes that have been given a list type in the schema, the typed value of the node will appear in the stylesheet as a sequence rather than as a single string value.

The enterprise edition of Saxon also allows you to validate result documents (both final result documents and intermediate results). By default, elements constructed by the query are validated in lax mode, which means that they are validated if a schema declaration is available, and are not validated otherwise. You can set a different initial validation mode either using the `declare validation` declaration in the Query Prolog, or by issuing a call such as `staticQueryContext.pushValidationMode(Validation.SKIP)` in the calling API.

The `-vw` option on the command line causes validation errors encountered in processing a final result tree to be treated as warnings, allowing processing to continue. This allows more than one error to be reported in a single run. The result document is serialized as if validation were successful, but with XML comments inserted to show where the validation errors were found. This option does not necessarily recover from all validation errors, for example at present it does not recover from errors in uniqueness or referential constraints.

By default, the validation context for element constructors in the query depends on the textual nesting of the element constructors as written in the query. You can change the validation context (and the validation mode) if you need to, by using a `validate{}` expression within the query. For details of this expression, refer to the XQuery 1.0 specification. Validation of result documents is done on-the-fly, so if the query attempts to produce invalid output, you will usually get an error message that identifies the approximate location in the query where the error occurred.

With the enterprise edition of Saxon, declarations of functions and variables can refer to schema-defined types, for example you can write `let $a as schema-element(ipo:invoice)* := //inv`. You can also use the `element()` and `attribute()` tests to select nodes by their schema type in path expressions.

Saxon-EE does a certain amount of static analysis of the XQuery code based on schema information. For example, if a function argument is defined with a type such as `as="schema-element(invoice)"`, then it will check any path expressions used in the function body to ensure that they are valid against the schema when starting from `invoice` as the context node. Similarly, if the result type of a function is declared using an `as` attribute, then Saxon will check any direct element constructors in the body of the function to ensure that they are consistent with this declared type. This analysis can reveal many simple user errors at compile time that would otherwise result in run-time errors or simply in incorrect output. But this is only possible if the source code explicitly declares the types of variables and of function arguments and results.

Schema-Aware XQuery from Java

When queries are controlled using the Java API, the equivalent to the `-val` option is to create a `EnterpriseConfiguration [Javadoc: com.saxonica.config.EnterpriseConfiguration]` instead of a `Configuration` object, and then to call `setSchemaValidationMode(net.sf.saxon.lib.Validation.STRICT)` on this object. The value `Validation.LAX [Javadoc: net.sf.saxon.lib.Validation#LAX]` can also be used.

This option switches validation on for all source documents used by any transformation under the control of this `EnterpriseConfiguration`. If you want finer control, so that some documents are validated and others are not, you can achieve this by using the `AugmentedSource [Javadoc: net.sf.saxon.lib.AugmentedSource]` object. An `AugmentedSource` is a wrapper around a normal JAXP `Source` object, in which additional properties can be set: for example, a property to request validation of the document. The `AugmentedSource` itself implements the JAXP `Source` interface, so it can be used anywhere that an ordinary `Source` object can be used, for example as the first argument to the `buildDocument()` method of the `QueryProcessor`, and as the return value from a user-written `URIResolver`.

If the `PTreeURIResolver [Javadoc: com.saxonica.ptree.PTreeURIResolver]` is used, it is also possible to control validation for each source document by means of query parameters in the document URI. For example, `doc('source.xml?val=strict')` requests the loading of the file `source.xml` with strict validation.

The `Configuration [Javadoc: net.sf.saxon.Configuration]` method `setValidationWarnings()` has the same effect as the `-vw` option on the command line: validation errors encountered when processing the final result tree are reported to the `ErrorListener` as warnings, not as fatal errors. They are also reported as XML comments in the result tree.

Schemas can be loaded using either of the techniques used with the command-line interface: that is, by specifying them in the `import schema` directive in the query prolog, or by including them in an `xsi:schemaLocation` attribute in a source document. In addition, they can be loaded using the `addSchemaSource()` method on the `EnterpriseConfiguration` class.

All schemas that are loaded are cached as part of the `EnterpriseConfiguration [Javadoc: com.saxonica.config.EnterpriseConfiguration]`. This is true whether the schema is loaded explicitly using the Java API, whether it is loaded as a result of `import schema` in a query, or whether it is referenced in an `xsi:schemaLocation` attribute in a source document. There can only be one schema document loaded for each namespace: any further attempts to load a schema for a given target namespace will return the existing loaded schema, rather than loading a new one. Note in particular that this means there can only be one loaded no-namespace schema document. If you want to force loading of a different schema document for an existing namespace, the only way to do it is to create a new `EnterpriseConfiguration`.

XML Schema 1.1

Saxon-EE release 9.4 includes full support for the XML Schema 1.1 specification, which at the time of writing is close to becoming a W3C Recommendation. The main changes between XSD 1.0 and XSD 1.1 are listed in the following pages. Note that these features are subject to change as the specifications mature.

To enable use of XML Schema 1.1 features, set the command line flag `-xsdversion:1.1` or the equivalent in the API.

- Assertions on Complex Types
- Assertions on Simple Types
- Conditional Type Assignment
- All Model Groups
- Open Content
- Miscellaneous XSD 1.1 Features

Assertions on Complex Types

Saxon-EE supports the definition of assertions on both simple and complex types.

Assertions enable cross-validation of different elements or attributes within a complex type. For example, specifying:

```
<xs:assert test="xs:date(@date-of-birth) lt xs:date(@date-of-death)"/>
```

will cause a run-time validation error if an instance document is validated in which the relevant condition does not hold.

Saxon allows any XPath 2.0 expression to be used in the `test` attribute. This includes expressions that call Java or .NET extension functions.

For assertions on complex types, the context node supplied to the expression is the element being validated. The element being validated is presented as type `xs:anyType`, but its attributes and children, because they have already been validated, are annotated with their respective types. The static context for the expression comes from the containing schema document: any namespace prefixes used in the expression must be declared using namespace declarations in the schema in the usual way. The default namespace for elements and types may be set using the `xpathDefaultNamespace` attribute either on the element containing the XPath expression, or on the `xs:schema` element). It is not possible to use any variables or user-defined functions within the expression.

For the purpose of generating diagnostics, Saxon recognizes an assertion of the form `empty(expr)` specially. For example, if you are validating an XSLT stylesheet, you might write on the top-level complex type `<xs:assert test="empty(if (@version='1.0') then xsl:variable[@as] else ())"/>`. If you use this form of assertion, the validator will not only report that the assertion is false for the top-level element, it will also report the location of all the `xsl:variable` elements that caused the assertion to be false. This also works for `not(expr)` provided that `expr` has a static item type of `node()`.

Another aid to diagnostics is the `saxon:message` attribute: if present on the `xs:assert` element, this provides a message to be output when the assertion is not satisfied: see saxon:message.

The XPath expression is evaluated against a temporary document that contains the subtree rooted at this element: more specifically, the subtree contains a document node with this element as its only child. Validation succeeds if the effective boolean value (EBV) of the expression is true, and fails if the EBV is false or if an error occurs during the evaluation.

If a complex type is derived by extension or by restriction, then the assertions supplied on the base type must be satisfied as well as those supplied on the type itself.

Note that when assertions are defined on a complex type, the subtree representing an element with that type will be built in memory. It is therefore advisable to exercise care when applying this facility to elements that have very large subtrees.

For assertions on simple types, `<xs:assertion>` is treated as a facet. It may be applied to any variety of type, that is to a type derived by restriction from an atomic type, a list type, or a union type. The value against which the assertion is being tested is available to the expression as the value of variable `$value`; this will be typed as an instance of the base type (the type being restricted). There is no context node. The variable `$value` is also available in the same way for complex types with simple content.

Assertions on Simple Types

Saxon allows assertions on simple types to be defined. The mechanism is to define an `xs:assertion` element as a child of the `xs:restriction` child of the `xs:simpleType` element (that is, it acts as an additional facet). The type must be an atomic type. The value of the `test` attribute of `xs:assert` is an XPath expression. .

The expression is evaluated with the value being validated supplied as the value of the variable `$value`. This will be an instance of the base type: for example, if you are restricting from `xs:string`, it will be a string; if you are restricting from `xs:date`, it will be an `xs:date`; if you are validating a list of integers, then `$value` will be a sequence of integers.

If the effective boolean value of the expression is true, the value is valid. If the effective boolean value is false, or if a dynamic error occurs while evaluating the expression, the value is invalid. Currently no diagnostics are produced to indicate why the value is deemed invalid, other than a statement that the `xs:assertion` facet is violated. You can supply a message in a `saxon:message` attribute: see saxon:message.

The XPath expression has no access to any part of the document being validated, other than the atomic value of the actual element or attribute node. So the validation cannot be context-sensitive.

The XPath expression may make calls on Java extension functions in the normal way: see Writing extension functions (Java). Allowing call-out to procedural programming languages means that you can perform arbitrary procedural validation of element and attribute values. Take care to disable use of extension functions if validating against a schema that is untrusted.

The following example validates that a date is in the past:

```
<xs:element name="date">
  <xs:simpleType>
     <xs:restriction base="xs:date">
       <xs:assertion test="$value lt current-date()"/>
     </xs:restriction>
  </xs:simpleType>
</xs:element>
```

The following example validates that a string is a legal XPath expression. This relies on the fact that a failure evaluating the assertion is treated as "false":

```
<xs:element name="xpath">
  <xs:simpleType>
     <xs:restriction base="xs:string">
       <xs:assertion test="exists(saxon:expression($value))" xmlns:saxon="htt
     </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Note how the in-scope namespaces for the XPath expression are taken from the in-scope namespaces of the containing `xs:assert` element.

Conditional Type Assignment

Saxon supports Conditional Type Assignment.

The full syntax of XPath 2.0 can be used, but the expression is constrained to access the element node and its attributes: it has no access to the descendants, siblings, or ancestors of the element.

All Model Groups

Saxon allows arbitrary values of `minOccurs` and `maxOccurs` on the particles of a model group using the compositor `xs:all`.

A complex type defined using `xs:all` may be derived by extension or restriction from another type using `xs:all`.

Element wildcards (`xs:any`) are allowed within `xs:all` content models.

Open Content

Open content is implemented.

The allows a complex type to specify open content with mode "interleave" or "suffix", allowing arbitrary elements (satisfying a wildcard) to be added either anywhere in the content sequence, or at the end.

At the level of a schema document, the `defaultOpenContent` defines the default open content mode for all types defined in the schema document (or, for all types except those with an empty content model).

Similarly, the `defaultAttributes` attribute of the `xs:schema` element defines a default attribute wildcard to be permitted for all complex types defined in the schema document.

Miscellaneous XSD 1.1 Features

The `notNamespace` and `notQName` attributes are now supported on `xs:any` and `xs:anyAttribute` wildcards. The `definedSibling` option is not currently implemented.

The `targetNamespace` attribute is available for use on local element and attribute declarations appearing within the restriction of a complex type.

Saxon allows conditional inclusion of elements in a schema document, using attributes such as `vc:minVersion` and `vc:maxVersion`. This feature is available whether the schema processor is run in 1.0 or 1.1 mode, allowing new 1.1 features such as assertions to be ignored when running in 1.0 mode.

The type `xs:error` is implemented.

An element may now appear in more than one substitution group.

A new facet `xs:explicitTimezone` is available with values required, optional, or prohibited.

The new built-in data type `xs:dateTimeStamp` (an `xs:dateTime` with timezone required) is implemented.

Importing and Exporting Schema Component Models

Saxon provides the ability to export or import a compiled schema. The export format is an XML file, known as an SCM file (for schema component model). Exporting a schema in SCM format makes it quicker to reload when the same schema is used again. It is also a format that is easier for programs to analyze, in comparison with raw XSD schema documents.

The simplest way to create an SCM file is from the command line, using the `com.saxonica.Validate` command with the `-scmout` option. This is described here. Alternatively, an SCM file can be generated programmatically using the `exportComponents()` method of the `com.saxonica.config.EnterpriseConfiguration [Javadoc: com.saxonica.config.EnterpriseConfiguration]` class, which is described in the JavaDoc. The serializer is unselective: it will output an SCM containing all the schema components that have been loaded into the `Configuration`, other than built-in schema components.

An SCM file can be imported using the `-scmin` option of the `com.saxonica.Validate` command. It can also be loaded programmatically using the `SchemaModelLoader [Javadoc: com.saxonica.schema.SchemaModelLoader]` class. For example:

```
SchemaModelLoader loader = new SchemaModelLoader(config);
loader.load(new StreamSource(new File("input.scm")));
```

A schema loaded in this way is then available for all tasks performed using this `Configuration`, including validation of source documents and compiling of schema-aware queries and stylesheets. In particular, it can be used when compiled queries are run under this `Configuration`.

Schema Component Models can also be imported and exported using the `importComponents()` and `exportComponents()` methods of the `SchemaManager` in the s9api interface.

The structure of an SCM file is defined in the schema `scmschema.xsd` which is available in the directory `samples/scm/` in the `saxon-resources` download file. This is annotated to explain the mappings between elements and attributes in the SCM file and components and properties as defined in the W3C XML Schema Specification. The same directory contains a file `scmschema.scm` which contains the schema for SCM in SCM format.

The SCM file includes a representation of the finite state machines used to validate instances against a complex type. This means that the FSM does not need to be regenerated when a schema is loaded from an SCM file, which saves a lot of time. However, it also means that the SCM format is not currently suitable as a target format for software-generated schemas. A variant of SCM in which the finite state machines can be omitted may be provided in a future release.

Handling minOccurs and maxOccurs

Prior to release 9.1, Saxon used the validation algorithm described in Thompson and Tobin 2003 [http://www.ltg.ed.ac.uk/~ht/XML_Europe_2003.html]. This algorithm can be very inefficient when large bounded values of minOccurs and maxOccurs are used in a content model; indeed, it can be so inefficient that the finite state machine is too large to fit in memory, and an OutOfMemory exception occurs.

Since Saxon 9.1, many common cases of minOccurs and maxOccurs are handled using a finite state machine that makes use of counters at run-time. This eliminates the need to have one state in the machine for each possible number of occurrences of the repeating item. Instead, counters are maintained at run-time and compared against the minOccurs and maxOccurs values.

This technique is used under the following circumstances:

- Either minOccurs > 1, or maxOccurs > 1 (and is not unbounded), or both
- The minOccurs/maxOccurs values must be defined on an element (xs:element) or wildcard (xs:any) particle
- If the repeating particle is , then it must not be part of a model group that is itself repeatable. A particle is vulnerable if it is part of a choice group, or if it is part of a sequence group in which all the other particles are optional or emptiable, except in the case where minOccurs is equal to maxOccurs. The reason for this restriction is that in such situations there are two nested repetitions, and it is ambiguous whether a new instance of the repeating term should be treated as a repetition at the inner level or at the outer level.

In cases where counters cannot be used, Saxon will still attempt to compile a finite state machine, but will use configuration-defined limits on minOccurs and maxOccurs to approximate the values requested. If the values used in the schema exceed these limits, Saxon will therefore approximate by generate a schema that does not strictly enforce the specified minOccurs and maxOccurs. The default limits are 100 and 250 respectively. Different limits can be set on the command line or via the Java API on the `Configuration` object. Note however that when several nested repeating groups are defined it is still possible for out-of-memory conditions to occur, even with quite modest values of minOccurs and maxOccurs.

Saxon extensions to XML Schema 1.1

The Working Draft XSD 1.1 specification allows implementations to define their own primitive types and facets.

At present Saxon provides one additional facet, `saxon:preprocess`

Saxon extensions to the XML Schema Language are implemented in the Saxon namespace `http://saxon.sf.net/`:

- Messages associated with assertions and other facets
- The saxon:preprocess facet
- Saxon extensions to XSD uniqueness and referential constraints

Messages associated with assertions and other facets

In assertions, and on all elements representing facets (for example `pattern`), Saxon supports the attribute `saxon:message="message text"`. This message text is used in error messages when the assertion or other facet is not satisfied.

For example:

```
<xs:element name="date">
  <xs:simpleType>
     <xs:restriction base="xs:date" xmlns:saxon="http://saxon.sf.net/">
       <xs:assertion test=". lt current-date()"
                  saxon:message="The date must not be in the future"/>
       <xs:pattern value="[^Z:]*"
                  saxon:message="The date must not have a timezone"/>
     </xs:restriction>
  </xs:simpleType>
</xs:element>
```

The saxon:preprocess facet

Saxon provides the `saxon:preprocess` facet as an addition to the standard facets defined in the XSD 1.1 specification. It is available only when XSD 1.1 support is enabled.

Like `xs:whiteSpace`, this is a pre-lexical facet. It is used to transform the supplied lexical value of an element or attribute from the form as written (but after whitespace normalization) to the lexical space of the base type. Constraining facets such as `pattern`, `enumeration`, and `minLength` apply to the value after the `saxon:preprocess` facet has done its work. In addition, if the primitive type is say `xs:date` or `xs:decimal`, the built-in lexical rules for parsing a date or a decimal number are applied only after `saxon:preprocess` has transformed the value. The makes it possible, for example, to accept `yes` and `no` as values of an `xs:boolean`, `3,14159` as the value of an `xs:decimal`, or `13DEC1987` as the value of an `xs:date`.

Like other facets, `saxon:preprocess` may be used as a child of `xs:restriction` when restricting a simple type, or a complex type with simple content.

The attributes are:

Table 8.3.

id	Standard attribute
action	Mandatory. An XPath expression. The rules for writing the XPath expression are generally the same as the rules for the `test` expression of `xs:assert`. The value to be transformed is supplied (as a string) as the value of the variable `$value`; the context item is undefined. The expression must return a single string. If evaluation of the expression fails with a dynamic error, this is interpreted as a validation failure.

reverse	Optional. An XPath expression used to reverse the transformation. Used (in XPath, XSLT, and XQuery) when a value of this type is converted to a string. When a value of this type is converted to a string, it is first converted according to the rules of the base type. The resulting string is then passed, as the value of variable `$value`, to the XPath expression, and the result of the XPath expression is used as the final output. This attribute does not affect the schema validation process itself.
xpathDefaultNamespace	The default namespace for element names (unlikely to appear in practice) and types.

The following example converts a string to upper-case before testing it against the enumeration facet.

```
<xs:simpleType name="currency">
  <xs:restriction base="xs:string">
    <xs:saxon:preprocess action="upper-case($value)" xmlns:saxon="http://saxon.sf.
    <xs:enumeration value="USD"/>
    <xs:enumeration value="EUR"/>
    <xs:enumeration value="GBP"/>
  </xs:restriction>
</xs:simpleType>
```

Of course, it is not only the constraining facets that will see the preprocessed value (in this case, the upper-case value), any XPath operation that makes use of the typed value of an element or attribute node will also see the value after preprocessing. However, the string value of the node is unchanged.

The following example converts any commas appearing in the input to full stops, allowing decimal numbers to be represented in Continental European style as `3,15`. On output, the process is reversed, so that full stops are replaced by commas.

```
<xs:simpleType name="euroDecimal">
  <xs:restriction base="xs:decimal">
    <saxon:preprocess action="translate($value, ',', '.')"
                      reverse="translate($value, '.', ',')"
                      xmlns:saxon="http://saxon.sf.net/"/>
  </xs:restriction>
</xs:simpleType>
```

Note that in this example, the user-defined type also accepts numbers written in the "standard" style `3.15`.

The following example allows an `xs:time` value to be written with the seconds part omitted. Again, it also accepts the standard `hh:mm:ss` notation:

```
<xs:simpleType name="hoursAndMinutes">
  <xs:restriction base="xs:time">
    <saxon:preprocess action="concat($value, ':00'[string-length($value) = 5])"
                      xmlns:saxon="http://saxon.sf.net/"/>
  </xs:restriction>
</xs:simpleType>
```

The following example uses extension function calls within the XPath expression to support integers written in hexadecimal notation:

```
<xs:simpleType name="hexInteger">
  <xs:restriction base="xs:long">
    <saxon:preprocess action="Long:parseLong($value, 16)" reverse="Long:toHexSt
      xmlns:Long="java:java.lang.Long"
      xmlns:saxon="http://saxon.sf.net/"/>
  </xs:restriction>
</xs:simpleType>
```

Given the input `<val>0040</val>`, validated against this schema, the query `(val*3) cast as hexInteger` will produce the output `c0`.

Saxon extensions to XSD uniqueness and referential constraints

This extension is only available if enabled by specifying the attribute `saxon:extensions="id-xpath-syntax"` on the `xs:schema` element of the containing schema document.

With this extension enabled, restrictions are removed on the syntax allowed in the `selector/@xpath` and `field/@xpath` attributes of the `xs:unique`, `xs:key`, and `xs:keyref` elements. Instead of the very limited XPath subset defined in the XSD 1.0 and XSD 1.1 specifications, Saxon will allow the same syntax as is permitted for streamable XPath expressions in XSLT. Specifically, the syntax for both attributes is the same as allowed in the `select` attribute of a streaming `xsl:apply-templates`, which is an extended form of the XSLT 3.0 syntax for patterns. It permits, for example, any sequence of downwards axes, arbitrary predicates on any step provided they do no downwards selection, and conditional expressions.

For example, the following is permitted, indicating that US-based employees must have a unique social security number, but not imposing any such constraint on other employees:

```
<xs:element name="company">
  <xs:unique>
    <xs:selector xpath="employee[@location='us']"/>
    <xs:field xpath="@ssid"/>
  </xs:unique>
</xs:element>
```

Chapter 9. XPath API for Java

Introduction

For information about the different ways of loading source documents, see Handling Source Documents.

Saxon supports three Java APIs for XPath processing, as follows:

- The JAXP API is a (supposedly) standard API defined in Java 5. Saxon implements this interface. Details of Saxon's implementation are described at The JAXP XPath API. Note that there are some extensions and other variations in the Saxon implementation. Some of the extensions to this interface are provided because Saxon supports XPath 2.0, whereas JAXP 1.3 is designed primarily for XPath 1.0; some are provided because Saxon supports multiple object models, not only DOM; some are for backwards compatibility; and some are provided to allow applications a finer level of control if required.
- The preferred interface for XPath processing is the s9api interface, which also supports XSLT and XQuery processing, schema validation, and other Saxon functionality in an integrated set of interfaces. This is described at Evaluating XPath Expressions using s9api
- For historical reasons Saxon continues to support a legacy XPath API in package `net.sf.saxon.sxpath`. This is documented only in the Javadoc specifications. It is a lower-level API, so as well as being retained for backwards compatibility, it may also be appropriate for applications that require a very intimate level of integration with Saxon internals.

Evaluating XPath Expressions using s9api

The s9api interface is a custom-designed API for Saxon, allowing integrated access to all Saxon's XML processing capabilities in a uniform way, taking advantage of the type safety offered by generics in Java 5.

You can evaluate an XPath expression using the s9api interface as follows:

1. Create a `Processor [Javadoc: net.sf.saxon.s9api.Processor]` and set any global configuration options on the `Processor`.
2. Build the source document by calling `newDocumentBuilder()` to create a document builder, setting appropriate options, and then calling the `build()` method. This returns an `XdmNode [Javadoc: net.sf.saxon.s9api.XdmNode]` which can be supplied as the context item to the XPath expression.
3. Call `newXPathCompiler()` to create an `XPathCompiler [Javadoc: net.sf.saxon.s9api.XPathCompiler]`, and set any options that are local to a specific compilation (notably declaring namespace prefixes that are used in the XPath expression).
4. Call the `compile()` method to compile an expression. The result is an `XPathExecutable [Javadoc: net.sf.saxon.s9api.XPathExecutable]`, which can be used as often as you like in the same thread or in different threads.
5. To evaluate the expression, call the `load()` method on the `XPathExecutable`. This creates an `XPathSelector [Javadoc: net.sf.saxon.s9api.Processor]`. The `XPathSelector` can be serially reused, but it must not be shared across multiple threads. Set any options required for the specific XPath execution (for example, the initial context node, the values of any variables referenced in the expression), and then call one of the methods `iterator()` `evaluate()`, or `evaluateSingle()` to execute the XPath expression.

6. Because the `XPathSelector` is an `Iterable`, it is possible to iterate over the results directly using the Java 5 "for-each" construct.
7. The result of an XPath expression is in general an `XdmValue [Javadoc: net.sf.saxon.s9api.XdmValue]`, representing a value as defined in the XDM data model (that is, a sequence of nodes and/or atomic values). Subclasses of `XdmValue` include `XdmItem [Javadoc: net.sf.saxon.s9api.XdmItem]`, `XdmNode [Javadoc: net.sf.saxon.s9api.XdmNode]`, and `XdmAtomicValue [Javadoc: net.sf.saxon.s9api.XdmAtomicValue]`, and these relate directly to the corresponding concepts in XDM. Various methods are available to translate between this model and native Java data types.

The `XdmCompiler [Javadoc: net.sf.saxon.s9api.XdmCompiler]` also has compile-and-go methods `evaluate()` and `evaluateSingle()` to execute an expression directly without going through an explicit compilation process. This provides a simpler approach if the expression is only evaluated once. The `XdmCompiler` also has the option of caching compiled expressions, so that if the same expression is evaluated repeatedly, the compiled form of the expression is re-used.

Examples of the use of s9api to evaluate XPath expressions are included in the Saxon resources file, see module S9APIExamples.java.

The JAXP XPath API

Saxon implements the JAXP 1.3 API for executing XPath expressions as defined in package `java.xml.xpath`, which is a standard part of the Java class library since JDK 1.5. Three sample applications using this API are available: they are called `XPathExample.java`, `XPathExampleSA.java`, and `ApplyXPathJAXP.java`, and can be found in the `samples/java` directory after downloading the `saxon-resources` archive.

To run the `XPathExample.java` application, see the instructions in Shakespeare XPath Sample Application.

The `XPathExampleSA.java` application demonstrates use of schema-aware XPath. It is designed to be used with the files `books.xml` and `books.xsd` in the directory `samples/data`.

The `ApplyXPathJAXP.java` application is an enhanced version of the class of the same name issued as a sample application in the JAXP 1.3 distribution. It has been enhanced to show the use of more advanced features, such as the ability to bind namespaces, variables, and functions, and also to demonstrate use of the XPath API with different object models.

The XPath API in Saxon predates the introduction of the JAXP 1.3 XPath API, so it contains some facilities that are obsolescent. However, there are also differences caused by the fact that Saxon supports XPath 2.0, with its richer data model, whereas JAXP 1.3 operates only on XPath 1.0.

The following sections describe use of the XPath API in more detail.

- Selecting the XPath implementation: Describes how to ensure that Saxon is selected as the XPath API implemenation
- Setting the context item: Describes how to supply a context item
- Return types: Describes the mapping of XPath return types to Java classes
- Additional Saxon methods: Describes features of the Saxon implementation additional to the JAXP specification
- Calling JAXP XPath extension functions: Describes how to call out from XPath expressions to Java code

Selecting the XPath implementation

An application using the JAXP 1.3 XPath API starts by instantiating a factory class. This is done by calling:

```
XPathFactory xpathFactory = XPathFactory.newInstance(objectModel);
XPath xpath = xpathFactory.newXPath();
```

Here `objectModel` is a URI that identifies the object model you are using. Saxon recognizes the following values for the object model:

Table 9.1.

XPathConstants.DOM_OBJECT_MODEL	The DOM object model
NamespaceConstant.OBJECT_MODEL_SAXON	Saxon's native object model. This means anything that implements the `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` interface, including the standard tree, the tiny tree, and third-party implementations of `NodeInfo`.
NamespaceConstant.OBJECT_MODEL_JDOM	The JDOM object model
NamespaceConstant.OBJECT_MODEL_XOM	The XOM object model
NamespaceConstant.OBJECT_MODEL_DOM4J	The DOM4J object model

To ensure that Saxon is selected as your XPath implementation, you must specify one of these constants as your chosen object model, and it is a good idea to ensure that the Java system property `javax.xml.xpath.XPathFactory` is set to the value `net.sf.saxon.xpath.XPathFactoryImpl`. Normally, if Saxon is on your classpath then the Saxon XPath implementation will be picked up automatically, but if there are other implementations on the classpath as well then it is best to set the system property explicitly to be sure.

Alternatively, if you know that you want to use the Saxon implementation, you can simply instantiate the class `net.sf.saxon.xpath.XPathFactoryImpl [Javadoc: net.sf.saxon.xpath.XPathFactoryImpl]` directly. If you want to take advantage of features in Saxon-PE (Professional Edition), use `com.saxonica.config.ProfessionalXPathFactory [Javadoc: com.saxonica.config.ProfessionalXPathFactory]`, or for Saxon-EE (Enterprise Edition) use `com.saxonica.config.EnterpriseXPathFactory [Javadoc: com.saxonica.config.EnterpriseXPathFactory]`.

It is important to note that a compiled XPath expression can only be used with a source document that was built using the same Saxon `Configuration [Javadoc: net.sf.saxon.Configuration]`. When you create an `XPathFactory`, a Saxon `Configuration` is created automatically. You can extract this configuration and use it to build source documents. Alternatively, there is a constructor that allows you to create an `XPathFactory` that uses a preexisting `Configuration`.

Saxon's implementation of `java.xml.xpath.XPath` is the class `net.sf.saxon.xpath.XPathEvaluator [Javadoc: net.sf.saxon.xpath.XPathEvaluator]`. This class provides a few simple configuration interfaces to set the source document, the static context, and the context node, plus a number of methods for evaluating XPath expressions.

The `XPath` object allows you to set the static context for evaluating XPath expressions (you can pre-declare namespaces, variables, and functions), and to compile XPath expressions in this context. A compiled XPath expression (an object of class `XPathExpression`) can then be evaluated, with a supplied node (represented by a class in the selected object model) supplied as the context node. For further details, see the Javadoc specifications and the supplied example applications.

Setting the context item

Many of the methods in the JAXP interface take an argument to represent the context node. Because the API is designed to be used with multiple object models, this is typed simply as an `Object`. In the case of Saxon, this can be an object in any of the supported data models: DOM, JDOM, DOM4J, XOM, or Saxon's native `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` representation. Note that if a `NodeInfo` is supplied, the tree must have been constructed using the same Saxon `Configuration [Javadoc: net.sf.saxon.config.Configuration]` as the one used by the `XPathEvaluator [Javadoc: net.sf.saxon.xpath.XPathEvaluator]` itself.

Many of these methods specify that if the context node supplied is null, the XPath expression will be evaluated using an empty document node as the context node. Saxon does not implement this rule. There are two reasons for this. Firstly, it's unnatural in XPath 2.0, where it is permissible for the context item for an expression to be undefined. Secondly, it's not clear what kind of document node it would be appropriate to create, given that Saxon supports multiple object models. Instead, if null is supplied, then the node supplied to the Saxon-specific method `setContextNode()` is used, and if that is null, the expression is evaluated with the context item undefined (which will cause a dynamic error if the expression actually refers to the context item).

Return types

The JAXP specification leaves it rather up to the implementation how the results of an XPath expression will be returned. This is partly because it is defined only for XPath 1.0, which has a much simpler type system, and partly because it is deliberately designed to be independent of the object model used to represent XML trees.

If you specify the return type `XPathConstants.BOOLEAN` then Saxon will return the effective boolean value of the expression, as a `java.lang.Boolean`. This is the same as wrapping the expression in a call of the XPath `boolean()` function.

If you specify the return type `XPathConstants.STRING` then Saxon will return the result of the expression converted to a string, as a `java.lang.String`. This is the same as wrapping the expression in a call of the XPath `string()` function.

If you specify the return type `XPathConstants.NUMBER` then Saxon will return the result of the expression converted to a double as a `java.lang.Double`. This is the same as wrapping the expression in a call of the XPath `number()` function.

If you specify the return type `XPathConstants.NODE` then Saxon will return the result the result as a node object in the selected object model. With the DOM model this will be an instance of `org.w3.dom.Node`, with the native Saxon model it will be an instance of `net.sf.saxon.om.NodeInfo`, and so on.

If the return type is `XPathConstants.NODESET`, the result will in general be a Java `List` containing node objects in the selected object model. It may also contain non-node objects if that's what the XPath expression returned. As a special case, if the supplied context node is a DOM node, and if the results are all DOM nodes, then they will be returned in the form of a DOM `NodeList` object.

Saxon does not recognize additional values for the return type other than the values defined in JAXP. If you want to return a different result type, for example a list of integers or a date, then it is probably best not to use this API; if you must, however, then use one of the methods in which the result type is unspecified. If any conversions are necessary, do them within the XPath expression itself, using casts or constructor functions. The Java object that is returned will be a representation of the XPath value, converted in the same way as arguments to a extension functions.

Additional Saxon methods

Saxon's implementation of `XPathExpression` (namely `net.sf.saxon.xpath.XPathExpressionImpl [Javadoc: net.sf.saxon.xpath.XPathExpressionImpl]`) provides additional methods for evaluating the XPath expression. In particular the `rawIterator()` method with no arguments returns a Saxon `SequenceIterator [Javadoc: net.sf.saxon.om.SequenceIterator]` which allows the application to process the results of any XPath expression, with no conversion: all values will be represented using a native Saxon class, for example a node will be represented as a `NodeInfo` and a QName as a `QNameValue [Javadoc: net.sf.saxon.value.QNameValue]`. The `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` interface is described in the next section.

You can call methods directly on the `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` object to get information about a node: for example `getDisplayName()` gets the name of the node in a form suitable for display, and `getStringValue()` gets the string value of the node, as defined in the XPath data model. You can also use the node as the context node for evaluation of subsequent expressions.

Calling JAXP XPath extension functions

The JAXP XPath interface includes an interface `FunctionResolver` which can be used to bind a function call appearing in the XPath expression to a user-written Java implementation of the interface `java.xml.xpath.XPathFunction`. The form in which parameters are passed to such a function, and the form in which it returns its results, are not precisely defined in the JAXP specification, so this section fills the gap. Note that the calling conventions are likely to differ from those used by other products.

The extension function is called by invoking the method `XPathFunction.evaluate()`, which takes a list of arguments, and returns the function result.

The arguments are therefore supplied as a list. Each item in this list represents one argument. The argument value is represented as follows:

1. if the value of the argument is a singleton item, it will be passed as the "natural Java equivalent" of the item's type. For example, a double will be passed as an instance of `java.lang.Double`, a string as an instance of `java.lang.String`. An untyped atomic value is treated as a string. An `xs:integer` (even if it belongs to a subtype such as `xs:short`) is converted to a `java.lang.BigInteger`. The more specialized XML Schema primitive types such as `xs:hexBinary` and `xs:duration` are passed in their native Saxon representation (a subclass of `net.sf.saxon.value.AtomicValue [Javadoc: net.sf.saxon.value.AtomicValue]`). A node will be passed as an instance of `net.sf.saxon.NodeInfo [Javadoc: net.sf.saxon.NodeInfo]`, unless it wraps a foreign node (e.g. a DOM or JDOM node) in which case the foreign node is passed.
2. if the value is a sequence of any length other than one (including zero), then the value that is passed is a List, where each item in the list is converted as described in rule (1).

If the return value conforms to the above conventions, then it will be accepted. However Saxon will also accept a wide range of other return values, including of course a `List` containing one item.

The NodeInfo interface

The `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` object represents a node of an XML document. It has a subclass `DocumentInfo [Javadoc: net.sf.saxon.om.DocumentInfo]` to represent the root node, but all other nodes are represented by `NodeInfo` itself. These follow the XPath data model closely.

The `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` object provides the application with information about the node. The most commonly used methods include:

Table 9.2.

getNodeKind()	gets a short identifying the node type (for example, element or attribute). The values are consistent with those used in the DOM, and are referenced by constants in the class `net.sf.saxon.type.Type [Javadoc: net.sf.saxon.type.Type]`
getDisplayName(), getLocalPart(), getPrefix(), getURI()	These methods get the name of the element, or its various parts. The `getDisplayName()` method returns the QName as used in the original source XML.
getAttributeValue()	get the value of a specified attribute, as a String.
getStringValue()	get the string value of a node, as defined in the XPath data model
getTypedValue()	get the typed value of a node, as defined in the XPath 2.0 data model. This is in general a sequence of atomic values, so the result is a `SequenceIterator [Javadoc: net.sf.saxon.om.SequenceIterator]`.
getParent()	get the `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` representing the parent element, (which will be a `DocumentInfo [Javadoc: net.sf.saxon.om.DocumentInfo]` object if this is the outermost element).
iterateAxis()	returns an `SequenceIterator [Javadoc: net.sf.saxon.om.SequenceIterator]` object that can be used to iterate over the nodes on any of the XPath axes. The first argument is an integer identifying the axis; the second is a `NodeTest [Javadoc: net.sf.saxon.pattern.NodeTest]` (a simple form of pattern) which can be used to filter the nodes on the axis. Supply `null` if you want all the nodes on the axis. (For most applications, it is probably simpler to navigate from a node by compiling an XPath expression, and executing it with the correct starting node as the context node).

For other methods, see the JavaDoc documentation.

It is possible (though not easy) to provide your own implementation of the `NodeInfo` interface, perhaps allowing Saxon queries to run directly against some non-XML data source. There are helper methods in the `net.sf.saxon.om.Navigator [Javadoc: net.sf.saxon.om.Navigator]` class that reduce the amount of code you need to write to achieve this. See the implementations that map `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` to DOM, DOM4J, JDOM or XOM to see how it is done.

Chapter 10. Saxon on .NET

Introduction

Saxon is available on both the Java and .NET platforms. This section of the documentation describes aspects of Saxon that are specific to the .NET platform.

The Saxon source code is written in Java. It has been ported to the .NET platform by cross-compiling the bytecode produced by the Java compiler into the IL code used on .NET, and adding various components designed to integrate Saxon fully into the .NET environment. These additions include:

1. Addition of command-line interfaces `Transform`, `Query`, and `Validate`.
2. Addition of a front-end API suitable for calling Saxon from .NET languages such as C#, VB.NET, or ASP.NET
3. Internal changes within Saxon to take advantage of services offered by the .NET Common Language Runtime, for example collation support, URI handling, and XML parsing.

Both the basic and schema-aware versions of Saxon are available on .NET, with the same functionality as the Java versions.

> When Saxon-EE is used on .NET, queries and stylesheets are selectively compiled to IL code for faster execution. Internally, the Saxon compiler generates Java bytecode; the IKVM runtime automatically translates this to IL code when it is loaded. Code generation typically improves the performance of queries by around 25%. The code is transient (in memory): it cannot be saved as a persistent executable.

Installation of Saxon on the .NET platform is described at Installation (.NET).

The commands `Transform`, `Query`, and `Validate` have the same format as their Java counterparts. They are described in the following places:

1. Transform
2. Query
3. Validate

An introduction to the API for Saxon on .NET is provided at Saxon API for .NET; full specifications are available here [../dotnetdoc/index.html].

For information about writing extension functions using .NET languages, see Writing extension functions for .NET.

> Saxon has been successfully run under the Mono environment (Mono is a .NET emulation for non-Microsoft platforms). However, it has not been comprehensively validated under Mono and there may be undiscovered restrictions.

Saxon API for .NET

A new API has been developed providing access to XSLT, XQuery, XPath, and XML Schema processing on the .NET platform. This is available from any .NET-supported language, although the examples are all expressed in C# terms.

This section provides a brief introduction to the structure and concepts of the API. Full specifications are available here [../dotnetdoc/index.html].

A set of example C# programs illustrating various use cases for the API is available in the `samples/cs` directory.

All the classes referred to below are in the namespace `Saxon.Api`, and can be loaded from the assembly `saxonapi.dll`.

The first thing the application needs to do is to create a `Processor`. The `Processor` holds configuration information for Saxon, and shared resources such as the name pool and schema pool. It is possible to run multiple processors concurrently if required, but it is usually more economical for all Saxon processes within a single application to use the same `Processor`.

XSLT, XQuery, and XPath processing all follow the same pattern:

- From the `Processor`, create a Compiler for the appropriate language, using one of the methods `NewXsltCompiler`, `NewXQueryCompiler`, or `NewXPathCompiler`.
- Set any required properties or configuration options on the resulting Compiler object (these establish the static evaluation context), and then call its `Compile` method to create an Executable. The `Compile` methods are overloaded to accept input from a variety of sources.
- The Executable object represents the compiled stylesheet, query, or XPath expression. It can be evaluated as often as required, in the same thread or in different threads. The first stage in this evaluation is to call the `Load` method on the Executable. The resulting loaded object is an `XsltTransformer`, `XQueryEvaluator`, or `XPathSelector` depending on the language in use.
- Properties and configuration methods can then be set on the loaded object to establish the dynamic evaluation context, and the real processing is then finally invoked using another method: this may be `Run` in the case of XSLT or XQuery where the output is a newly constructed XML document, or `Evaluate`, `EvaluateSingle`, or `GetEnumerator` in the case of XQuery and XPath where the output is an arbitrary sequence.

The API includes a number of classes that reflect the XSLT/XQuery/XPath data model (XDM). These are as follows:

1. `XdmValue`: an XPath value. This is in general a sequence, whose items are nodes or atomic values. You can supply an `XdmValue` as the value of a stylesheet or query parameter, and receive an `XdmValue` as the result of evaluating a query or XPath expression.
2. `XdmItem`: an XPath item. This is a subtype of `XdmValue`, since any item can be treated as a sequence of length one. You can call `GetEnumeration` on an `XdmValue` object to iterate over the items in the sequence.
3. `XdmNode`: a node. This object provides access to most of the properties of nodes defined in the XDM model: the node kind, the string value, the name, the typed value, the base URI. It also provides a method `EnumerateAxis` which allows you to find related nodes using any of the 13 XPath axes. For convenience, the `OuterXml` property provides a simple way to serialize the node.
4. `XdmAtomicValue`: an atomic value, as defined in the XDM model. You can construct an atomic value directly from common objects such as an integer, a string, a double, or a Uri; or you can construct one by specifying a string containing the lexical representation, and a QName identifying the required type.

The `Processor` provides a method `NewDocumentBuilder` which, as the name implies, returns a `DocumentBuilder`. This may be used to construct a document (specifically, an `XdmNode`) from a variety of sources. The input can come from raw lexical XML by specifying a `Stream` or a `Uri`, or it may come from a DOM document built using the Microsoft XML parser by specifying an `XmlNode`, or it may be supplied programmatically by nominating an `XmlReader`. Various processing options can be set as properties of the `DocumentBuilder`: these determine, for example, how whitespace is handled and whether schema validation is performed. The resulting document can be used as the input to a transformation, a query, or an XPath expression. It might also contain a stylesheet or a schema which can then be used as input to the `XsltCompiler` or the `SchemaManager`.

The `SchemaManager` exists only in the `Saxon-EE` product, and its job is to compile schema documents and to maintain a cache containing the compiled schemas. It thus contains method to compile schemas from a variety of document sources. It also contains a factory method `NewSchemaValidator`, which returns a `SchemaValidator`. The `SchemaValidator`, in turn, is used to validate a source document against the set of schema definitions held in the `SchemaManager`'s cache.

Finally, the API offers a class `XmlDestination` to define the possible ways of handling a document constructed as the output of a transformation, query, or validation episode. Various subtypes of `XdmDestination` allow such results to be serialized as XML (using either the Saxon serializer or an `XmlTextWriter`), or to be materialized as a Saxon `XdmNode` or as a DOM `XmlNode`.

These classes are designed to be combined in arbitrary ways. For example, you might run an XQuery whose result is a sequence of newly-constructed document nodes. You could then iterate over these nodes, and for each one, apply an XSLT transformation whose result is then serialized.

There are several places where the classes in the `Saxon.Api` package provide an "escape hatch" into the underlying implementation classes. These are provided for the benefit of applications that for some reason need to mix use of the .NET API with the established Java API. The underlying implementation classes are documented in Java terms and use Java calling conventions, but this does not stop them being used from any .NET language: you may need to consult the IKVM [http://www.ikvm.net] documentation for details of the mappings. The places where such escape hatches are provided are shown below:

Table 10.1.

Interface class	Property	Implementation class
Saxon.Api.Processor	Implementation	net.sf.saxon.Configuration
Saxon.Api.XdmNode	Implementation	net.sf.saxon.om.NodeInfo
Saxon.Api.XsltTransformer	Implementation	net.sf.saxon.Controller
Saxon.Api.XQueryCompiler	Implementation	net.sf.saxon.query.StaticQueryContext

XML Parsing in .NET

When you run Saxon on the .NET platform, there are two XML parsers available: the `System.Xml` parser supplied with the .NET platform, and the Xerces Apache parser, which is supplied as part of the Saxon product.

Saxon generally uses the Xerces parser by preference. However, you can force Saxon to use the OpenJDK parser by calling `processor.SetProperty("http://saxon.sf.net/feature/preferJaxpParser", "false")`. Or from the command line, set the option `--preferJaxpParser:off`.

Note that the Microsoft parser does not notify ID or IDREF values from the DTD, or expand attributes with fixed or default values, unless DTD validation is requested. This can be requested using the -v option on the command line, or via the API. (See the `DtdValidation` property of the `DocumentBuilder` class)

Unparsed entities are not notified to Saxon by the Microsoft parser. The XSLT functions `unparsed-entity-system-id()` and `unparsed-entity-public-id()`will therefore not work.

Support for OASIS catalogs is provided at the command-line level by means of the command-line option `-catalog:filename`. To enable this to work, the Apache catalog resolver is integrated in the Saxon DLL. This works only with the JAXP (Xerces) parser. To use catalogs when running an application using the Saxon API, the interface is not quite so convenient. Call the static method `net.sf.saxon.trans.XmlCatalogResolver.setCatalog()` (found in `saxon9he.dll`) with three arguments: the filename of the catalog file, the Saxon Configuration object, and a boolean indicating whether tracing is required.

Chapter 11. Extensibility

Introduction

This section describes how to extend the capability of Saxon XSLT stylesheets and XQuery queries by adding extension functions and other user hooks.

The first two columns of the table below indicate which sections of this page are applicable to XSLT and which are applicable to XQuery. The next three columns indicate which Saxon editions the information applies to.

Table 11.1.

XSLT	XQuery	HE	PE	EE	
§	§	§	§	§	Integrated extension functions
§	§		§	§	Reflexive extension functions (Java)
§	§		§	§	Reflexive extension functions (.NET)
§			§	§	Writing XSLT extension instructions
§	§	§	§	§	Customizing serialization
§	§	§	§	§	Implementing a collating sequence
§	§	§	§	§	Implementing localized numbers and dates
§	§	§	§	§	Writing a URI Resolver for input files
§		§	§	§	Writing a URI Resolver for output files

Integrated extension functions

There are two ways of writing extension functions. The traditional way is to map the name of the function to a Java or .NET method: specifically, the namespace URI of the function name maps to the Java or .NET class name, and the local part of the function name maps to the Java or .NET method name. These are known as extension functions, and are described in later pages.

From Saxon 9.2, this technique is supplemented by a new mechanism, referred to as .

There are several advantages in this approach:

- You can choose any function name you like, in any namespace.
- The function signature is made explicit, in terms of XPath types for the arguments and result.
- There is no ambiguity about which of several candidate Java or .NET methods is invoked.
- There is less scope for configuration problems involving dynamic loading of named classes.
- All conversions from XPath values to Java or .NET values are entirely under user control.
- The function implementation is activated at compile time, allowing it to perform optimization based on the expressions supplied as arguments, or to save parts of the static context that it needs, such as the static base URI or the current namespace context.
- The function declares its properties, for example whether it uses the context item and whether it has side-effects, making it easier for the optimizer to manipulate the function call intelligently.
- Integrated extension functions are more secure, because the function must be explicitly registered by the calling application before it can be called.

There are two ways of writing integrated extension functions: the simple API and the full API. The full API is available for both the Java and .NET platforms; the simple API is available only for Java. The resulting combinations are described on the following pages.

- Java extension functions: simple interface
- Java extension functions: full interface
- .NET extension functions

Java extension functions: simple interface

The simple API for integrated Java extension functions is available via the s9api class `ExtensionFunction [Javadoc: net.sf.saxon.s9api.ExtensionFunction]`. Here is an example that defines an extension function to calculate square roots, registers this extension function with the s9api `Processor`, and then invokes it from an XPath expression:

```
Processor proc = new Processor(false);
ExtensionFunction sqrt = new ExtensionFunction() {
    public QName getName() {
        return new QName("http://math.com/", "sqrt");
    }

    public SequenceType getResultType() {
        return SequenceType.makeSequenceType(
            ItemType.DOUBLE, OccurrenceIndicator.ONE
        );
    }

    public net.sf.saxon.s9api.SequenceType[] getArgumentTypes() {
        return new SequenceType[]{
            SequenceType.makeSequenceType(
                ItemType.DOUBLE, OccurrenceIndicator.ONE)};
    }

    public XdmValue call(XdmValue[] arguments) throws SaxonApiExcep
        double arg = ((XdmAtomicValue)arguments[0].itemAt(0)).getDou
        double result = Math.sqrt(arg);
        return new XdmAtomicValue(result);
```

```
            }
        };

        proc.registerExtensionFunction(sqrt);
        XPathCompiler comp = proc.newXPathCompiler();
        comp.declareNamespace("mf", "http://math.com/");
        comp.declareVariable(new QName("arg"));
        XPathExecutable exp = comp.compile("mf:sqrt($arg)");
        XPathSelector ev = exp.load();
        ev.setVariable(new QName("arg"), new XdmAtomicValue(2.0));
        XdmValue val = ev.evaluate();
        String result = val.toString();
```

Full details of the interface are defined in the Javadoc for class `ExtensionFunction [Javadoc: net.sf.saxon.s9api.ExtensionFunction]`.

The main restrictions of the simple interface are (a) that the extension function has no access to static or dynamic context information, and (b) that it does not support pipelined evaluation of the arguments or result. To avoid these restrictions, use the full interface described on the next page.

Java extension functions: full interface

With this approach, each extension function is implemented as a pair of Java classes. The first class, the `ExtensionFunctionDefinition [Javadoc: net.sf.saxon.lib.ExtensionFunctionDefinition]`, provides general static information about the extension function (including its name, arity, and the types of its arguments and result). The second class, an `ExtensionFunctionCall [Javadoc: net.sf.saxon.lib.ExtensionFunctionCall]`, represents a specific call on the extension function, and includes the `call()` method that Saxon invokes to evaluate the function.

When a stylesheet or query uses integrated extension functions and is run from the command line, the classes that implement these extension functions must be registered with the `Configuration [Javadoc: net.sf.saxon.Configuration]`. On Saxon-PE and Saxon-EE this can conveniently be done by declaring them in a configuration file. It can also be achieved (on all editions including Saxon-HE) by subclassing `net.sf.saxon.Transform [Javadoc: net.sf.saxon.Transform]` or `net.sf.saxon.Query [Javadoc: net.sf.saxon.Query]`, overriding the method `applyLocalOptions()` so that it makes the appropriate calls on `config.registerExtensionFunction()`; or it can be done in a user-defined class that implements the interface `net.sf.saxon.Initializer [Javadoc: net.sf.saxon.lib.Initializer]`, and that is nominated on the command line using the `-init` option.

The arguments passed in a call to an integrated extension function are type-checked against the declared types in the same way as for any other XPath function call, including the standard conversions such as atomization and numeric promotion. The return value is checked against the declared return type but is not converted: it is the responsibility of the function implementation to return a value of the correct type.

Here is an example extension written to the Java version of this interface. It takes two integer arguments and performs a "shift left" operation, shifting the first argument by the number of bit-positions indicated in the second argument:

```
private static class ShiftLeft extends ExtensionFunctionDefinition {
        @Override
        public StructuredQName getFunctionQName() {
            return new StructuredQName("eg", "http://example.com/saxon-extensio
        }

        @Override
        public SequenceType[] getArgumentTypes() {
            return new SequenceType[] {SequenceType.SINGLE_INTEGER, SequenceTyp
        }

        @Override
        public SequenceType getResultType(SequenceType[] suppliedArgumentTypes)
            return SequenceType.SINGLE_INTEGER;
        }

        @Override
        public ExtensionFunctionCall makeCallExpression() {
            return new ExtensionFunctionCall() {
                public SequenceIterator call(SequenceIterator[] arguments, XPat
                    long v0 = ((IntegerValue)arguments[0].next()).longValue();
                    long v1 = ((IntegerValue)arguments[1].next()).longValue();
                    long result = v0<<v1;
                    return Value.asIterator(Int64Value.makeIntegerValue(result)
                }
            };
        }
    }
```

The extension must be registered with the configuration:

```
configuration.registerExtensionFunction(new ShiftLeft())
```

and it can then be called like this:

```
declare namespace eg="http://example.com/saxon-extension";
          for $i in 1 to 10 return eg:shift-left(2, $i)"
```

The methods that must be implemented (or that may be implemented) by an integrated extension function are listed in the table below. Further details are in the Javadoc.

First, the `ExtensionFunctionDefinition` `[Javadoc: net.sf.saxon.lib.ExtensionFunctionDefinition]` class:

Table 11.2.

getFunctionQName	Returns the name of the function, as a QName (represented by the Saxon class `StructuredQName`). Like all other functions, integrated extension functions must be in a namespace. The prefix part of the QName is immaterial.
getMinumumNumberOfArguments	Indicates the minimum number of arguments that must be supplied in a call to the function. A call with fewer arguments than this will be rejected as a static error.
getMaximumNumberOfArguments	Indicates the maximum number of arguments that must be supplied in a call to the function. A call with more arguments than this will be rejected as a static error.
getArgumentTypes	Returns the static type of each argument to the function, as an array with one

	member per argument. The type is returned as an instance of the Saxon class `net.sf.saxon.type.SequenceType`. Some of the more commonly-used types are represented by static constants in the `SequenceType` class. If there are fewer members in the array than there are arguments in the function call, Saxon assumes that all arguments have the same type as the last one that is explicitly declared; this allows for function with a variable number of arguments, such as `concat()`.
getResultType	Returns the static type of the result of the function. The actual result returned at runtime will be checked against this declared type, but no conversion takes place. Like the argument types, the result type is returned as an instance of `net.sf.saxon.type.SequenceType`.When Saxon calls this method, it supplies an array containing the inferred static types of the actual arguments to the function call. The implementation can use this information to return a more precise result, for example in cases where the value returned by the function is of the same type as the value supplied in the first argument.
trustResultType	This method normally returns `false`. It can return `true` if the implementor of the extension function is confident that no run-time checking of the function result is needed; that is, if the method is guaranteed to return a value of the declared result type.
dependsOnFocus	This method must return true if the implementation of the function accesses the context item, context position, or context size from the dynamic evaluation context. The method does not need to be implemented otherwise, as its default value is false.
hasSideEffects	This method should be implemented, and return true, if the function has side-effects of any kind, including constructing new nodes if the identity of the nodes is signficant. When this method returns true, Saxon will try to avoid moving the function call out of loops or otherwise rearranging the sequence of calls. However, functions with side-effects are still discouraged, because the optimizer cannot always detect their presence if they are deeply nested within other calls.
makeCallExpression	This method must be implemented; it is called at compile time when a call to this extension function is identified, to create an instance of the relevant `ExtensionFunctionCall` object to hold details of the function call expression.

The methods defined on the second object, the `ExtensionFunctionCall [Javadoc: net.sf.saxon.lib.ExtensionFunctionCall]`, are:

Table 11.3.

supplyStaticContext	Saxon calls this method fairly early on during the compilation process to supply details of the static context in which the function call appears. The method may in some circumstances be called more than once; it will always be called at least once. As well as the static context information itself, the expressions supplied as arguments are also made available. If evaluation of the function depends on information in the static context, this information should be copied into private variables for use at run-time.
rewrite	Saxon calls this method at a fairly late stage during compilation to give the implementation the opportunity to optimize itself, for example by performing partial evaluation of intermediate results, or if all the arguments are compile-time constants (instances of `net.sf.saxon.expr.Literal`) even by early evaluation of the entire function call. The method can return any `Expression` (which includes the option of returning a `Literal` to represent the final result); the returned expression will then be evaluated at run-time in place of the original. It is entirely the responsibility of the implementation to ensure that the substitute expression is equivalent in every way, including the type of its result.
copyLocalData	Saxon occasionally needs to make a copy of an expression tree. When it copies an integrated function call it will invoke this method, which is responsible for ensuring that any local data maintained within the function call objects is correctly copied.
call	Saxon calls this method at run-time to evaluate the function.The value of each argument is supplied in the form of a `SequenceIterator`, that is, an iterator over the items in the sequence that make up the value of the argument. This may use lazy evaluation, which means that a dynamic error can occur when reading the next item from the SequenceIterator; it also means that if the implementation does not require all the items from the value of one of the arguments, they will not necessarily be evaluated at all (it is good practice to call the close() method on the iterator if it is not read to completion.) The implementation delivers the result also in the form of a `SequenceIterator`, which in turn means that the result may be subject to delayed evaluation: the calling code will only access items in the result as they are required, and may not always read the result to completion. To return a singleton result, use the class `net.sf.saxon.om.SingletonIterator`;

	to return an empty sequence, return the unique instance of `net.sf.saxon.om.EmptyIterator`.

Having written an integrated extension function, it must be registered with Saxon so that calls on the function are recognized by the parser. This is done using the `registerExtensionFunction` method available on the `Configuration [Javadoc: net.sf.saxon.Configuration]` class, and also on the s9api `Processor [Javadoc: net.sf.saxon.s9api.Processor]` class. It can also be registered via an entry in the configuration file. The function can be given any name, although names in the `fn:`, `xs:`, and `saxon:` namespaces are strongly discouraged and may not work.

It is also possible to register integrated extension functions under XQJ: this is done by locating the `Configuration` that underpins the `XQDataSource` or `XQConnection` by casting it to the Saxon implementation class (`SaxonXQDataSource` or `SaxonXQConnection`) and calling `getConfiguration()`.

.NET extension functions

The API for integrated .NET extension functions is available via the .NET classes `ExtensionFunctionDefinition` and `ExtensionFunctionCall`, both defined in the Saxon.API module. Here is a simple example that defines an extension function to calculate square roots, registers this extension function with the s9api `Processor`, and then invokes it from an XPath expression:

```
public class Sqrt : ExtensionFunctionDefinition {

    public override QName FunctionName {
        get { return new QName("http://math.com/", "sqrt") };
    }

    public override int MinimumNumberOfArguments {
        get { return 1 };
    }

    public override int MaximumNumberOfArguments {
        get { return 1 };
    }

    public override XdmSequenceType[] ArgumentTypes {
        get { return new XdmSequenceType[] {
                new XdmSequenceType(XdmAtomicType.BuiltInAtomicType
              }
            }
    }

    public override XdmSequenceType ResultType(XdmSequenceType[] Ar
        return new XdmSequenceType(XdmAtomicType.BuiltInAtomicType(
    }

    public override bool TrustResultType {
        get { return true };
    }

    public override ExtensionFunctionCall MakeFunctionCall() {
        return new SqrtCall();
    }
}

public class SqrtCall : ExtensionFunctionCall {

    public override IXdmEnumerator Call(IXdmEnumerator[] arguments,
        Boolean exists = arguments[0].MoveNext();
        if (exists) {
            XdmAtomicValue arg = (XdmAtomicValue)arguments[0].Curre
            double val = (double)arg.Value;
            double sqrt = System.Math.Sqrt(val);
            XdmAtomicValue result = new XdmAtomicValue(sqrt);
            return (IxdmEnumerator)result.GetEnumerator();
        } else {
            return EmptyEnumerator.INSTANCE;
        }
    }
}

Processor proc = new Processor();
proc.RegisterExtensionFunction(new Sqrt());
XPathCompiler xpc = proc.NewXPathCompiler();
xpc.DeclareNamespace("mf", "http://math.com/");
XdmItem result = xpc.EvaluateSingle("mf:sqrt(2)", null);
Console.WriteLine("Square root of 2 is " + result);
```

Full details of the interface are defined in the .NET API documentation.

Writing reflexive extension functions in Java

This section applies to Saxon-PE and Saxon-EE only

Reflexive extension functions written in Java map the namespace of the XPath function name to a Java fully-qualified class name, and the local name of the function to a Java method or field name.

Java extension functions can also be used when you are running on the .NET platform, provided the class implementing the function is a standard class in the OpenJDK class library (which covers nearly all the classes defined in the JDK). In other cases, you should compile the Java code into a .NET assembly using the IKVMC compiler, in which case it behaves in the same way as an extension function written in any other .NET language and compiled into CIL: see Writing extension functions under .NET

An extension function is invoked using a name such as `prefix:localname()`. The prefix must be the prefix associated with a namespace declaration that is in scope. The namespace URI is used to identify a Java class, and the local name is used to identify a method, field, or constructor within the class.

There are a number of extension functions supplied with the Saxon product: for details, see Extensions. The source code of these methods, which in most cases is extremely simple, can be used as an example for writing other user extension functions. It is found in class `net.sf.saxon.functions.Extensions`.

Identifying the Java Class

There are various ways a mapping from URIs to Java classes can be established. The simplest is to use a URI that identifies the Java class explicitly. The namespace URI should be "java:" followed by the fully-qualified class name (for example `xmlns:date="java:java.util.Date"`). The class must be on the classpath.

> For compatibility with other products and previous Saxon releases, Saxon also supports certain other formats of URI. The URI may be a string containing a "/", in which the fully-qualified class name appears after the final "/". (for example `xmlns:date="http://www.jclark.com/xt/java/java.util.Date"`). The part of the URI before the final "/" is immaterial. The format `xmlns:date="java.util.Date"` is also supported. To permit this extended syntax for namespaces, you need to set a property on the Configuration: `config.setConfigurationProperty(FeatureKeys.ALLOW_OLD_JAVA_URI_FORMAT, true)`. The flag can also be set in the configuration file.

The Saxon namespace URI `http://saxon.sf.net/` is recognised as a special case. In most cases it causes the function to be loaded from the class `net.sf.saxon.functions.Extensions` but in a few cases, such as `saxon:evaluate`, the function is recognized by the compiler as if it were a built-in function. The various EXSLT namespaces are also recognized specially.

In XSLT, the system function `function-available(String name)` returns true if there appears to be a method available with the right name. The function also has an optional second argument to test whether there is a method with the appropriate number of arguments. However, it is not possible to test whether the arguments are of appropriate types. If the function name is "new" it returns true so long as the class is not an abstract class or interface, and so long as it has at least one constructor.

Identifying the Java constructor, method, or field

The local name used in the XPath function call determines which constructor, method, or field of the Java class is invoked. This decision (called binding) is always made at the time the XPath expression is compiled. (In previous Saxon releases it was sometimes delayed until the actual argument values were known at run-time).

- If the local name is `new`, a constructor is invoked. If several constructors are available, the one that is chosen is based on the number and types of the supplied arguments.

- In other cases, the system looks for a matching method or field.

  Firstly, the name must match, after converting hyphenated names to camelCase, which is done by removing any hyphen in the XPath name and forcing the immediately following character to upper case. For example the XPath function call `to-string()` matches the Java method `toString()`; but the function call can also be written as `toString()` if you prefer.

  Secondly, the number of arguments must match, after taking into account that (a) if the Java method expects a first argument of class `net.sf.saxon.expr.XPathContext` then this will be supplied automatically by the system and does not correspond to any explicit argument in the XPath function call, and (b) when invoking an instance-level (non-static) method or field, the XPath function call must supply an extra first argument, which identifies the target object for the invocation.

  A public field in a class is treated as if it were a zero-argument method, so public static fields can be accessed in the same way as public static methods, and public instance-level fields in the same way as instance-level methods.

- If there are several matching methods, the one that is chosen is determined by comparing the static types of the supplied arguments with the required types in the method signature. See Choosing among overloaded methods.
- Choosing among overloaded methods
- Calling Static Methods in a Java Class
- Calling Java Constructors
- Calling Java Instance-Level Methods

Choosing among overloaded methods

If there is no method with the required name and number of parameters, Saxon reports a compile-time error.

If there is only one method with the required name and number of parameters, then Saxon chooses it, and goes on to the next stage, which allocates converters for the supplied arguments.

If there are several methods in the class that match the localname, and that have the correct number of arguments, then the system attempts to find the one that is the best fit to the types of the supplied arguments: for example if the call is `f(1,2)` then a method with two `int` arguments will be preferred to one with two `float` arguments. The rules for deciding between methods are quite complex. Essentially, for each candidate method, Saxon calculates the "distance" between the types of the supplied arguments and the Java class of the corresponding method in the method's signature, using a set of tables. For example, the distance between the XPath data type `xs:integer` and the Java class `long` is very small, while the distance between an XPath `xs:integer` and a Java `Object` is much larger. If there is one candidate method where the distances of all arguments are less-than-or-equal-to the distances computed for other candidate methods, and the distance of at least one argument is smaller, then that method is chosen.

If there are several methods with the same name and the correct number of arguments, but none is preferable to the others under these rules, an error is reported: the message indicates that there is more than one method that matches the function call.

This binding is carried out statically, using the static types of the supplied arguments, not the dynamic types obtained when the arguments are evaluated. If there is insufficient static type information to distinguish the candidate methods, an error is reported. You can supply additional type information using the `treat as` expression, or by casting. Often it is enough simply to declare the types of the variables used as arguments to the function call.

The distances are calculated using the following rules (in order).

- If the required type is `Object`, the distance is 100.
- If the required type is one of the following Saxon-specific classes (or is a superclass of that class), then the distance is as given:
  - `net.sf.saxon.om.SequenceIterator [Javadoc: net.sf.saxon.om.SequenceIterator]`: 26
  - `net.sf.saxon.om.ValueRepresentation [Javadoc: net.sf.saxon.om.ValueRepresentation]`: 25
  - `net.sf.saxon.value.Value [Javadoc: net.sf.saxon.value.Value]`: 24
  - `net.sf.saxon.om.Item [Javadoc: net.sf.saxon.om.Item]`: 23
  - `net.sf.saxon.om.NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]`: 22

  - `net.sf.saxon.om.DocumentInfo` [Javadoc: `net.sf.saxon.om.DocumentInfo`]: 21

  - `net.sf.saxon.value.AtomicValue` [Javadoc: `net.sf.saxon.value.AtomicValue`], 20

- If the static type of the supplied argument allows more than one item, the distance is the first one of the following that applies:

  - If the method expects `java.lang.Collection` or a class that implements `Collection`: 30

  - If the method expects an array: 31

  - In all other cases: 80

  Note that the item type of the supplied value plays no role in choosing the method to call, even though it could potentially be used to disambiguate overloaded methods when arrays or parameterized collection types are used.

- Otherwise (the static type allows only one item):

  - If the static type of the supplied value matches `node()`: 80

  - If the static type of the supplied value is a wrapped Java object, then 10 if the class of the object matches the required Java class, else -1 (meaning this method is not a candidate)

  - Otherwise, the value given in the table of atomic types below.

The distances for atomic types are given below. If there is no entry for the combination of supplied type and required type, the method is removed from consideration. For unboxed types (int, float, etc) the distance is always one less than the corresponding boxed type (java.lang.Integer, java.lang.Float).

Table 11.4.

xs:boolean	, Boolean
xs:dateTime	, Date
xs:date	, Date
xs:decimal	, BigDecimal, Double, Float
xs:double	, Double
xs:duration	
xs:float	, Float, Double
xs:integer	, BigInteger, BigDecimal, Long, Integer, Double, Float
xs:short	, BigInteger, BigDecimal, Long, Integer, Short, Double, Float
xs:byte	, BigInteger, BigDecimal, Long, Integer, Short, Byte, Double, Float
xs:string	, (String, CharSequence)
xs:anyURI	, java.net.URI, java.net.URL, (String, CharSequence)
xs:QName	, javax.xml.namespace.QName

Saxon tries to select the appropriate method based on the of the arguments to the function call. If there are several candidate methods, and there is insufficient information available to decide which is most appropriate, an error is reported. The remedy is to cast the arguments to a more specific type.

A required type of one of the Java primitive types such as `int` or `bool` is treated as equivalent to the corresponding boxed type (`Integer` or `Boolean`), except that with the boxed types, an empty sequence can be supplied in the function call and is translated to a Java null value as the actual argument.

The fact that a particular method is chosen as the target does not give a guarantee that conversion of the arguments will succeed at run-time. This is particularly true with methods that expect a node in an external object model such as DOM or XOM.

Calling Static Methods in a Java Class

can be called directly.

For example (in XSLT):

```
<xsl:value-of select="math:sqrt($arg)"
   xmlns:math="java:java.lang.Math"/>
```

This will invoke the static method `java.lang.Math#sqrt()`, applying it to the value of the variable `$arg`, and copying the value of the square root of `$arg` to the result tree.

Similarly (in XQuery):

```
<a xmlns:double="java:java.lang.Double">
                           {double:MAX_VALUE()} </a>
```

This will output the value of the static field `java.lang.Double#MAX_VALUE`. (In practice, it is better to declare the namespace in the query prolog, because it will then not be copied to the result tree.)

A static Java method called as an extension function may have an extra first argument of class `net.sf.saxon.expr.XPathContext` `[Javadoc: net.sf.saxon.expr.XPathContext]`. This argument is not supplied by the calling XPath or XQuery code, but by Saxon itself. The `XPathContext` object provides methods to access many internal Saxon resources, the most useful being `getContextItem()` which returns the context item from the dynamic context. The XPathContext object is available with static or instance-level methods, but not with constructors.

The following example shows a function that obtains the line number of the context node (this is actually a built-in Saxon extension):

```
    /**
    * Return the line number of the context node.
    */
    public static int lineNumber(XPathContext c) {
        Item item = c.getCurrentIterator().current();
        if (item instanceof NodeInfo) {
            return ((NodeInfo)item).getLineNumber();
        } else {
            return -1;
        }
    }
```

If this method appears in class `com.example.code.NodeData`, then it can be accessed using the following code in XSLT:

```
<xsl:value-of select="nd:line-number()"
    xmlns:nd="java:com.example.code.NodeData"/>
```

or the following in XQuery:

```
<line xmlns:nd="java:com.example.code.NodeData">
    { nd:line-number() }
</line>
```

Calling Java Constructors

are called by using the function named `new()`. If there are several constructors, then again the system tries to find the one that is the best fit, according to the types of the supplied arguments. The result of calling `new()` is an XPath value whose type is denoted by a QName whose local name is the actual Java class (for example `java.sql.Connection` or `java.util.List`) and whose namespace URI is `http://saxon.sf.net/java-type` (conventional prefix `class`). Any '$' characters in the class name are replaced by '-' characters in the QName. The only things that can be done with a wrapped Java Object are to assign it to a variable, to pass it to an extension function, and to convert it to a string, number, or boolean, using the rules given below.

The use of external object types in namespace `http://saxon.sf.net/java-type` reflects the Java type hierarchy. For example, if a variable is declared to accept a type of `jt:java.util.List`, then a value of type `jt:java.util.ArrayList` will be accepted, but a value of type `jt:java.util.HashMap` will not.

Calling Java Instance-Level Methods

(that is, non-static methods) are called by supplying an extra first argument of type Java Object which is the object on which the method is to be invoked. A Java Object is usually created by calling an extension function (e.g. a constructor) that returns an object; it may also be passed to the style sheet as the value of a global parameter. Matching of method names is done as for static methods. If there are several methods in the class that match the localname, the system again tries to find the one that is the best fit, according to the types of the supplied arguments.

For example, the following XSLT stylesheet prints the date and time. (In XSLT 2.0, of course, this no longer requires extension functions, but the example is still valid.)

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:date="java:java.util.Date">

<xsl:template match="/">
  <html>
    <xsl:if test="function-available('date:to-string') and
                          function-available('date:new')">
      <p><xsl:value-of select="date:to-string(date:new())"/></p>
    </xsl:if>
  </html>
</xsl:template>

</xsl:stylesheet>
```

The equivalent in XQuery is:

```
declare namespace date="java:java.util.Date";
<p>{date:to-string(date:new())}</p>
```

As with static methods, an instance-level Java method called as an extension function may have an extra first argument of class `net.sf.saxon.expr.XPathContext [Javadoc:`

`net.sf.saxon.expr.XPathContext]`. This argument is not supplied by the calling XPath or XQuery code, but by Saxon itself. The `XPathContext` object provides methods to access many internal Saxon resources, the most useful being `getContextItem()` which returns the context item from the dynamic context. The XPathContext object is not available with constructors.

If any exceptions are thrown by the method, or if a matching method cannot be found, processing of the stylesheet will be abandoned. If the tracing option has been set (-T) on the command line, a full stack trace will be output. The exception will be wrapped in a `TransformerException` and passed to any user-specified `ErrorListener` object, so the `ErrorListener` can also produce extra diagnostics.

Converting Arguments to Java Extension Functions

This section describes how XPath values supplied in a call to a Java extension function are converted to Java values.

There are three phases of decision-making:

- First, Saxon decides which method to call. If there are several methods with the same name, this takes into account the static types of the supplied arguments and the Java types expected by the method signature. However, this process does not influence how the arguments are subsequently converted.
- Saxon allocates a converter, wherever possible at compile time, based on the static type of the supplied argument and the Java type expected by the method.
- At run-time the converter performs the conversion from the supplied value to the required type. Some converters will make further decisions based on run-time types at this stage.

These stages are described further in the following pages.

- Converting Method Arguments - General Rules
- Converting Atomic Values
- Converting Nodes
- Converting Wrapped Java Objects

Converting Method Arguments - General Rules

Having decided which method to call, Saxon has to convert the supplied XPath argument values to the Java objects required by this method.

If the expected type is `Object`, the supplied value must either be a singleton, or an empty sequence. If it is an empty sequence, null will be passed. If it is a singleton node, an instance of `net.sf.saxon.om.NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` will be passed. If it is a wrapped Java object, that Java object will be passed. If it is a singleton atomic value, the value will be converted to the nearest equivalent Java object: for example an `xs:boolean` becomes `java.lang.Boolean`, an `xs:string` becomes `java.lang.String`, and so on. An untyped atomic value is treated as a string. An `xs:integer` (even if it belongs to a subtype such as `xs:short`) is converted to a Java `BigInteger`. The more specialized XML Schema primitive types such as `xs:hexBinary` and `xs:duration` are passed in their native Saxon representation (a subclass of `net.sf.saxon.value.AtomicValue [Javadoc: net.sf.saxon.value.AtomicValue]`).

If the expected type is one of the Saxon-specific classes (`SequenceIterator [Javadoc: net.sf.saxon.om.SequenceIterator]`, `ValueRepresentation [Javadoc: net.sf.saxon.om.ValueRepresentation]`, `Item [Javadoc:`

`net.sf.saxon.om.Item]`, `Value [Javadoc: net.sf.saxon.value.Value]`, `AtomicValue [Javadoc: net.sf.saxon.value.AtomicValue]`, `SequenceExtent [Javadoc: net.sf.saxon.value.SequenceExtent]`), then the value is passed unchanged. An error occurs if the supplied value contains more than one item and the expected type does not allow this.

If the expected type implements `java.util.Collection`, Saxon attempts to convert each value in the supplied sequence to the most appropriate Java class, following the same rules as when converting a singleton to `java.lang.Object`. This process takes no account of parameterized collection types (such as `List<String>`). If the required collection type accepts an `java.util.ArrayList`, Saxon will create an `ArrayList` to hold the values; otherwise it will attempt to instantiate the required type of collection, which will only work if it is a concrete class with a zero-argument public constructor (so it will fail, for example, if the required type is `java.util.Set`). If an empty sequence is supplied as the argument value, this is converted to an empty Collection.

If the required type is an array, Saxon will attempt to create an array of the required type. This will not always succeed, for example if the array has type `X[]` where X is an interface rather than a concrete class. If it is an array of items or nodes, the nodes in the supplied sequence will be inserted into the array directly; if it is an array of a type such as integer or double, the sequence will first be atomized.

Converting Atomic Values

This section describes the conversions that occur when calling a method that expects an atomic value, such as a String or a boolean.

If the supplied value is a node, then it is atomized.

If the supplied value contains more than item and only a single item is expected, an error is reported. There is no implicit extraction of the first value (as happened in earlier Saxon releases).

If the supplied value is an empty sequence, then a null value is passed. However, if the required type is a primitive Java type such as `int` or `bool`, then passing an empty sequence will result in a type error.

In other cases, the supported conversions are as follows. Italicized names are Saxon-specific classes in package `net.sf.saxon.value`.

Table 11.5.

boolean	, Boolean
dateTime	, Date
date	, Date
decimal	, BigDecimal, Double, Float
double	, Double
duration	
float	, Float, Double
integer, long, int	, BigInteger, BigDecimal, Long, Integer, Double, Float
short	, BigInteger, BigDecimal, Long, Integer, Short, Double, Float
byte	, BigInteger, BigDecimal, Long, Integer, Short, Byte, Double, Float
string	, (String, CharSequence)
anyURI	, java.net.URI, java.net.URL, (String, CharSequence)

QName	, javax.xml.namespace.QName

A required type of one of the Java primitive types such as `int` or `bool` is treated as equivalent to the corresponding boxed type (`Integer` or `Boolean`), except that with the boxed types, an empty sequence can be supplied in the function call and is translated to a Java null value as the actual argument.

Converting Nodes

If the expected type is a generic collection type, or an array of the Saxon class `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]`, or a `Value [Javadoc: net.sf.saxon.om.Value]` or `SequenceIterator [Javadoc: net.sf.saxon.om.SequenceIterator]`, Saxon will pass the nodes supplied in the call in their native Saxon representation, that is, as instances of `net.sf.saxon.om.NodeInfo`.

Saxon recognizes methods that expect nodes in an external object model (DOM, DOM4J, JDOM, or XOM) only if the supporting JAR file is on the classpath (that is, saxon9-dom.jar, saxon9-dom4j.jar, saxon9-jdom.jar, or saxon9-xom.jar). In all four cases, if the XPath node is actually a view of a DOM, DOM4J, JDOM, or XOM node, then the underlying node will be passed to the method. If the XPath node is a text node that maps to a sequence of adjacent text and/or CDATA nodes in the underlying model, the first node in this sequence will be passed to the extension function.

In addition, in the case of DOM only (but only if saxon9-dom.jar is on the classpath), if the XPath node is a view of a DOM node, Saxon will create a DOM wrapper for the native Saxon node, and pass this wrapper. This is also done if the required type is a DOM NodeList. Note that the wrapper is a read-only DOM implementation: any attempt to update nodes through the wrapper interface will throw an exception. A consequence of the way the wrapping works is that it's not safe to rely on object identity when testing node identity - the same node can be represented by more than one Java object. Use the DOM method `isSameNode()` instead.

Converting Wrapped Java Objects

Saxon allows an extension function to return an arbitrary Java object. This will then be wrapped as an XPath item, so that it can be held in a variable and passed subsequently as an argument to another extension function. This second extension function will see the original Java object minus its wrapper, provided it is declared to expect the appropriate Java class.

A wrapped Java object may be converted to another data type as follows.

- It is converted to a string by using its toString() method; if the object is null, the result is the empty string "".
- It is converted to a number by converting it first to a string, and then applying the XPath number() conversion. If it is null, the result is NaN.
- It is converted to a boolean as follows: if it is null, the result is false, otherwise it is converted to a string and the result is true if and only if the string is non-empty.

The type of a wrapped Java object may be declared in a variable declaration or function signature using a type name whose namespace URI is "http://saxon.sf.net/java-type", and whose local name is the fully qualified name of the Java class, with any "$" signs replaced by hyphens. For example, the `sql:connection` extension function returns a value of type `{http://saxon.sf.net/java-type}java.sql.Connection`.

Note that a call on a constructor function (using prefix:new()) always returns a wrapped Java object, regardless of the class. But a call on a static method, instance-level method, or field will return a wrapped Java object only if the result is a class that Saxon does not specifically recognize as one that it can convert to a regular XPath value. Such classes include `String`, `Long`, `Double`, `Date`, `BigInteger`, `URI`, `List` and so on.

Converting the Result of a Java Extension Function

This section explains how the value returned by a Java extension function is converted to an XPath value. The same rules are used in converting a Java object supplied as a parameter to a stylesheet or query.

The result type of the method is converted to an XPath value as follows.

- If the method returns void, the XPath value is an empty sequence.
- If the method returns null, the XPath value is an empty sequence.
- If the method is a constructor, the XPath value is of type "wrapped Java object". The only way of using this is by passing it to another external function, or by converting it to one of the standard XPath data types as described above.
- If the returned value is a Java boolean or Boolean, the XPath result is a boolean.
- If the returned value is a Java double or Double, the XPath result is a double.
- If the returned value is a Java float or Float, the XPath result is a float.
- If the returned value is a Java int, short, long, character, or byte, or one of their object wrapper equivalents, the XPath result is an integer.
- If the returned value is a Java String, the XPath result is a string.
- If the returned value is an instance of the Saxon class `net.sf.saxon.om.NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]` (a node in a Saxon tree), the XPath value will be a sequence containing a single node.
- If the returned value is an instance of `javax.xml.transform.Source` (other than a `NodeInfo`), a tree is built from the specified `Source` object, and the root node of this tree is returned as the result of the function.
- If the returned value is an instance of the Saxon class `net.sf.saxon.value.ValueRepresentation [Javadoc: net.sf.saxon.value.ValueRepresentation]`, the returned value is used unchanged.
- If the returned value is is an instance of the Saxon class `net.sf.saxon.om.SequenceIterator [Javadoc: net.sf.saxon.om.SequenceIterator]` (an iterator over a sequence), the XPath value will be the sequence represented by this iterator. It is essential that this iterator properly implements the method `getAnother()` which returns a new iterator over the same sequence of nodes or values, positioned at the start of the sequence.
- If the returned value is an instance of the Java class `java.util.Collection`, or if it is an array, the XPath value will be the sequence represented by the contents of this `Collection` or array. The members of the collection or array will each be converted to an XPath value, as if each member was supplied from a separate function call. An error is reported if the result contains a list or array nested within another list or array. The contents of the list or array are copied immediately on return from the function, so the original `List` or array object itself may be safely re-used.
- If the returned value is a DOM Node, and it is recognized as a wrapper around a Saxon node, then the node is unwrapped and the underlying Saxon node is returned. If the returned value is some other kind of DOM Node, then a Saxon wrapper is added. (This is an imperfect solution, since it can lead to problems with node identity and document order.)

- If the returned value is a DOM `NodeList`, the list of nodes is returned as a Saxon node-set. Each node is handled in the same way as a Node that is returned directly as the result.
- If the result is any other Java object (including null), it is returned as a "wrapped Java object".

Writing reflexive extension functions for .NET

> Reflexive extension functions involve dynamic loading of assemblies, which can be tricky to get working. Also, they aren't supported under Saxon-HE. Consider using integrated extension functions instead.

On the .NET platform extension functions may be implemented in any .NET language (the examples here assume C#).

> Queries and stylesheets running on the .NET platform may also call Java extension functions, provided the Java class is part of the standard library implemented in the OpenJDK DLL, or in Saxon itself. For calling conventions, see Writing extension functions in Java. If you want to write your own extensions in Java, you will need to compile them to .NET assemblies using IKVMC.

An extension function is invoked using a name such as `prefix:localname()`. The prefix must be the prefix associated with a namespace declaration that is in scope. The namespace URI is used to identify a .NET class, and the local name is used to identify a method, property, or constructor within the class.

The basic form of the namespace URI is "clitype:" followed by the fully-qualified type name (for example `xmlns:env="clitype:System.Environment"`). This form works for system classes and classes in a loaded assembly. If an assembly needs to be loaded, extra information can be given in the form of URI query parameters. For example `xmlns:env="clitype:Acme.Payroll.Employee?asm=payroll;version=4.12.0.0"`.

The parameters that are recognized are:

Table 11.6.

Keyword	Value
asm	The simple name of the assembly
ver	The version number, up to four integers separated by periods
loc	The culture (locale), for example "en-US"
sn	The public key token of the assembly's strong name, as 16 hex digits
from	The location of the assembly, as a URI
partialname	The partial name of the assembly (as supplied to `Assembly.LoadWithPartialName()`).

If the `from` keyword is present, the other parameters are ignored. The value of `from` must be the URI of the DLL to be loaded: if it is relative, it is relative to the base URI of the expression containing the call to the extension function (regardless of where the namespace is actually declared).

If the `partialName` keyword is present, the assembly is loaded (if possible) using `Assembly.LoadWithPartialName()` and the other parameters are ignored.

If the assembly is a library DLL in the global assembly cache, use the `gacutil /l` command to list the assemblies present in the GAC, and to extract the required version, culture, and strong name attributes. For example, suppose you want to call a static method `disappear()` in class `Conjurer` in namespace `Magic.Circle`, and this class is contained in an assembly `Magic` listed as:

```
Magic,              Version=7.2.2200.0,              Culture=neutral,
PublicKeyToken=b03f5f7f11d50a4b, Custom=null
```

Then the URI you would use to identify this class is `clitype:Magic.Circle.Conjurer?asm=Magic;ver=7.2.2200.0;sn=b03f5f7f11d50a4b`, and an actual call of the function might take the form:

```
<xsl:value-of select="m:disappear()"
     xmlns:m="clitype:Magic.Circle.Conjurer?asm=Magic;ver=7.2.2200.0;sn=b03f5f7
```

Tips for Dynamic Loading in .NET"

Here are some hints and tips that might help you to get dynamic loading working under .NET.

.

First decide whether you want to load the assembly (DLL) containing the extension functions from local filestore or from the Global Assembly Cache.

If you want to load it from the GAC you must compile the assembly with a strong name, and deploy it to the GAC. For advice on doing this, see http://support.microsoft.com/kb/324168. (In .NET framework 2.0 there was a handy Control Panel tool for this, but it is no longer available for security reasons.)

For local loading, the following techniques work:

- If the query or transformation is controlled from a user-written application in C# or another .NET language, include the extension function implementation in the same assembly as the controlling application, or in another assembly which is statically referenced from the controlling application. In this case it is only necessary to name the assembly, for example `<e att="{my:ext()}" xmlns:my="clitype:Namespace.ClassName?asm=Samples"/>`
- Another approach is to copy the assembly into the same directory as the controlling application or, if using the Saxon `Query` or `Transform` command from the command line, the directory containing the Saxon `Query.exe` and `Transform.exe` executables. Again in this case it should simply be necessary to give the assembly name as above.
- As an alternative, an assembly held in local filestore can be loaded by reference to the file, for example `xmlns:my="clitype:Namespace.ClassName?from=file:///c:/lib/Samples.dll"`. The URI can be given as an absolute URI, or as a relative URI which is interpreted relative to the base URI of the expression containing the function call.

For production running it is probably more appropriate to place the assembly holding extension functions in the global assembly cache. It can then generally be referenced in one of two ways:

- By partial name, for example `xmlns:my="clitype:Namespace.ClassName?partialname=Samples"`
- By fully-qualified name, for example `xmlns:my="clitype:Namespace.ClassName?asm=Samples;ver=3.0.0.1;loc=neutral;sn=e1f2a3b4e1f2a3b4"`

The following example shows how to call system methods in the .NET framework:

```
<out xmlns:Environment="clitype:System.Environment"
     xmlns:OS="clitype:System.OperatingSystem">
    <xsl:variable name="os" select="Environment:OSVersion()"/>
    <v platform="{OS:Platform($os)}" version="{OS:Version($os)}"/>
</out>
```

Identifying and Calling Specific Methods

The rest of this section considers how a .NET method, property, or constructor is identified. This decision (called binding) is always made at the time the XPath expression is compiled.

There are three cases to consider: static methods, constructors, and instance-level methods. In addition, a public property in a class is treated as if it were a zero-argument method, so static properties can be accessed in the same way as static methods, and instance-level properties in the same way as instance-level methods. (Note that the property name is used directly: it is not prefixed by "get".)

- Calling Static Methods in a .NET Class
- Calling .NET Constructors
- Calling .NET Instance-Level Methods

Calling Static Methods in a .NET Class

can be called directly. The localname of the function must match the name of a public static method in this class. The names match if they contain the same characters, excluding hyphens and forcing any character that follows a hyphen to upper-case. For example the XPath function call `To-string()` matches the .NET method `ToString()`; but the function call can also be written as `ToString()` if you prefer.

If there are several methods in the class that match the localname, and that have the correct number of arguments, then the system attempts to find the one that is the best fit to the types of the supplied arguments: for example if the call is `f(1,2)` then a method with two `int` arguments will be preferred to one with two `float` arguments. The rules for deciding between methods are quite complex. Essentially, for each candidate method, Saxon calculates the "distance" between the types of the supplied arguments and the .NET class of the corresponding argument in the method's signature, using a set of tables given below. For example, the distance between the XPath data type `xs:integer` and the .NET type `long` is very small, while the distance between an XPath `xs:integer` and a .NET `bool` is much larger. If there is one candidate method where the distances of all arguments are less-than-or-equal-to the distances computed for other candidate methods, and the distance of at least one argument is smaller, then that method is chosen.

If there are several methods with the same name and the correct number of arguments, but none is preferable to the others under these rules, an error is reported: the message indicates that there is more than one method that matches the function call.

This binding is carried out statically, using the static types of the supplied arguments, not the dynamic types obtained when the arguments are evaluated. If there is insufficient static type information to distinguish the candidate methods, an error is reported. You can supply additional type information using the `treat as` expression, or by casting. Often it is enough simply to declare the types of the variables used as arguments to the function call.

For example (in XSLT):

```
<xsl:value-of select="math:Sqrt($arg)" xmlns:math="clitype:System.Math"/>
```

This will invoke the static method `System.Math#Sqrt()`, applying it to the value of the variable `$arg`, and copying the value of the square root of `$arg` to the result tree. The value of $arg must be convertible to a double under the same rules as for a native XPath function call.

Similarly (in XQuery):

```
<a xmlns:double="type:System.Double"/>
                              {double:MaxValue()} </a>
```

This will output the value of the static field `System.Double#MaxValue`. (In practice, it is better to declare the namespace in the query prolog, or predeclare it using the API, because it will then not be copied to the result tree.)

A static method called as an extension function may have an extra first argument of type `net.sf.saxon.expr.XPathContext` [Javadoc: `net.sf.saxon.expr.XPathContext`]. This argument is not supplied by the calling XPath or XQuery code, but by Saxon itself. The `XPathContext` object provides methods to access many internal Saxon resources, the most useful being `getContextItem()` which returns the context item from the dynamic context. The class `net.sf.saxon.expr.XPathContext` is in the assembly `saxon9.dll`, and the module containing the code of the extension function will therefore need to contain a reference to that DLL. The class itself is written in Java, and is documented in the Javadoc documentation. The `XPathContext` object is available with static or instance-level methods, but not with constructors.

The following example shows a function that obtains the host language from the Saxon evaluation context:

```
public static string HostLanguage(net.sf.saxon.expr.XPathContext contex
{
    int lang = context.getController().getExecutable().getHostLanguage(
    if (lang == net.sf.saxon.Configuration.XQUERY)
    {
        return "XQuery";
    }
    else if (lang == net.sf.saxon.Configuration.XSLT)
    {
        return "XSLT";
    }
    else if (lang == net.sf.saxon.Configuration.XPATH)
    {
        return "XPath";
    }
    else
    {
        return "unknown";
    }
}
```

If this method appears in class `com.example.code.Utils`, then it can be accessed using the following code in XSLT:

```
<xsl:value-of select="nd:HostLanguage()"
    xmlns:nd="java:com.example.code.Utils"/>
```

or the following in XQuery:

```
<line xmlns:nd="java:com.example.code.Utils">
    { nd:HostLanguage() }
</line>
```

Calling .NET Constructors

are called by using the function named `new()`. If there are several constructors, then again the system tries to find the one that is the best fit, according to the types of the supplied arguments. The result of calling `new()` is an XPath value whose type is denoted by a QName whose local name is the actual .NET class (for example `System.Data.SqlClient.SqlConnection` or `System.Collections.ArrayList`) and whose namespace URI is `http://saxon.sf.net/clitype`. The only things that can be done with a wrapped .NET Object are to assign it to a variable, to pass it to an extension function, and to convert it to a string or boolean, using the rules given below.

The use of external object types in namespace `http://saxon.sf.net/clitype` reflects the .NET type hierarchy. For example, if a variable is declared to accept a type of `net:System.Collections.IList`, then a value of type `net:System.Collections.ArrayList` will be accepted, but a value of type `net:System.Collections.HashTable` will not.

Calling .NET Instance-Level Methods

(that is, non-static methods) are called by supplying an extra first argument of type `external .NET object` which is the object on which the method is to be invoked. An external .NET Object may be created by calling an extension function (e.g. a constructor) that returns an object; it may also be passed to the query or stylesheet as the value of a global parameter. Matching of method names is done as for static methods. If there are several methods in the class that match the localname, the system again tries to find the one that is the best fit, according to the types of the supplied arguments.

For example, the following XSLT stylesheet prints the operating system name and version.

```
<xsl:stylesheet version="2.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template name="main">
  <out xmlns:env="clitype:System.Environment" xmlns:os="clitype:System.Operatin
    <xsl:variable name="os" select="env:OSVersion()"/>
      <v platform="{os:Platform($os)}" version="{os:Version($os)}"/>
  </out>
</xsl:template>
</xsl:stylesheet>
```

The equivalent in XQuery is:

```
declare namespace env="clitype:System.Environment";
declare namespace os="clitype:System.OperatingSystem";
let $os := env:OSVersion() return
<v platform="{os:Platform($os)}" version="{os:Version($os)}"/>
```

As with static methods, an instance-level Java method called as an extension function may have an extra first argument of class `net.sf.saxon.expr.XPathContext [Javadoc: net.sf.saxon.expr.XPathContext]`. This argument is not supplied by the calling XPath or XQuery code, but by Saxon itself. The `XPathContext` object provides methods to access many internal Saxon resources, the most useful being `getContextItem()` which returns the context item from the dynamic context.

If any exceptions are thrown by the method, or if a matching method cannot be found, processing of the stylesheet will be abandoned. If the tracing option has been set (-TJ) on the command line, a full stack trace will be output. The exception will be wrapped in a `TransformerException` and passed to any user-specified `ErrorListener` object, so the `ErrorListener` can also produce extra diagnostics.

Converting Arguments to .NET Extension Functions

This section describes how XPath values supplied in a call to a .NET extension function are converted to .NET values.

- Converting Atomic Values and Sequences
- Converting Nodes and Sequences of Nodes
- Converting Wrapped .NET Objects

Converting Atomic Values and Sequences

The following conversions are supported between the static type of the supplied value of the argument, and the declared .NET type of the argument. The mappings are given in order of preference; a type that appears earlier in the list has smaller "conversion distance" than one appearing later. These priorities are used to decide which method to call when the class has several methods of the same name. Simple classes (such as boolean) are acceptable wherever the corresponding wrapper class (Boolean) is allowed. Class names shown in italics are Saxon-specific classes.

If the value is a (typically the result of a call to another extension function, then the underlying .NET object is passed to the method. An error will occur if it is the wrong .NET type.

If the required type is one of the types used in the `Saxon.Api` namespace to represent XPath values (for example `XdmValue`, `XdmNode`, `XdmAtomicValue`, then the value is converted to an instance of this class and passed over unchanged. If the expected type is `XdmAtomicValue` and the supplied value is a node, the node will be atomized (this must result in a single atomic value). Use of types such as `XdmValue` is available only when the query or transformation is invoked using the .NET API, it does not work when running from the command line.

If the static type of the supplied value allows a sequence of more than one item, then Saxon looks for a method that expects a `net.sf.saxon.om.SequenceIterator`, a `net.sf.saxon.value.Value`, an `ICollection` or an array in that order. (The first two classes are Saxon-specific). In all these cases except the last the item type of the supplied value plays no role.

Nodes in the supplied sequence are atomized only if the .NET method requires an atomic type such as an integer or string. If the method requires an ICollection, then the contents of the sequence will be supplied . The objects in the List will typically be Saxon `net.sf.saxon.om.Item [Javadoc: net.sf.saxon.om.Item]` objects. (Saxon does not yet recognize the generic types in .NET 2.0, which allow the item type of a collection to be declared). If atomization is required, you can force it by calling the `data()` function.

If the required type is an array, Saxon will attempt to create an array of the required type. This will not always succeed, for example if the array has type `X[]` where X is an interface rather than a concrete class. If it is an array of items or nodes, the nodes in the supplied sequence will be inserted into the array directly; if it is an array of a type such as integer or double, the sequence will first be atomized.

If the supplied value is a singleton (a sequence of one item) then the type of that item is decisive. If it is a sequence of length zero or more than one, then the general rules for a sequence are applied, and the types of the items within the sequence are irrelevant.

If the supplied value contains more than item and only a single item is expected, an error is reported. There is no implicit extraction of the first value (as happened in earlier releases).

Table 11.7.

boolean	, bool
dateTime	, Date
date	, Date
decimal	, decimal, double, float
double	, double
duration	
float	, float, double
integer	, decimal, long, integer, short, byte, double, float
string	, string
anyURI	, Uri, string

QName	
node	, NodeList, (Element, Attr, Document, DocumentFragment, Comment, Text, ProcessingInstruction, CharacterData), Node, Boolean, Byte, Character, Double, Float, Integer, Long, Short, (String, CharSequence), Object
sequence	, , List, NodeList, , Node, (String, CharSequence), Boolean, Byte, Character, Double, Float, Integer, Long, Short, Object

Saxon tries to select the appropriate method based on the of the arguments to the function call. If there are several candidate methods, and there is insufficient information available to decide which is most appropriate, an error is reported. The remedy is to cast the arguments to a more specific type.

Converting Nodes and Sequences of Nodes

If the expected type is a generic collection type, or an array of the Saxon class `NodeInfo [Javadoc: net.sf.saxon.om.NodeInfo]`, or a `Value` or `SequenceIterator [Javadoc: net.sf.saxon.om.SequenceIterator]`, Saxon will pass the nodes supplied in the call in their native Saxon representation.

Saxon also recognizes `System.Xml.Node` and its subtypes. However, calling a method that expects an instance if `System.Xml.Node` will only work if the actual node supplied as the argument value is a Saxon-wrapped DOM node, that is (a) the input to the stylesheet or query must be supplied in the form of a wrapped DOM, and (b) the external function must be called supplying a node obtained from the input document, not some node in a temporary tree created in the course of stylesheet or query processing.

If an external function returns an `System.Xml.Node` then it will be wrapped as a Saxon node. This may be expensive if it is done a lot, since each such node will acquire its own document wrapper.

Converting Wrapped .NET Objects

Saxon allows an extension function to be return an arbitrary .NET object. This will then be wrapped as an XPath item, so that it can be held in a variable and passed subsequently as an argument to another extension function. This second extension function will see the original Java object minus its wrapper, provided it is declared to expect the appropriate Java class.

A wrapped .NET object may be converted to another data type as follows.

- It is converted to a string by using its ToString() method; if the object is null, the result is the empty string "".
- It is converted to a boolean as follows: if it is null, the result is false, otherwise it is converted to a string and the result is true if and only if the string is non-empty.

The type of a wrapped Java object may be declared in a variable declaration or function signature using a type name whose namespace URI is "http://saxon.sf.net/.net-type", and whose local name is the fully qualified name of the .NET class, with any "$" signs replaced by hyphens. For example, the `sql:connection` extension function returns a value of type `{http://saxon.sf.net/clitype}System.Data.SQlClient.SqlConnection`.

Converting the Result of a .NET Extension Function

This section explains how the value returned by a .NET extension function is converted to an XPath value.

The result type of the method is converted to an XPath value as follows.

- If the method returns void, the XPath value is an empty sequence.
- If the method returns null, the XPath value is an empty sequence.
- If the method is a constructor, the XPath value is of type "wrapped .NET object". The only way of using this is by passing it to another external function, or by converting it to one of the standard XPath data types as described above.
- If the returned value is an `XdmValue` or one of its subclasses such as `XdmNode` or `XdmAtomicValue`, then it is used , after unwrapping. (These types are defined in the Saxon.Api namespace). Note that if the method constructs a new XdmNode and returns it, then it must be built using the Name Pool that is in use by the transformation or query, which means in practice that it must be built using a `DocumentBuilder` derived from the same `Saxon.Api.Processor`.
- If the returned value is a .NET bool, the XPath result is a boolean.
- If the returned value is a .NET double, the XPath result is a double.
- If the returned value is a .NET float, the XPath result is a float.
- If the returned value is a .NET Int64, Int32, or Int16, or one of their object wrapper equivalents, the XPath result is an integer.
- If the returned value is a .NET string, the XPath result is a string.
- If the returned value is an instance of the Saxon class `net.sf.saxon.om.NodeInfo` (a node in a Saxon tree), the XPath value will be a sequence containing a single node.
- If the returned value is an instance of the Saxon class `net.sf.saxon.value.Value`, the returned value is used unchanged.
- If the returned value is is an instance of the Saxon class `net.sf.saxon.om.SequenceIterator` (an iterator over a sequence), the XPath value will be the sequence represented by this iterator. It is essential that this iterator properly implements the method `getAnother()` which returns a new iterator over the same sequence of nodes or values, positioned at the start of the sequence.
- If the returned value is an instance of the .NET interface `System.Collections.IEnumerable`, or if it is an array, the XPath value will be the sequence represented by the contents of this collection or array. The members of the list or array will each be converted to an XPath value, as if each member was supplied from a separate function call. An error is reported if the result contains a list or array nested within another list or array. The contents of the list or array are copied immediately on return from the function, so the original collection or array object itself may be safely re-used.
- If the result is any other Java object (including null), it is returned as a "wrapped Java object".

In XSLT, the system function `function-available(String name)` returns true if there appears to be a method available with the right name. The function also has an optional second argument to test whether there is a method with the appropriate number of arguments. However, it is not possible to test whether the arguments are of appropriate types. If the function name is "new" it returns true so long as the class is not an abstract class or interface, and so long as it has at least one constructor.

Writing XSLT extension instructions

Saxon implements the element extensibility feature defined in the XSLT standard. This feature allows you to define your own instruction types for use in the stylesheet. These instructions can be used anywhere within a , for example as a child of `xsl:template`, `xsl:if`, `xsl:variable`, or of a literal result element.

To implement and use extension instructions, three steps are necessary:

1. There must be a class that implements the interface `ExtensionElementFactory [Javadoc: com.saxonica.xsltextn.ExtensionElementFactory]`, which recognizes all the extension elements in a particular namespace and provides the Java code to implement them.

2. This factory class must be associated with a namespace URI and registered with the `Configuration [Javadoc: net.sf.saxon.Configuration]`, which can be done either by calling the method `setExtensionElementNamespace(namespace, classname) [Javadoc: net.sf.saxon.Configuration#setExtensionElementNamespace]`, or by means of an entry in the configuration file.

3. Within the stylesheet, there must be a namespace declaration that binds a prefix to this namespace URI, and the prefix must be declared as an extension namespace by means of the `extension-element-prefixes` attribute, typically on the `xsl:stylesheet` element. (A rarely-used alternative is to declare it in the `xsl:extension-element-prefixes` attribute of an enclosing literal result element.)

Saxon itself provides a number of stylesheet elements beyond those defined in the XSLT specification, including `saxon:assign`, `saxon:entity-ref`, and `saxon:while`. To enable these, use the standard XSLT extension mechanism: define `extension-element-prefixes="saxon"` on the xsl:stylesheet element, or `xsl:extension-element-prefixes="saxon"` on any enclosing literal result element.

Any element whose prefix matches a namespace listed in the `extension-element-prefixes` attribute of an enclosing element is treated as an extension element. If no class can be instantiated for the element (for example, because no `ExtensionElementFactory [Javadoc: com.saxonica.xsltextn.ExtensionElementFactory]` has been registered for the relevant namespace, or because the `ExtensionElementFactory` doesn't recognise the local name), then fallback action is taken as follows. If the element has one or more `xsl:fallback` children, they are processed. Otherwise, an error is reported. When `xsl:fallback` is used in any other context, it and its children are ignored.

Within the stylesheet it is possible to test whether an extension element is implemented by using the system function `element-available()`. This returns true if the namespace of the element identifies it as an extension element (or indeed as a standard XSLT instruction) and if a class can be instantiated to represent it. If the namespace is not that of an extension element, or if no class can be instantiated, it returns false.

The interface `net.sf.saxon.style.ExtensionElementFactory [Javadoc: com.saxonica.xsltextn.ExtensionElementFactory]` interface. defines a single method, `getExtensionClass()`, which takes the local name of the element (that is, the name without its namespace prefix) as a parameter, and returns the Java class used to implement this extension element (for example, `return SQLConnect.class`). The class returned must be a subclass of `net.sf.saxon.style.StyleElement`, and the easiest way to implement it is as a subclass of `net.sf.saxon.style.ExtensionInstruction`.

Implementing extension instructions

The best way to see how to implement an extension element is by looking at the example, for SQL extension elements, provided in package `net.sf.saxon.option.sql`, and at the sample stylesheet which uses these extension elements. Start with the class `net.sf.saxon.option.sql.SQLElementFactory` [Javadoc: net.sf.saxon.option.sql.SQLElementFactory]

The `StyleElement` class represents an element node in the stylesheet document. Saxon calls methods on this class to validate and type-check the element, and to generate a node in the expression tree that is evaluated at run-time. Assuming that the class is written to extend `ExtensionInstruction` [Javadoc: net.sf.saxon.style.ExtensionInstruction], the methods it should provide are:

Table 11.8.

prepareAttributes()	This is called while the stylesheet tree is still being built, so it should not attempt to navigate the tree. Its task is to validate the attributes of the stylesheet element and perform any preprocessing necessary. For example, if the attribute is an attribute value template, this includes creating an Expression that can subsequently be evaluated to get the AVT's value.
validate()	This is called once the tree has been built, and its task is to check that the stylesheet element is valid "in context": that is, it may navigate the tree and check the validity of the element in relation to other elements in the stylesheet module, or in the stylesheet as a whole. By convention, a parent element contains checks on its children, rather than the other way around: this allows child elements to be reused in a new context without changing their code. The system will automatically call the method `mayContainSequenceConstructor()`. If this returns true, it will automatically check that all the children are instructions (that is, that their `isInstruction()` method returns true).If the extension element is not allowed to have any children, you can call `checkEmpty()` from the `validate()` method. However, users will normally expect that an extension instruction is allowed to contain an `xsl:fallback` child instruction, and you should design for this.If there are any XPath expressions in attributes of the extension instruction (for example a `select` attribute or an attribute value template), then the `validate()` method should call the `typeCheck()` method to process these expressions: for example `select = typeCheck("select", select);`
compile()	This is called to create an Expression object which is added to the expression tree. See below for further details.
isInstruction()	This should return true, to ensure that the element is allowed to appear within a template body.
mayContainSequenceConstructor()	This should return true, to ensure that the element can contain instructions. Even if it can't contain anything else, extension elements should allow an xsl:fallback instruction to provide portability between processors

The `StyleElement [Javadoc: net.sf.saxon.style.StyleElement]` class has access to many services supplied either via its superclasses or via the XPathContext object. For details, see the API documentation of the individual classes.

The simplest way to implement the `compile()` method is to return an instance of a class that is defined as a subclass of `SimpleExpression`. However, in principle any `Expression [Javadoc: net.sf.saxon.expr.Expression]` object can be returned, either an expression class that already exists within Saxon, or a user-written implementation. A subclass of `SimpleExpression` should implement the methods `getImplementationMethod()` and `getExpressionType()`, and depending on the value returned by `getImplementationMethod()`, should implement one of the methods `evaluateItem()`, `iterate()`, or `process()`.

Customizing Serialization

The output of a Saxon stylesheet or query can be directed to a user-defined output filter. This filter can be defined either as a SAX2 `ContentHandler`, or as a subclass of the Saxon classes `net.sf.saxon.event.Receiver`.

One advantage of using the Saxon classes is that more information is available from the stylesheet, for example the attributes of the `xsl:output` element; another is that (if you are using the schema-aware version of the product) type annotations are available on element and attribute nodes.

A transformation can be invoked from the Java API using the standard JAXP method `transformer.transform(source, result)`. The second argument must implement the JAXP class `javax.xml.transform.Result`. To send output to a SAX `ContentHandler`, you can wrap the `ContentHandler` in a JAXP `SAXResult` object. To send output to a Saxon `Receiver [Javadoc: net.sf.saxon.event.Receiver]` (which might also be an `Emitter [Javadoc: net.sf.saxon.serialize.Emitter]`), you can supply the `Receiver` directly, since the Saxon `Receiver` interface extends the JAXP `Result` interface.

When running XQuery, Saxon offers a similar method on the `XQueryExpression [Javadoc: net.sf.saxon.query.XQueryExpression]` object: the `run()` method. This also takes an argument of type `Result`, which may be (among other things) a `SAXResult` or a Saxon `Receiver`.

Some `ContentHandler` implementations require a sequence of events corresponding to a well-formed document (that is, one whose document node has exactly one element node and no text nodes among its children). If this is the case, you can specify the additional output property `saxon:require-well-formed="yes"`, which will cause Saxon to report an error if the result tree is not well-formed.

As specified in the JAXP interface, requests to disable or re-enable output escaping are also notified to the content handler by means of special processing instructions. The names of these processing instructions are defined by the constants `PI_DISABLE_OUTPUT_ESCAPING` and `PI_ENABLE_OUTPUT_ESCAPING` defined in class `javax.xml.transform.Result`.

As an alternative to specifying the destination in the `transform()` or `run()` methods, the `Receiver [Javadoc: net.sf.saxon.event.Receiver]` or `ContentHandler` to be used may be specified in the `method` attribute of the `xsl:output` element, as a fully-qualified class name; for example `method="prefix:com.acme.xml.SaxonOutputFilter"`. The namespace prefix is ignored, but must be present to meet XSLT conformance rules.

An abstract implementation of the `Receiver [Javadoc: net.sf.saxon.event.Receiver]` interface is available in the `Emitter [Javadoc: net.sf.saxon.serialize.Emitter]` class. This class provides additional functionality useful if you want to serialize the result to a byte or character output stream. If the `Receiver` that you supply as an output destination is an instance of `Emitter`, then it has access to all the serialization parameters supplied in the `xsl:output` declaration, or made available using the Java API.

See the documentation of class `net.sf.saxon.event.Receiver [Javadoc: net.sf.saxon.event.Receiver]` for details of the methods available, or implementations such as `HTMLEmitter [Javadoc: net.sf.saxon.serialize.HTMLEmitter]` and `XMLEmitter [Javadoc: net.sf.saxon.serialize.XMLEmitter]` and `TEXTEmitter [Javadoc: net.sf.saxon.serialize.TEXTEmitter]` for the standard output formats supported by Saxon.

It can sometimes be useful to set up a chain of `Receiver`s working as a pipeline. To write a filter that participates in such a pipeline, the class `ProxyReceiver [Javadoc: net.sf.saxon.event.ProxyReceiver]` is supplied. See the class `XMLIndenter [Javadoc: net.sf.saxon.serialize.XMLIndenter]`, which handles XML indentation, as an example of how to write a `ProxyReceiver`.

Saxon sets up such a pipeline when an output file is opened, using a class called the `SerializerFactory [Javadoc: net.sf.saxon.lib.SerializerFactory]`. You can override the standard `SerializerFactory` with your own subclass, which you can nominate to the `setSerializerFactory()` method of the `Configuration [Javadoc: net.sf.saxon.Configuration]`. This uses individual methods to create each stage of the pipeline, so you can either override the method that constructs the entire pipeline, or override a method that creates one of its stages. For example, if you want to subclass the `XMLEmitter [Javadoc: net.sf.saxon.serialize.XMLEmitter]` (perhaps to force all non-ASCII characters to be output as hexadecimal character references), you can override the method `newXMLEmitter()` to return an instance of your own subclass of `XMLEmitter`, which might override the method `writeEscape()`.

Rather than writing an output filter in Java, Saxon also allows you to process the output through another XSLT stylesheet. To do this, simply name the next stylesheet in the `saxon:next-in-chain` attribute of `xsl:output`.

Any number of user-defined attributes may be defined on `xsl:output`. These attributes must have names in a non-null namespace, which must not be either the XSLT or the Saxon namespace. The value of the attribute is inserted into the `Properties` object made available to the `Emitter` handling the output; they will be ignored by the standard output methods, but can supply arbitrary information to a user-defined output method. The name of the property will be the expanded name of the attribute in JAXP format, for example `{http://my-namespace/uri}local-name`, and the value will be the value as given, after evaluation as an attribute value template.

Implementing a collating sequence

Collations used for comparing strings can be specified by means of a URI. A collation URI may be used as an argument to many of the standard functions, and also as an attribute of `xsl:sort` in XSLT, and in the `order by` clause of a FLWOR expression in XQuery.

Saxon provides a range of mechanisms for binding collation URIs. The language specifications simply say that collations used in sorting and in string-comparison functions are identified by a URI, and leaves it up to the implementation how these URIs are defined.

There is one predefined collation that cannot be changed. This is the Unicode Codepoint Collation defined in the W3C specifications `http://www.w3.org/2005/xpath-functions/collation/codepoint`. This collates strings based on the integer values assigned by Unicode to each character, for example "ah!" sorts before "ah?" because the Unicode codepoints for "ah!" are (97, 104, 33) while the codepoints for "ah?" are (97, 104, 63).

You can use the Saxon configuration file to define collations: see The collations element.

In addition, by default, Saxon allows a collation URI to take the form `http://saxon.sf.net/collation?keyword=value;keyword=value;...`. The query parameters in the URI can be separated either by ampersands or semicolons, but semicolons are usually more convenient.

The same keywords are available on the Java and .NET platforms, but because of differences in collation support between the two platforms, they may interact in slightly different ways. The same collation URI may produce different sort orders on the two platforms. (One noteworthy difference is that the Java collations treat spaces as significant, the .NET collations do not.)

The keywords available in such a collation URI are the same as in the configuration file, and are as follows:

Table 11.9.

class	fully-qualified Java class name of a class that implements `java.util.Comparator`.	This parameter should not be combined with any other parameter. An instance of the requested class is created, and is used to perform the comparisons. Note that if the collation is to be used in functions such as `contains()` and `starts-with()`, this class must also be a `java.text.RuleBasedCollator`. This approach allows a user-defined collation to be implemented in Java.This option is also available on the .NET platform, but the class must implement the Java interface java.util.Comparator.
rules	details of the ordering required, using the syntax of the Java `RuleBasedCollator`	This defines exactly how individual characters are collated. (It's not very convenient to specify this as part of a URI, but the option is provided for completeness.) This option is also available on the .NET platform, and if used will select a collation provided using the OpenJDK implementation of `RuleBasedCollator`.
lang	any value allowed for `xml:lang`, for example `en-US` for US English	This is used to find the collation appropriate to a Java locale or .NET culture. The collation may be further tailored using the parameters described below.
ignore-case	yes, no	Indicates whether the upper and lower case letters are considered equivalent. Note that even when ignore-case is set to "no", case is less significant than the actual letter value, so that "XPath" and "Xpath" will appear next to each other in the sorted sequence.On the Java platform, setting ignore-case sets the collation strength to secondary.

ignore-modifiers	yes, no	Indicates whether non-spacing combining characters (such as accents and diacritical marks) are considered significant. Note that even when ignore-modifiers is set to "no", modifiers are less significant than the actual letter value, so that "Hofen" and "Höfen" will appear next to each other in the sorted sequence.On the Java platform, setting ignore-case sets the collation strength to primary.
ignore-symbols	yes, no	Indicates whether symbols such as whitespace characters and punctuation marks are to be ignored. This option currently has no effect on the Java platform, where such symbols are in most cases ignored by default.
ignore-width	yes, no	Indicates whether characters that differ only in width should be considered equivalent.On the Java platform, setting ignore-width sets the collation strength to tertiary.
strength	primary, secondary, tertiary, or identical	Indicates the differences that are considered significant when comparing two strings. A/B is a primary difference; A/a is a secondary difference; a/ä is a tertiary difference (though this varies by language). So if strength=primary then A=a is true; with strength=secondary then A=a is false but a=ä is true; with strength=tertiary then a=ä is false.This option should not be combined with the ignore-XXX options. The setting "primary" is equivalent to ignoring case, modifiers, and width; "secondary" is equivalent to ignoring case and width; "tertiary" ignores width only; and "identical" ignores nothing.
decomposition	none, standard, full	Indicates how the collator handles Unicode composed characters. See the JDK documentation for details. This option is ignored on the .NET platform.
alphanumeric	yes, no, codepoint	If set to yes, the string is split into a sequence of alphabetic

		and numeric parts (a numeric part is any consecutive sequence of ASCII digits; anything else is considered alphabetic). Each numeric part is considered to be preceded by an alphabetic part even if it is zero-length. The parts are then compared pairwise: alphabetic parts using the collation implied by the other query parameters, numeric parts using their numeric value. The result is that, for example, AD985 collates before AD1066. (This is sometimes called natural sorting.) The value `codepoint` requests alphanumeric collation with the "alpha" parts being collated by Unicode codepoint, rather than by the default collation for the Locale. This may give better results in the case of strings that contain spaces. Note that an alphanumeric collation cannot be used in conjunction with functions such as contains() and substring-before().
case-order	upper-first, lower-first	Indicates whether upper case letters collate before or after lower case letters.

This format of URI, `http://saxon.sf.net/collation?keyword=value;keyword=value;...`, is handled by Saxon's default `CollationURIResolver` `[Javadoc: net.sf.saxon.lib.CollationURIResolver]`. It is possible to replace or supplement this mechanism by registering a user-written `CollationURIResolver`. This must be an implementation of the Java interface `net.sf.saxon.lib.CollationURIResolver`, which only requires a single method, `resolve()`, to be implemented. The result of the method is in general a Java `Comparator`, though if the collation is to be used in functions such as `contains()` which match parts of a string rather than the whole string, then the result must also be an instance of either `java.text.RuleBasedCollator`, or of the Saxon interface `net.sf.saxon.sort.SubstringMatcher`.

In the Java API, a user-written `CollationURIResolver` is registered with the `Configuration` `[Javadoc: net.sf.saxon.Configuration]` object, either directly or in the case of XSLT by using the JAXP `setAttribute()` method on the `TransformerFactory` (the relevant property name is `FeatureKeys.COLLATION_URI_RESOLVER` `[Javadoc: net.sf.saxon.lib.FeatureKeys#COLLATION_URI_RESOLVER]`). This applies to all stylesheets and queries compiled and executed under that configuration.

It is also possible to register a collation (for example as an instance of the Java class `Collator` or `Comparator` with the `Configuration`. Such explicitly registered collations (together with those registered via the configuration file) are used before calling the `CollationURIResolver`. In addition, the APIs provided for executing XPath and XQuery expressions allow named collations to be registered by the calling application, as part of the static context.

At present there are no equivalent facilities in the .NET API (other than the use of the configuration file), though it is possible to manipulate collations by dropping down into the Java interface.

Localizing numbers and dates

It is possible to define a localized numbering sequence for use by `xsl:number` and `format-date()`. This sequence will be used when you specify a language in the `lang` attribute of the `xsl:number` element, or in the third argument of the functions `format-date()`, `format-time()`, and `format-dateTime()`. The feature is primarily intended to provide language-dependent numbers and dates, but in fact it can be used to provide arbitrary numbering sequences.

To implement a numberer for language X, you need to define a class that implements the interface `Numberer [Javadoc: net.sf.saxon.lib.Numberer]`; usually it will be convenient to write the class as a subclass of the supplied `AbstractNumberer [Javadoc: net.sf.saxon.number.AbstractNumberer]`. `Numberer` implementations are supplied for a number of languages, and you can use these as a prototype to write your own.

The languages supplied with the product are:

Table 11.10.

code	Language
da	Danish
de	German
en	English
fr	French
fr-BE	French (Belgium)
it	Italian
nl	Dutch
nl-BE	Flemish (Belgium)
sv	Swedish

The numbering sequence for English is used by default if no other can be loaded.

Normally your localization class will extend the class `AbstractNumberer` so that you can reuse functionality like roman numerals which do not need to be localized. Alternatively, if you only want to modify the existing English localization, you could choose to implement a subclass of `Numberer_en`.

You can override any of the non-private methods in the base class, but the most useful ones to implement are the following:

Table 11.11.

Method	**Effect**
ordinalSuffix	Supplies a suffix to be appended to a number to create the ordinal form, for example "1" becomes "1st" in English
toWords	Displays a number in words, in title case: for example "53" becomes "Fifty Three" in English
toOrdinalWords	Displays an ordinal number in words, in title case: for example "53" becomes "Fifty Third" in English

Method	Effect
monthName	Displays the name of a month, optionally abbreviated
dayName	Displays the name of a day of the week, optionally abbreviated

The class name can be anything you like, but by convention the Numberer for language LL is named `net.sf.saxon.number.Numberer_LL`.

The way that the numberer is registered with the Saxon Configuration differs between Saxon-HE on the one hand, and Saxon-PE/EE on the other. On Saxon-HE, you need to supply a `LocalizerFactory` that responds the request for a particular language: for example:

```
Configuration config = new Configuration();
config.setLocalizerFactory(new LocalizerFactory() {
  public Numberer getNumberer(String language, String country) {
    if (language.equals("jp")) {
      return Numberer_JP.getInstance();
    } else {
      return null;
    }
  }
});
```

You can also use this mechanism on Saxon-PE/HE, but an alternative is to register the localization module in the configuration file.

Writing a URI Resolver for Input Files

Saxon allows you to write your own `URIResolver` to handle the URIs of input documents, as defined in the JAXP specification. Such a `URIResolver` is used to process the URIs supplied to the `doc()` and `document()` functions. It is also used to process the URIs supplied for the source document and the stylesheet on the command line. In XSLT it is used to process the URIs used in the `xsl:include` and `xsl:import` and `xsl:import-schema` declarations, and in XQuery it supports the location URIs in `import schema`.

The `URIResolver` is called to process the supplied URI, and it returns a JAXP `Source` object, which Saxon uses as the source of the input. Note that the `Source` must be one of the implementations of `Source` that Saxon recognizes: you cannot write your own implementations of the JAXP `Source` class.

The `URIResolver` is used only for XML files. It is therefore not used to support the `unparsed-text()` function in XSLT, or to support `import module` in XQuery.

Writing a URI Resolver for Output Files

Saxon also allows you to write an `OutputURIResolver`, which performs an analogous role for URIs specified in the `href` attribute of `xsl:result-document`. This is therefore applicable to XSLT only. The `OutputURIResolver` is called when writing of the output document starts, at which point it must return a JAXP `Result` object to act as the output destination. It is called again when writing of an output document is complete.

You can nominate an `OutputURIResolver` by calling `((Controller)transformer).setOutputURIResolver(new`

`UserOutputResolver())`, or by calling `factory.setAttribute("http://saxon.sf.net/feature/outputURIResolver", new UserOutputResolver())`.

Chapter 12. Saxon Extensions

Introduction

This section describes the extensions and implementation-defined features provided with the Saxon product.

If you want to implement your own extensions, see Extensibility.

> Most of the extensions described in this section require Saxon-PE (Professional Edition) or higher. A few work with Saxon-HE (Home Edition), and some require Saxon-EE (Enterprise Edition). Check the details for each extension.

The extensions described here were provided because there are things that are difficult to achieve, or inefficient, using standard XSLT/XQuery facilities alone. In some cases they are retained from earlier releases even though equivalent functionality is available using the standard language. As always, it is best to stick to the standard if you possibly can: and most things possible, even if it's not obvious at first sight.

All Saxon extensions require a namespace declaration such `xmlns:saxon="http://saxon.sf.net/"` to appear. In XSLT this is typically declared on the `xsl:stylesheet` element; it is also useful to write `exclude-result-prefixes="saxon"` to prevent the Saxon namespace appearing in the result tree. In XQuery the namespace is declared by writing `declare namespace saxon="http://saxon.sf.net/";` in the Query prolog.

If you use XSLT extension instructions such as `saxon:doctype`, it is also necessary to include the attribute `extension-element-prefixes="saxon"`.

> Before using a Saxon extension, check whether there is an equivalent EXSLT extension available. EXSLT extensions are more likely to be portable across XSLT processors.

For details of additional extensions available in Query only, see Query Extensions.

Saxon also provides a set of extension elements providing access to SQL databases. These are described here.

Further information:

- EXSLT Extensions
- Extension attributes (XSLT only)
- Additional serialization parameters
- Extension functions
- The Map Extension
- Extension instructions

EXSLT Extensions

EXSLT [http://www.exslt.org/] is an initiative to define a standardized set of extension functions and extension elements that can be used across different XSLT processors.

Saxon supports the EXSLT modules Common, Math, Sets, DatesAndTimes, and Random. These functions are available both in XSLT and in XQuery. The full list of EXSLT extension functions implemented is:

- node-set(), object-type()
- abs(), acos(), asin(), atan(), atan2(), constant(), cos(), exp(), highest(), log(), lowest(), max(), min(), power() , random(), sin(), sqrt(), tan().
- difference(), intersection(), leading(), trailing(), has-same-node()
- add(), add-duration(), date(), date-time(), day-abbreviation(), day-in-month(), day-in-week(), day-in-year(), day-name(), day-of-week-in-month(), difference(), duration(), hour-in-day(), leap-year(), minute-in-hour(), month-abbreviation(), month-in-year(), month-name(), second-in-minute(), seconds(), sum(), time(), week-in-month(), week-in-year(), year().
- random-sequence()

There are some known restrictions and local interpretations:

- In the `set:leading()` and `set:trailing()` functions, Saxon does not implement the rule "If the first node in the second node set is not contained in the first node set, then an empty node set is returned." This rule prevents a pipelined implementation. Saxon returns all nodes that precede/follow the first/last node of the second node-set in document order, whether or not the two node-sets intersect.

EXSLT extensions that overlap XSLT 2.0 functionality have sometimes been retained in cases where they have no impact on the Saxon core code, but in cases (such as `func:function`) where the semantics are inconveniently different from XSLT 2.0, they have been withdrawn.

The function `math:power()` has been extended from the EXSLT definition to handle numeric data types other than xs:double. The result will now be an xs:integer if the first argument is an xs:integer and the second argument is a non-negative xs:integer. Otherwise, the result will be an xs:decimal if the first argument is an xs:decimal or xs:integer, and the second argument is a whole number (a number of any data type that is equal to some integer). In other cases the arguments are converted to xs:double and the result is an xs:double.

The specifications of the EXSLT date-and-time handling functions have little to say about timezones. Saxon generally handles inputs with or without a timezone, and uses the XPath 2.0 concept of implicit timezone to interpret the meaning of dates/times without a timezone. The current date and time used by EXSLT functions is the same as that used by the XPath 2.0 current-dateTime() function.

Extension attributes (XSLT only)

An extension attribute is an extra attribute on an XSLT-defined element. These attributes are all in the Saxon namespace `http://saxon.sf.net/`. This namespace needs to be declared in a namespace declaration, but it does not need to be listed in the `extension-element-prefixes` attribute.

For example, the `saxon:assignable` attribute can be set as follows:

```
<xsl:variable name="counter" saxon:assignable="yes"
    xmlns:saxon="http://saxon.sf.net/">
```

The extension attributes provided with the Saxon product are as follows:

- saxon:assignable: marks a variable as updateable
- saxon:explain: requests display of optimized expression tree
- saxon:memo-function: marks a function as a memo function

- saxon:read-once: enables serial processing of source documents
- saxon:threads: enables parallel processing of xsl:for-each instructions

saxon:assignable

This attribute may be set on a global `xsl:variable` element. The permitted values are "yes" and "no". If the variable is the subject of a saxon:assign instruction, it must be set to the value "yes". Setting this value to "yes" also ensures that the variable is actually evaluated, which is useful if the `select` expression calls extension functions with side-effects; without this, a variable that is never referenced may never be evaluated.

saxon:explain

This attribute may be set on any instruction in the stylesheet, including a literal result element, though the recommended use is to set it on an `xsl:template` or `xsl:function` declaration. The permitted values are "yes" and "no". If the value is "yes", then at compile time Saxon outputs (to the standard error output) a representation of the optimized expression tree for the template or function containing that instruction. The tree is represented by indentation. For example, consider this source code:

```

 <xsl:variable name="p" select="0"/>

 <xsl:template match="/" saxon:explain="yes" xmlns:saxon="http://saxon.sf.net/"
   <a>
     <xsl:choose>
     <xsl:when test="$p != 0"><xsl:value-of select="1 div $p"/></xsl:when>
     <xsl:otherwise>12</xsl:otherwise>
     </xsl:choose>
   </a>
 </xsl:template>
```

This produces the output:

```
Optimized expression tree for template at line 8 in file:/e:/temp/test.xsl:
<directElement name="a" validation="skip">
  <valueOf>
    <literal value="12" type="xs:string"/>
  </valueOf>
</directElement>
```

This indicates that the template has been reduced to an instruction to create an element with name `a`, whose content is a single text node holding the string "12". This is because Saxon has established at compile time that it will always take the "otherwise" branch of the `xsl:choose` instruction. There is no `xsl:value-of` instruction in the source code, but the literal text node "12" is compiled to the same code as if the user had written `<xsl:value-of select="'12'"/>`

To get this output for all templates and functions in the stylesheet, you can use the `-explain` option on the command line.

saxon:memo-function

This attribute may be set on the `xsl:function` element. The permitted values are "yes" and "no". Specifying "yes" indicates that Saxon should remember the results of calling the function in a cache, and if the function is called again with the same arguments, the result is retrieved from the cache rather than being recalculated. Don't use this if the function has side-effects (for example, if it calls `saxon:assign`, or an extension function with side-effects). Don't use it if the function accesses context information such as the context node or `position()` or `last()`. And be careful if the function constructs and returns a temporary tree: the effect will be that the function returns the same tree each time, rather than a copy of the tree (this difference will only show up if you compare the identity of nodes in the two trees).

saxon:read-once

This attribute may be set on the `xsl:copy-of` element. The allowed values are "yes" and "no". If set to yes, this attribute enables the optimization described in Streaming of Large Documents.

The attribute should be set to "yes" only if any external document read by this instruction is read at most once by the stylesheet. If this is not the case, the optimization is unsafe, since the functions `doc()` and `document()` are required to return the same results each time they are called.

saxon:threads

This attribute may be set on the `xsl:for-each` instruction. The value must be an integer. When this attribute is used with Saxon-EE, the items selected by the `select` expression of the instruction are processed in parallel, using the specified number of threads.

The threads are allocated on a round-robin basis: for example if `threads="3"` is specified, then the first item will be processed using thread 1, the second using thread 2, the third using thread 3, the fourth using thread 1 again, and so on. These threads are in addition to the main control thread (so there will be 4 threads in total). Before firing off the processing of the third item, the control thread will read off the results of processing the first item, and send them to the destination of the `xsl:for-each` instruction.

It is possible to specify `saxon:threads="1"`. In this case all the items in the input will be processed sequentially, but asynchronously with the thread that reads the items in the input sequence.

Processing using multiple threads can take advantage of multi-core CPUs. However, there is an overhead, in that the results of processing each item in the input need to be buffered. The overhead of coordinating multiple threads is proportionally higher if the per-item processing cost is low, while the overhead of buffering is proportionally higher if the amount of data produced when each item is processed is high. Multi-threading therefore works best when the body of the `xsl:for-each` instruction performs a large amount of computation but produces a small amount of output.

It is possible to combine multi-threading with sorting. However, the input is first read and sorted synchronously, and the items in the sorted sequence are then processed in parallel.

The effect of using extensions that have side-effects (including `saxon:assign`) in a multi-threaded loop is undefined (and probably fatal).

Multi-threaded processing is available only with Saxon-EE. The attribute `saxon:threads` is ignored with a warning if Saxon-EE is not in use. Under Saxon-EE it may also be disabled using the configuration option `FeatureKeys.ALLOW_MULTITHREADING` or `xslt/@allowMultiThreading='false'` in the configuration file. Multi-threaded processing is also disabled if code is compiled with tracing enabled, for example by using the -T option on the command line or by using an IDE debugger (this is because otherwise, the trace events would be hopelessly intermingled).

With multi-threaded processing, the output of different `xsl:message` instructions may appear in an unpredictable order. However, Saxon ensures that the `xsl:message` instruction is atomic, so one message will be completed before another starts. The same is true of the output from the `trace()` function call.

Additional serialization parameters

Saxon provides a number of additional serialization parameters, and several additional serialization methods: these have names in the Saxon namespace. These can be specified as attributes on the `xsl:output` and `xsl:result-document` elements (XSLT-only), in the Query prolog (XQuery only), or as extra parameters on the Query or Transform command line. They can also be specified in the query or transformation API.

For example, to request an indentation depth of one column, specify `!{http://saxon.sf.net/}indent-spaces=1` on the command line.

In XQuery, Saxon allows both standard serialization options and Saxon-specific serialization parameters to be specified by means of a `saxon:output` option declaration in the query prolog. For example:

```
declare namespace saxon="http://saxon.sf.net/";
declare option saxon:output "indent=yes";
declare option saxon:output "saxon:indent-spaces=3";
```

The standard serialization parameters described in The W3C Serialization specification [http://www.w3.org/TR/xslt-xquery-serialization/] are all available, namely:

- byte-order-mark
- cdata-section-elements
- doctype-public
- doctype-system
- encoding
- escape-uri-attributes
- include-content-type
- indent
- media-type
- method
- normalization-form
- omit-xml-declaration
- standalone
- undeclare-prefixes
- use-character-maps (only useful in XSLT)
- version

The Saxon-supplied serialization parameters are described on the following pages.

- The method attribute
- The saxon:base64Binary serialization method
- The saxon:hexBinary serialization method
- The saxon:ptree serialization method

- The saxon:character-representation attribute
- The saxon:double-space attribute
- The saxon:indent-spaces attribute
- The saxon:line-length attribute
- The saxon:next-in-chain attribute
- The saxon:recognize-binary attribute
- The saxon:require-well-formed attribute
- The saxon:supply-source-locator attribute
- The saxon:suppress-indentation attribute
- The saxon:xquery serialization method
- User-defined serialization attributes

The method attribute

The `method` attribute of `xsl:output` can take the standard values "xml", "html", "xhtml", or "text", or a .

If a QName in the Saxon namespace is specified, the name must be one of Saxon serialization methods supported in Saxon-PE and Saxon-EE. These are:

Table 12.1.

saxon:base64Binary	The saxon:base64Binary serialization method
saxon:hexBinary	The saxon:hexBinary serialization method
saxon:ptree	The saxon:ptree serialization method
saxon:xquery	The saxon:xquery serialization method

If a QName is specified, the local name must be the fully-qualified class name of a class that implements either the SAX2 `org.xml.sax.ContentHandler` interface, or the `net.sf.saxon.event.Receiver` interface. If such a value is specified, output is directed to a newly-created instance of the user-supplied class. You can pass additional information to this class by means of extra user-defined attributes on the `xsl:output` element.

The prefix of the must correspond to a valid namespace URI. It is recommended to use the Saxon URI "http://saxon.sf.net/", but this is not enforced.

When output is sent to a user-specified ContentHandler or Receiver, other serialization options (for example indentation, addition of meta elements in HTML, and generation of a DOCTYPE declaration) have no effect.

As an alternative to specifying your own output method, you can customize Saxon's serialization pipeline. There is a Saxon class `net.sf.saxon.event.SerializationFactory` that constructs the pipeline, from individual components that perform the various stages of serialization. You can define your own subclass of `net.sf.saxon.event.SerializationFactory`, and register this using the method `setSerializationFactory()` on the `Configuration` object. This allows you to insert your own steps into the pipeline, or to override any of the standard pipeline components with classes of your own.

The saxon:base64Binary serialization method

An additional serialization method `saxon:base64Binary` is available. This is intended to be useful when creating binary output files, for example images. All serialization properties other than `method` are ignored.

As with the `text` output method, all nodes in the result tree other than text nodes are ignored. Each text node must hold a string that is in the lexical space of the `xs:base64Binary` data type. The sequence of octets corresponding to this base64 value is written to the binary output file.

When invoking this method via an API, use the property value "{http://saxon.sf.net/}base64Binary".

When using this serialization method, the `omit-xml-declaration` parameter is automatically set to "yes".

```
<xsl:template name="main">
  <html>
    <head>
      <title>An image</title>
    </head>
    <body>
      <h1>An image</h1>
      <p><img src="image.gif"/></p>
    </body>
  </html>
  <xsl:result-document method="saxon:base64Binary" href="image.gif" media-type=
    <xsl:text>iVBORw0KGgoAAAANSUhEUgAAAAoAAAAKCAYAAACNMs+9AAAABGdBTUEAALGPC/xhB
        LEwAACxMBAJqcGAAAAAd0SU1FB9YGARc5KB0XV+IAAAAddEVYdENvbW1lbnQAQ3JlYXRlZC
        SU1Q72QlbgAAAF1JREFUGNO9zL0NglAAxPEfdLTs4BZM4DIO4C7OwQg2JoQ9LE1exdlYvBB
        4TLzw4d6+ErXMMcXuHWxId3KOETnnXXV6MJpcq2MLaI97CER3N0vr4MkhoXe0rZigAAAABJ
    </xsl:text>
  </xsl:result-document>
</xsl:template>
```

It is of course possible to construct the base64Binary value programmatically. The extension function saxon:octets-to-base64Binary may be useful to achieve this.

When writing output that is not entirely binary, but contains mixed binary and text, it may be more convenient to use the text output method with the saxon:recognize-binary serialization property.

See also the saxon:hexBinary serialization method.

The saxon:hexBinary serialization method

An additional serialization method `saxon:hexBinary` is available. This is intended to be useful when creating binary output files, for example images. All serialization properties other than `method` are ignored.

As with the `text` output method, all nodes in the result tree other than text nodes are ignored. Each text node must hold a string that is in the lexical space of the `xs:hexBinary` data type. The sequence of octets corresponding to this base64 value is written to the binary output file.

When invoking this method via an API, use the property value "{http://saxon.sf.net/}hexBinary".

When using this serialization method, the `omit-xml-declaration` parameter is automatically set to "yes".

```
<xsl:template name="main">
  <html>
    <head>
      <title>An image</title>
    </head>
    <body>
      <h1>An image</h1>
      <p><img src="image004.gif"/></p>
    </body>
  </html>
  <xsl:result-document method="saxon:hexBinary" href="image004.gif" media-type=
    <xsl:text>89504E470D0A1A0A0000000D49484452</xsl:text>
    <xsl:text>0000000A0000000A08060000008D32CF</xsl:text>
    <xsl:text>BD0000000467414D410000B18F0BFC61</xsl:text>
    <xsl:text>0500000009704859730000B1300000B</xsl:text>
    <xsl:text>1301009A9C180000000774494D4507D6</xsl:text>
    <xsl:text>06011739281D1757E20000001D744558</xsl:text>
    <xsl:text>74436F6D6D656E7400437265617465</xsl:text>
    <xsl:text>20776974682054686520474494D50EF64</xsl:text>
    <xsl:text>256E0000005D4944415418D3BDCCBD0D</xsl:text>
    <xsl:text>825000C4F11F74B4ECE0164CE0320EE0</xsl:text>
    <xsl:text>2ECEC1083626843D2C4D5EC5D958BC10</xsl:text>
    <xsl:text>5E67B8EA721F7FFEAD6E1F84CBCF0E1D</xsl:text>
    <xsl:text>EBE12B5CC31C5EE1D6C4877728E1139E</xsl:text>
    <xsl:text>75D757A309A5CAB630B688F7B0844773</xsl:text>
    <xsl:text>74BEBE0C921A177B4AD98A0000000049</xsl:text>
    <xsl:text>454E44AE426082</xsl:text>
  </xsl:result-document>
</xsl:template>
```

It is of course possible to construct the hexBinary value programmatically. The extension function saxon:octets-to-hexBinary may be useful to achieve this.

When writing output that is not entirely binary, but contains mixed binary and text, it may be more convenient to use the text output method with the saxon:recognize-binary serialization property.

See also the saxon:base64Binary serialization method.

The saxon:ptree serialization method

The serialization method `saxon:ptree` causes the result tree to be output as a PTree: see The PTree file format. This is a binary XML format that occupies roughly the same amount of disk space as standard lexical XML, but which saves time both on serializing and parsing, making it useful if the data has to be sent to another machine for the next stage of processing.

With this method, all serialization properties other than `method` are ignored.

The saxon:character-representation attribute

This attribute allows greater control over how non-ASCII characters will be represented on output.

With method="xml", two values are supported: "decimal" and "hex". These control whether numeric character references are output in decimal or hexadecimal when the character is not available in the selected encoding.

With HTML, the value may hold two strings, separated by a semicolon. The first string defines how non-ASCII characters within the character encoding will be represented, the values being "native", "entity", "decimal", or "hex". The second string defines how characters outside the encoding will be represented, the values being "entity", "decimal", or "hex". Here "native" means output the character as itself; "entity" means use a defined entity reference (such as "&eacute;") if known; "decimal" and "hex" refer to numeric character references. For example "entity;decimal" (the default) means that with encoding="iso-8859-1", characters in the range 160-255 will be represented using standard HTML entity references, while Unicode characters above 255 will be represented as decimal character references.

> This attribute is retained for the time being in the interests of backwards compatibility. However, the XSLT 2.0 specification makes it technically a non-conformance to provide attributes that change serialization behavior except in cases where the behavior is implementation-defined; and this is not such a case (the specification, at least in the case of the XML output method, does not allow a character to be substituted with a character reference in cases where the character is present in the chosen encoding). The best way of ensuring that non-ASCII characters are output using character references is to use `encoding="us-ascii"`.

The saxon:double-space attribute

When the output method is XML with `indent="yes"`, the `saxon:double-space` attribute may be used to generate an extra blank line before selected elements. The value is a whitespace-separated list of element names. The attribute follows the same conventions as `cdata-section-elements`: values specified in separate `xsl:output` or `xsl:result-document` elements are cumulative, and if the value is supplied programmatically via an API, or from the command line, then the element names are given in Clark notation, namely `{uri}local`. The effect of the attribute is to cause an extra blank line to be output before the start tag of the specified elements.

The saxon:indent-spaces attribute

When the output method is XML, HTML, or XHTML with `indent="yes"`, the `saxon:indent-spaces` attribute may be used to control the amount of indentation. The value must be an integer. The default value in the absence of this attribute is 3.

The saxon:line-length attribute

The value of `saxon:line-length` is an integer, with default value 80. With both the HTML and XML output methods, attributes are output on a new line if they would otherwise extend beyond this column position. With the HTML output method, furthermore, text lines are split at this line length when possible. In releases 9.2 and earlier, the HTML output method attempted to split lines that exceeded 120 characters in length.

The saxon:next-in-chain attribute

The `saxon:next-in-chain` attribute (XSLT-only) is used to direct the output to another stylesheet. The value is the URL of a stylesheet that should be used to process the output stream. In this case the output stream must always be pure XML, and attributes that control the format of the output (e.g. method, cdata-section-elements, etc) will have no effect. The output of the second stylesheet will be directed to the destination that would have been used for the first stylesheet if no `saxon:next-in-chain` attribute were not present.

Supplying a zero-length string is equivalent to omitting the attribute, except that it can be used to override a previous setting.

> An alternative way of chaining transformations is to use the saxon:compile-stylesheet() and saxon:transform() extension functions.

The saxon:recognize-binary attribute

This attribute is relevant only when using the text output method. If set to yes, the processing instructions `<?hex XXXX?>` and `<?b64 XXXX?>` will be recognized; the value is taken as a hexBinary or base64 representation of a character string, encoded using the encoding in use by the serializer, and this character string will be output without validating it to ensure it contains valid XML characters. Also recognized are `<?hex.EEEE XXXX?>` and `<?b64.EEEE XXXX?>`, where EEEE is the name of the encoding of the base64 or hexBinary data: for example `hex.ascii` or `b64.utf8`.

This enables non-XML characters, notably binary zero, to be output.

For example, given `<xsl:output method="text" saxon:recognize-binary="yes"/>`, the following instruction:

```
<xsl:processing-instruction name="hex.ascii" select="'00'"/>
```

outputs the Unicode character with codepoint zero ("NUL"), while

```
<xsl:processing-instruction name="b64.utf8" select="securityKey"/>
```

outputs the value of the `securityKey` element, on the assumption that this is base64-encoded UTF-8 text.

The saxon:require-well-formed attribute

This attribute affects the handling of result documents that contain multiple top-level elements or top-level text nodes. The W3C specifications allow such a result document, even though it is not a well-formed XML document. It is, however, a well-formed , which means it can be incorporated into a well-formed XML document by means of an entity reference.

The attribute `saxon:require-well-formed` is available, with values "yes" or "no". The default is "no". If the value is set to "yes", and a SAX destination (for example a `SAXResult`, a `JDOMResult`, or a user-written `ContentHandler`) is supplied to receive the results of the transformation, then Saxon will report an error rather than sending a non-well-formed stream of SAX events to the `ContentHandler`. This attribute is useful when the output of the stylesheet is sent to a component (for example an XSL-FO rendering engine) that is not designed to accept non-well-formed XML result trees.

Note also that namespace undeclarations of the form `xmlns:p=""` (as permitted by XML Namespaces 1.1) are passed to the `startPrefixMapping()` method of a user-defined `ContentHandler` only if `undeclare-prefixes="yes"` is specified on `xsl:output`.

The saxon:supply-source-locator attribute

This attribute is relevant only when output is sent to a user-written `ContentHandler`, that is, a `SAXResult`. It causes extra information to be maintained and made available to the `ContentHandler` for diagnostic purposes: specifically, the `Locator` that is passed to the `ContentHandler` via the `setDocumentLocator` method may be cast to a `ContentHandlerProxyLocator`, which exposes the method `getContextItemStack()`. This returns a `java.util.Stack`. The top item on the stack is the current context item, and below this are previous context items. Each item is represented by the interface `net.sf.saxon.om.Item`. If the item is a node, and if the node is one derived by parsing a source document with the line-numbering option enabled, then it is possible to obtain the URI and line number of this node in the original XML source.

For this to work, the code must be compiled with tracing enabled. This can be achieved by setting the option `config.setCompileWithTracing(true)` on the `Configuration` object, or equivalently by setting the property `FeatureKeys.COMPILE_WITH_TRACING` on the JAXP

`TransformerFactory`. Note that this compile-time option imposes a substantial run-time overhead, even if tracing is not switched on at run-time by providing a `TraceListener`.

The saxon:suppress-indentation attribute

When the output method is XML with `indent="yes"`, the `saxon:suppress-indentation` attribute may be used to suppress indentation for certain elements. The value is a whitespace-separated list of element names. The attribute follows the same conventions as `cdata-section-elements`: values specified in separate `xsl:output` or `xsl:result-document` elements are cumulative, and if the value is supplied programmatically via an API, or from the command line, then the element names are given in Clark notation, namely `{uri}local`. The effect of the attribute is to suppress indentation for the content of the specified elements, that is, no whitespace will be inserted within such elements, at any depth. The option is useful where elements contain mixed content in which whitespace is significant.

The saxon:xquery serialization method

An additional serialization method `saxon:xquery` is available. This is intended to be useful when generating an XQuery query as the output of a query or stylesheet. This method differs from the XML serialization method in that "<" and ">" characters appearing between curly braces (but not between quotes) in text nodes and attribute nodes are not escaped. The idea is to allow queries to generated, or to be written within an XML document, and processed by first serializing them with this output method, then parsing the result with the XQuery parser. For example, the document `<a>{$a &lt; '&lt;'}</a>` will serialize as `<a>{$a < '&lt;'}</a>`.

When invoking this method via an API, use the property value "{http://saxon.sf.net/}xquery".

When using this serialization method, the `omit-xml-declaration` parameter is automatically set to "yes".

User-defined serialization attributes

Any number of user-defined attributes may be defined on `xsl:output`. These attributes must have names in a non-null namespace, which must not be either the XSLT or the Saxon namespace. The value of the attribute is inserted into the `Properties` object made available to the `Receiver` handling the output; they will be ignored by the standard output methods, but can supply arbitrary information to a user-defined output method. The name of the property will be the expanded name of the attribute in JAXP format, for example "{http://my-namespace/uri}local-name", and the value will be the value as given in the stylesheet.

Extension functions

A Saxon extension function is invoked using a name such as `saxon:localname()`.

The `saxon` prefix (or whatever prefix you choose to use) must be associated with the Saxon namespace URI `http://saxon.sf.net/`.

For example, to invoke the `saxon:evaluate()` function in XSLT, write:

```
<xsl:variable name="expression"
      select="concat('child::', $param, '[', $index, ']')"/>
..
<xsl:copy-of select="saxon:evaluate($expression)"
     xmlns:saxon="http://saxon.sf.net/"/>
```

The equivalent in XQuery is:

```
declare namespace saxon="http://saxon.sf.net/";
declare variable $param as xs:string external;
declare variable $index as xs:integer external;
declare variable $expression :=
      concat('child::', $param, '[', $index, ']');
saxon:evaluate($expression)
```

The extension functions supplied with the Saxon product are as follows:

- saxon:adjust-to-civil-time(): converts an xs:dateTime to the local civil time in a named timezone
- saxon:analyze-string(): analyzes a string using a regular expression
- saxon:base64Binary-to-octets(): converts an xs:base64Binary value to a sequence of octets
- saxon:base64Binary-to-string(): converts an xs:base64Binary value to a string, given its encoding
- saxon:call(): calls a first-class function previously created using saxon:function()
- saxon:column-number(node): gets the column number of a node in the source document
- saxon:compile-query(): compiles a query that can subsequently be used as input to saxon:query()
- saxon:compile-stylesheet(): compiles a stylesheet that can subsequently be used as input to saxon:transform()
- saxon:current-mode-name(): returns the name of the current mode in XSLT
- saxon:decimal-divide(): performs decimal division with user-specified precision
- saxon:deep-equal(): compares two sequences for deep equality
- saxon:discard-document(): marks a document as being eligible for garbage collection
- saxon:eval(): evaluates a stored expression created using saxon:expression
- saxon:evaluate(): evaluates an XPath expression supplied dynamically as a string
- saxon:evaluate-node(): evaluates an XPath expression held in a node of a source document
- saxon:expression(): creates a stored expression for subsequent evaluation using saxon:eval()
- saxon:find(): finds items that match a given key value within an indexed sequence
- saxon:for-each-group(): groups a set of items on the basis of a grouping key
- saxon:format-dateTime(): formats a date, time, or dateTime value
- saxon:format-number(): formats a number for output
- saxon:function(): creates a first-class function that can be passed as an argument to other functions
- saxon:generate-id(): generates a unique ASCII identifier for a node
- saxon:get-pseudo-attribute(): parses the content of a processing instruction
- saxon:has-same-nodes(): tests whether two sequences contain the same nodes
- saxon:hexBinary-to-octets(): converts an xs:hexBinary value to a sequence of octets
- saxon:hexBinary-to-string(): converts an xs:hexBinary value to a string, given its encoding
- saxon:highest(): finds the nodes having the highest value for some expression
- saxon:index(): creates an indexed sequence, allowing efficient retrieval using a key value

- saxon:in-summer-time(): tests whether a given date/time is in summer time (daylight savings time)
- saxon:is-whole-number(): tests whether a given value has no fractional part
- saxon:item-at(): selects the item at a given position in a sequence
- saxon:last-modified(): determines when a file was last modified
- saxon:leading(): returns items in a sequence up to the first one matching a condition
- saxon:line-number(node): gets the line number of a node in the source document
- saxon:lowest(): finds the nodes having the lowest value for some expression
- saxon:namespace-node(): creates a namespace node
- saxon:stream(): evaluates an expression in streaming mode
- saxon:octets-to-base64Binary(): converts a sequence of octets to an xs:base64Binary value
- saxon:octets-to-hexBinary(): converts a sequence of octets to an xs:hexBinary value
- saxon:parse(): parses an XML document supplied as a string
- saxon:parse-html(): parses an HTML document supplied as a string
- saxon:path(): returns an XPath expression that identifies a node
- saxon:print-stack(): returns a formatted string representing the current execution stack
- saxon:query(): Runs an XQuery that was previously compiled using saxon:compile-query()
- saxon:result-document(): constructs a document and serializes it, writing the result to a file in filestore
- saxon:serialize(): returns the XML representation of a document or element, as a string
- saxon:sort(): sorts a sequence of nodes or atomic values
- saxon:string-to-base64Binary(): encodes a string to an xs:base64Binary value, using a given encoding
- saxon:string-to-hexBinary(): encodes a string to an xs:hexBinary value, using a given encoding
- saxon:string-to-utf8(): returns the UTF8 representation of a string
- saxon:system-id(): returns the system ID of the document containing the context node
- saxon:transform(): Runs an XSLT transformation
- saxon:try(): allows recovery from dynamic errors
- saxon:type-annotation(): returns the type annotation of a node or atomic value
- saxon:unparsed-entities(): returns a list of the unparsed entities declared within a document

saxon:adjust-to-civil-time()

If the input is an empty sequence, the result is an empty sequence.

Otherwise the input dateTime is adjusted to a dateTime in the civil timezone named in the second argument. This uses Olson timezone names, for example `America/New_York` or `Europe/Paris`. For example,

- `adjust-to-civil-time(xs:dateTime('2008-01-10T12:00:00Z', 'America/New_York')` returns `2008-01-10T07:00:00-05:00`
- `adjust-to-civil-time(xs:dateTime('2008-07-10T12:00:00Z', 'America/New_York')` returns `2008-07-10T08:00:00-04:00`

saxon:analyze-string()

The action of this function is analagous to the `xsl:analyze-string` instruction in XSLT 2.0. It is provided to give XQuery users access to regular expression facilities comparable to those provided in XSLT 2.0. (The function is available in XSLT also, but is unnecessary in that environment.)

The first argument defines the string to be analyzed. The second argument is the regex itself, supplied in the form of a string: it must conform to the same syntax as that defined for the standard XPath 2.0 functions such as `matches()`.

The third and fourth arguments are functions (created using saxon:function), called the matching and non-matching functions respectively. The matching function is called once for each substring of the input string that matches the regular expression; the non-matching function is called once for each substring that does not match. These functions may return any sequence. The final result of the `saxon:analyze-string` function is the result of concatenating these sequences in order.

The matching function takes two arguments. The first argument is the substring that was matched. The second argument is a sequence, containing the matched subgroups within this substring. The first item in this sequence corresponds to the value `$1` as supplied to the `replace()` function, the second item to `$2`, and so on.

The non-matching function takes a single argument, namely the substring that was not matched.

The detailed rules follow `xsl:analyze-string`. The regex must not match a zero-length string, and neither the matching nor non-matching functions will ever be called to process a zero-length string.

The following example is a "multiple match" example. It takes input like this:

```
<doc>There was a young fellow called Marlowe</doc>
```

and produces output like this:

```
<out>Th[e]r[e] was a young f[e]llow call[e]d Marlow[e]</out>
```

The XQuery code to achieve this is:

```
declare namespace f="f.uri";

declare function f:match ($c, $gps) { concat("[", $c, "]") };

declare function f:non-match ($c) { $c };

<out>
    {string-join(
        saxon:analyze-string(doc, "e",
                             saxon:function('f:match', 2),
                             saxon:function('f:non-match', 1)), "")}
</out>
```

The following example is a "single match" example. Here the regex matches the entire input, and the matching function uses the subgroups to rearrange the result. The input in this case is the document

`<doc>12 April 2004</doc>` and the output is `<doc>2004 April 12</doc>`. Here is the query:

```
declare namespace f="f.uri";

declare function f:match ($c, $gps) { string-join(($gps[3], $gps[2], $gps[1]),

declare function f:non-match ($c) { error("invalid date") };

<out> {
        saxon:analyze-string(doc, "([0-9][0-9]) ([A-Z]*) ([0-9]{4})",
                             saxon:function('f:match', 2),
                             saxon:function('f:non-match', 1), "i")}
</out>
```

This particular example could be achieved using the `replace()` function: the difference is that `saxon:analyze-string` can insert markup into the result, which `replace()` cannot do.

saxon:base64Binary-to-octets()

This function takes an `xs:base64Binary` value as input, and returns a sequence of integers representing this sequence of octets. The integers will be in the range 0-255.

saxon:base64Binary-to-string()

This function takes as input an `xs:base64Binary` value and the name of a character encoding (for example "UTF8"). It interprets the contents of the base64 value as a sequence of bytes representing a character string in a particular encoding, and returns the corresponding string.

For example, the call `saxon:base64Binary-to-string(xs:base64Binary("RGFzc2Vs"), "UTF8")` returns the string "Dassel".

saxon:call()

This function allows a first-class function created using `saxon:function()` to be invoked. The function (that is, the value returned by the call on `saxon:function()`) is passed as the first argument, and the parameters to be supplied to that function are passed as the second and subsequent arguments. The value returned by the target function is returned.

A call to `saxon:call` is typically contained within a , which accepts a parameter indicating the function to be called.

For examples of how this function is used, see saxon:function.

saxon:column-number(node)

This function returns the column number of a selected element within the XML document (or external entity) that contains it. If the argument is supplied, it must be a node; if the argument is omitted, the context item is used, in which case the context item must be a node. If line and column numbers are not maintained for the current document, the function returns -1.

To ensure that line and column numbers are maintained, use the -l (letter ell) option on the command line.

Note that the value returned is dependent on information supplied by the XML parser. For an element node, SAX parsers generally report the line and column position of the ">" character at the end of the start tag. StAX parsers by contrast report the position of the "<" character at the start of the start tag. SAX parsers report line and column numbers only for element nodes, so for any other kind of node, the returned value will be -1.

See also saxon:line-number().

saxon:compile-query()

This function takes as input a string containing an XQuery query, and produces as output a compiled query suitable for use with the saxon:query() extension function.

The static context for the query (for example, its namespace bindings and its base URI) must be defined within its own query prolog. It does not inherit any context values from the query or stylesheet in which the `saxon:compile-query()` function is called. The query cannot access variables or functions defined in the containing query or stylesheet.

There are several ways the string containing the query might be constructed. It can be built directly as a dynamic string in the calling application. In the case of XSLT, it can be read from a file using the `unparsed-text()` function. It can also be generated by calling saxon:serialize() on an XML representation of the query, using the serialization method saxon:xquery

The compiled query can be evaluated (repeatedly) using the saxon:query() extension function.

saxon:compile-stylesheet()

This function takes as input a document containing an XSLT stylesheet, and produces as output a compiled stylesheet suitable for use with the saxon:transform() extension function.

The document node can be supplied as a call on the `doc()` function to read the stylesheet from filestore (or from a remote URL), or it can be a variable containing a stylesheet that has been constructed programmatically. If the document contains any `xsl:include` or `xsl:import` declarations these will be resolved in the usual way (relative to the base URI of the element that contains them).

saxon:current-mode-name()

This function, designed for use in XSLT, returns the name of the current mode. If there is no current mode, or if the current mode is the default (unnamed) mode, it returns an empty sequence.

The function is useful where a template belongs to multiple modes, in that it allows the template at run-time to determine in which mode it was activated.

saxon:decimal-divide()

This performs a decimal division to a user-specified precision. The value of the first argument is divided by the second argument, and the result is returned to the number of decimal places indicated by the third argument. The exact result is rounded towards zero. For example, `decimal-divide(100, 30, 2)` returns 0.33. (The default for decimal division in Saxon using the `div` operator is to return decimal places in the result, where is the scale of the first operand and the scale of the second.)

saxon:deep-equal()

This function compares two sequences $seq1 and $seq2 for deep equality. The two sequences are supplied in the first two arguments. In the absence of any $flags, the function returns the same result as the `deep-equal` function in the standard XPath library. The $flags argument is used to modify the way in which the comparison is performed.

The $collation argument is mandatory. Supply an empty sequence to use the default collation.

The flags argument is a string containing characters acting as flags that cause the function to behave differently from the standard `fn:deep-equal()` function. The following flags are defined:

Table 12.2.

N	Include namespace nodes in the comparison. For two elements to be deep-equal, they must have the same in-scope namespaces (that is, same prefix and same URI).
F	Include namespace prefixes in the comparison. For two elements or attributes to be equal, their names must use the same namespace prefix (or none).
C	Include comment nodes in the comparison. For two element or document nodes to be deep-equal, they must have the same comment node children.
P	Include processing-instruction nodes in the comparison. For two element or document nodes to be deep-equal, they must have the same processing-instruction node children.
J	Join adjacent text nodes (for example, nodes either side of an ignored comment)
S	Compare string values rather than typed values of simple-typed elements and attributes.
A	Compare type annotations on elements and attributes. The type annotations must match exactly.
I	Compare the is-ID and is-IDREF properties on elements and attributes.
w	Exclude whitespace text nodes from the comparison. Any whitespace text node in either tree is ignored (except when determining the typed value of an element annotated with a simple type or a complex type with simple content).
?	Explain reason for a non-match. If the result is not-equal, a warning message explaining the reason will be sent to the ErrorListener. (In general, a sequence of warning messages will be sent, starting with the lowest-level difference and moving up the tree).

saxon:discard-document()

This function removes a document from Saxon's internal document pool. The document remains in memory for the time being, but will be released from memory by the Java garbage collector when all references to nodes in the document tree have gone out of scope. This has the benefit of releasing memory, but the drawback is that if the same document is loaded again during the same transformation, it will be reparsed from the source text, and different node identifiers will be allocated. The function returns the document node that was supplied as an argument, allowing it to be used in a call such as `select="saxon:discard-document(document('a.xml'))"`.

saxon:eval()

This function returns the result of evaluating the supplied stored expression. A stored expression may be obtained as the result of calling the saxon:expression function.

The stored expression is evaluated in the current dynamic context, that is, the context node is the current node, and the context position and context size are the same as the result of calling position() or last() respectively.

The second and subsequent arguments to `saxon:eval` supply values for the variables `$p1`, `$p2`, etc within the stored expression. For details see saxon:expression.

saxon:evaluate()

This function allows XPath expressions to be constructed and evaluated dynamically at runtime.

> Note that an `xsl:evaluate` instruction with similar functionality is available as a standard feature in XSLT 3.0.

The supplied string must contain an XPath expression. The result of the function is the result of evaluating the XPath expression. This is useful where an expression needs to be constructed at run-time or passed to the stylesheet as a parameter, for example where a sort key is determined dynamically.

The static context for the expression includes all the in-scope namespaces, types, and functions from the calling stylesheet or query. It does include any variables from the calling environment. The base URI and default function namespace are inherited from the calling environment. The default namespace for elements and types is taken from the value of the `xpath-default-namespace` attribute in the stylesheet, if present.

The expression may contain references to variables `$p1`, `$p2`, etc., and the values of these variables may be supplied in the second, third, and subsequent arguments to the `saxon:evaluate()` call.

The function `saxon:evaluate(string)` is shorthand for `saxon:eval(saxon:expression(string))`. For the rules governing what may and may not appear in the expression, see saxon:expression.

See also saxon:evaluate-node, which is a similar function intended for evaluating XPath expressions contained in a source document.

saxon:evaluate-node()

This function allows XPath expressions to be read from a source document and evaluated at run-time.

> Note that an `xsl:evaluate` instruction with similar functionality is available as a standard feature in XSLT 3.0.

The supplied argument must be a node, and the string-value of the node must contain an XPath expression. The result of the function is the result of evaluating this XPath expression. This is useful where XPath expressions are held in source documents, for example to parameterize the calculations performed by a query or stylesheet, or to provide XPointer-like cross-references within a document.

The static context for the expression takes the in-scope namespaces and the base URI from the node containing the expression (or from its parent element, if it is an attribute or text node). The default namespace defined using `xmlns="xyz.uri"` is used as the default namespace for elements and types, so that unprefixed names in path expressions (for example `//data/value`) refer to unprefixed elements in the containing document.

The expression cannot refer to any variables defined outside the expression itself, and it cannot refer to user-defined types or user-defined functions. However, it has access to the standard function library (in the default function namespace) and to Saxon extension functions and Java methods.

In the dynamic evaluation context, the context item is the node supplied as the first argument, while the context position and size are both set to one. This means that the expression can be a relative path expression referring to nodes reachable from the node that contains the expression. (Note that if this node is an attribute, the context node is the attribute node itself, not the containing element.)

For example, given the following source document:

```
<doc xmlns:alpha="http://www.alpha.com/">
  <data>23</data>
  <exp>preceding-sibling::alpha:data + 5</exp>
</doc>
```

the expression `saxon:evaluate-node(//alpha:exp)` returns `28`.

See also saxon:evaluate, which is a similar function intended for evaluating XPath expressions constructed dynamically by code in a stylesheet or query.

saxon:expression()

This function creates a stored expression that can be evaluated repeatedly with different argument values and a different dynamic context.

The supplied string must contain an XPath expression. The result of the function is a , which may be supplied as an argument to other extension functions such as saxon:eval. The result of the expression will usually depend on the current node. The context for the expression includes the namespaces in scope at this point in the stylesheet. The expression may contain references to the nine variables `$p1`, `$p2`, ... `$p9` only. It may contain calls on Java extension functions, including Saxon and EXSLT-defined functions, as well as user-defined function declared within the containing query or stylesheet. But it does not allow access to stylesheet variables, or functions defined in the XSLT specification such as `key()` or `format-number()`.

If the second argument is present, its value must be a single element node. The in-scope namespace bindings of this element node be used to resolve any namespace prefixes present in the XPath expression. The default namespace for the element is also used as the default namespace for elements and types within the XPath expression. In addition, standard namespace bindings are automatically available for the prefixes xml, xs, xsi, fn, and saxon.

If the second argument is omitted, then the namespace context for the expression is taken from the stylesheet. (This is also the rule used by `saxon:evaluate()`.) If the expression contains namespace prefixes, these are interpreted in terms of the namespace declarations in scope at the point where the `saxon:expression()` function is called, not those in scope where the stored expression is evaluated.

The stored expression (if it is to be evaluated using `saxon:eval()`) may contain references to variables named `$p1`, `$p2`, ... `$p9`. The values of these variables can be supplied when the expression is evaluated using `saxon:eval`. The second argument of `saxon:eval` supplies the value of `$p1`, the third argument supplies the value of `$p2`, and so on.

For example, following `<xsl:variable name="add" select="saxon:expression('$p1 + $p2')"/>`, the instruction `<xsl:value-of select="saxon:eval($add, 6, 7)"/>` will output `13`.

saxon:find()

The first argument must be an indexed sequence created using the function saxon:index(). This will invariably be provided in the form of a variable reference. The type of this object is `{http://saxon.sf.net/java-type}com.saxonica.expr.IndexedSequence`

If the second argument is omitted, the function returns the entire indexed sequence, that is, the value passed to the first argument of `saxon:index()`.

Otherwise, the second argument is a sequence of atomic values. The result of the function consists of all items in the indexed sequence that match one or more of these atomic values. Duplicates are not removed. The order of the result is sorted first by the position of the key value within the sequence of key values, and then by the order of the result items within the indexed sequence.

If a key value is of type `xs:untypedAtomic`, it is treated as a string. Values are matched according to the rules of the `eq` operator. This means, for example, that if the indexed values are numbers, then the key value must be supplied as a number. Supplying a value of a non-comparable type results in a type error.

For examples of use, see saxon:index().

saxon:for-each-group()

The action of this function is analagous to the `xsl:for-each-group` instruction (with a `group-by` attribute) in XSLT 2.0. It is provided to give XQuery users access to grouping facilities comparable to those provided in XSLT 2.0. (The function is available in XSLT also, but is unnecessary in that environment.)

The first argument defines the , a collection of items to be grouped. These may be any items (nodes or atomic values). The second argument is a function (created using saxon:function) that is called once for each item in the population, to calculate a grouping key for that item. The third argument is another function (also created using saxon:function) that is called once to process each group of items from the population.

Two items in the population are in the same group if they have the same value for the grouping key. Strings are compared using the default collation. If the value of the grouping key is a sequence of more than one item, then an item in the population may appear in more than one group; if it is an empty sequence, then the item will appear in no group.

The order in which the groups are processed is subject to change: at present it is the same as the default order in `xsl:for-each-group`, namely order of first appearance. There is no way to change this order; if the groups need to be sorted then it is best to sort the output afterwards. Each group is passed as an argument to a call on the function supplied as the third argument; the values returned by these calls form the result of the `saxon:for-each-group` call. The items within each group are in their original order (population order).

The following example groups cities by country. It takes input like this:

```
<doc>
<city name="Paris" country="France"/>
<city name="Madrid" country="Spain"/>
```

```
<city name="Vienna" country="Austria"/>
<city name="Barcelona" country="Spain"/>
<city name="Salzburg" country="Austria"/>
<city name="Bonn" country="Germany"/>
<city name="Lyon" country="France"/>
<city name="Hannover" country="Germany"/>
<city name="Calais" country="France"/>
<city name="Berlin" country="Germany"/>
</doc>
```

and produces output like this:

```
<out>
   <country leading="Paris" size="3" name="France">
      <city name="Calais"/>
      <city name="Lyon"/>
      <city name="Paris"/>
   </country>
   <country leading="Madrid" size="2" name="Spain">
      <city name="Barcelona"/>
      <city name="Madrid"/>
   </country>
   <country leading="Vienna" size="2" name="Austria">
      <city name="Salzburg"/>
      <city name="Vienna"/>
   </country>
   <country leading="Bonn" size="3" name="Germany">
      <city name="Berlin"/>
      <city name="Bonn"/>
      <city name="Hannover"/>
   </country>
</out>
```

The XQuery code to achieve this is:

```
declare namespace f="f.uri";

(: Test saxon:for-each-group extension function :)

declare function f:get-country ($c) { $c/@country };

declare function f:put-country ($group) {
    <country name="{$group[1]/@country}" leading="{$group[1]/@name}" size="{cou
       {for $g in $group
           order by $g/@name
           return <city>{ $g/@name }</city>
       }
    </country>
};

<out>
    {saxon:for-each-group(/*/city,
                          saxon:function('f:get-country', 1),
                          saxon:function('f:put-country', 1))}
</out>
```

saxon:format-dateTime()

These functions are identical to the XSLT functions of the same name, but are made available in the Saxon namespace for the benefit of XQuery users. For full details, see the current draft of the XSLT 2.0 specification [http://www.w3.org/TR/xslt20/].

For example, the call `saxon:format-date(current-date(), '[D1o] [MNn], [Y0001]')` might produce the output `3rd September, 2004`.

saxon:format-number()

This function is identical to the two-argument version of the `format-number` function in XSLT 2.0, but in the Saxon namespace. It uses the default behavior of the XSLT function when there is no `xsl:decimal-format` declaration in the stylesheet, which means that it is not possible to change the characters used as the decimal point, grouping separator, and so on.

For example, the call `saxon:format-number(123.4567, '0000.00')` produces the output `0123.46`.

For full details, see the XSLT 2.0 specification [http://www.w3.org/TR/xslt20/].

saxon:function()

From Saxon 9.2, provided XQuery 3.0 support is enabled and the query prolog specifies version "3.0", the syntax `my:function#3` can be used in place of the call `saxon:function('my:function', 3)`. The implementation (with either syntax) has also been extended so that it works with all functions, not only with user-written functions.

This function takes as its arguments the name and arity of a function, and returns a value that represents the function and can be used to invoke the function using saxon:call. This allows higher-order functions to be implemented in XSLT and XQuery, that is, functions that take other functions as arguments. An example of such a higher-order function is saxon:for-each-group, which provides grouping capability in XQuery similar to that of the `xsl:for-each-group` instruction in XSLT.

The arguments must be specified as literals (this function is always evaluated at compile time). The first argument gives the name of the function as a lexical QName (using the default function namespace if unprefixed), the second gives the function arity (number of arguments).

Here is an example, the textbook `fold` function in XSLT:

```
<xsl:function name="f:fold">
  <xsl:param name="sequence"/>
  <xsl:param name="operation"/>
  <xsl:param name="start-value"/>
  <xsl:sequence select="if (empty($sequence))
                        then $start-value
                        else f:fold(remove($sequence, 1),
                                    $operation,
                                    saxon:call($operation,
                                                     $start-value,
                                                     $sequence[1])"/>
</xsl:function>

<xsl:function name="f:plus">
  <xsl:param name="a"/>
  <xsl:param name="b"/>
  <xsl:sequence select="$a + $b"/>
</xsl:function>

<xsl:function name="f:times">
  <xsl:param name="a"/>
  <xsl:param name="b"/>
  <xsl:sequence select="$a * $b"/>
</xsl:function>

<xsl:function name="f:sum">
  <xsl:param name="sequence"/>
  <xsl:sequence select="f:fold($sequence, saxon:function('f:plus', 2), 0)"/>
</xsl:function>

<xsl:function name="f:product">
  <xsl:param name="sequence"/>
  <xsl:sequence select="f:fold($sequence, saxon:function('f:times', 2), 0)"/>
</xsl:function>
```

Here is the same example in XQuery (using XQuery 3.0 syntax):

```
xquery version "3.0";
declare function f:fold (
         $sequence as xs:double*, $operation, $start-value as xs:double) {
     if (empty($sequence))
     then $start-value
     else f:fold(remove($sequence, 1),
                 $operation,
                 saxon:call($operation, $start-value, $sequence[1]))
};

declare function f:plus ($a as xs:double, $b as xs:double) {$a + $b};

declare function f:times ($a as xs:double, $b as xs:double) {$a * $b};

declare function f:sum ($sequence as xs:double*) as xs:double {
   f:fold($sequence, f:plus#2, 0)
};

declare function f:product ($sequence as xs:double*) as xs:double {
   f:fold($sequence, f:times#2, 1)
};
```

The result of `f:sum(1 to 4)` is 10, while the result of `f:product(1 to 4)` is 24.

Higher-order functions allow many generic functions such as `fold` to be written, and their availability in Saxon-EE turns XSLT and XQuery into fully-fledged functional programming languages.

The type of the result of saxon:function is `function()`, a new type introduced in XQuery 1.1 - it is a third subtype of `item()` alongside nodes and atomic values.

saxon:generate-id()

This function returns a string that acts as a unique identifier for the supplied node. The identifier will always consist of ASCII letters and digits and will start with a letter.

The function is identical to the XSLT generate-id() function. It is provided as an extension function in the Saxon namespace to make it available in contexts other than XSLT, for example in XQuery.

saxon:get-pseudo-attribute()

This function parses the contents of a processing instruction whose content follows the conventional attribute="value" structure (as defined for the <?xsl-stylesheet?> processing instruction). The context should be a processing instruction node; the function returns the value of the pseudo-attribute named in the first argument if it is present, or an empty string otherwise.

If the attribute value contains a sequence of characters in the form of an XML character reference, for example "&#x0A;", this is parsed and converted into the corresponding character.

saxon:has-same-nodes()

This function returns a boolean that is true if and only if $nodes-1 and $nodes-2 contain the same sequence of nodes in the same order. Nodes are compared by identity, not by content. Note this is quite different from the "=" operator, which tests whether there is a pair of nodes with the same string-value, and the deep-equal() function, which compares nodes pairwise but by content.

saxon:hexBinary-to-octets()

This function takes an `xs:hexBinary` value as input, and returns a sequence of integers representing this sequence of octets. The integers will be in the range 0-255.

saxon:hexBinary-to-string()

This function takes as input an `xs:hexBinary` value and the name of a character encoding (for example "UTF8"). It interprets the contents of the hexBinary value as a sequence of bytes representing a character string in a particular encoding, and returns the corresponding string.

saxon:highest()

This function returns the item or items from the input sequence that have the highest value for the function `$key`. If the second argument is omitted, it defaults to the function `fn:data#1`, that is, it atomizes the item from the input sequence.

In XQuery 3.0, the function may be obtained in a variety of ways. In other environments, it may be obtained by calling the saxon:function() extension function.

The `$key` function is evaluated for each item in `$input` in turn, with that item supplied as the parameter to the function. Any NaN values are ignored. If the input sequence is empty, the result is an empty sequence. If several items have the highest value, the result sequence contains them all.

Example: saxon:highest(sale, function($x){$x/@price * $x/@qty'}) will evaluate price times quantity for each child `<sale>` element, and return the element or elements for which this has the highest value.

saxon:index()

The first argument is any sequence. Usually it will be a sequence of nodes, but this is not essential. This is the sequence being indexed.

The second argument is a compiled XPath expression. Most commonly, the argument will be written as a call to the saxon:expression() extension function. This expression is evaluated once for each item in the sequence being indexed, with that item as the context node. (The context position and size reflect the position of this item in the sequence, but this will not normally be useful.) The result of the expression is atomized. Each value in the atomized result represents a key value: the item in the indexed sequence can be efficiently found using any of these key values.

If a key value is of type `xs:untypedAtomic`, it is treated as a string. If you want to treat the value as numeric, say, then perform a conversion within the expression.

The optional third argument is the URI of a collation to be used when comparing strings. For example, if you want string matching to be accent- and case-blind, specify `"http://saxon.sf.net/collation?strength=primary"`.

The result is an object of type `{http://saxon.sf.net/java-type}com.saxonica.expr.IndexedSequence`, that can be supplied as input to the `saxon:find()` function.

For example, consider a source document of the form:

```
<doc>
  <town name="Amherst" state="NH"/>
  <town name="Amherst" state="MA"/>
  <town name="Auburn" state="MA"/>
  <town name="Auburn" state="NH"/>
  <town name="Auburn" state="ME"/>
  <town name="Bristol" state="RI"/>
  <town name="Bristol" state="ME"/>
  <town name="Bristol" state="CT"/>
  <town name="Bristol" state="NH"/>
  <town name="Bristol" state="VT"/>
  <town name="Cambridge" state="ME"/>
 </doc>
```

and suppose there is a requirement to find `town` elements efficiently given the abbreviation for the `state`. You can do this by first setting up an indexed sequence. In XQuery you can write:

```
declare namespace saxon="http://saxon.sf.net/";
 declare namespace java="http://saxon.sf.net/java-type";
 declare variable $indexedTowns
    as java:com.saxonica.expr.IndexedSequence
    := saxon:index(//town, saxon:expression("@state"));
```

This could be a local variable (declared in a `let` clause) rather than a global variable. The XSLT equivalent is:

```
<xsl:variable name="indexedTowns"
           select="saxon:index(//town, saxon:expression('@state'))"
           as="java:com.saxonica.expr.IndexedSequence"/>
```

You can then find all the towns in New Hampshire using the expression:

```
saxon:find($indexedTowns, "NH")
```

Indexed sequences are primarily useful in XQuery, where they provide functionality equivalent to the standard `xsl:key` mechanism in XSLT. There are some cases, however, where indexed sequences can also be useful in XSLT. One example is where there is a need for an index to span multiple documents: the XSLT `key()` function will only search within a single document.

An indexed sequence can only be used in the first argument to the `saxon:find()` function. If you want access to the sequence that was passed as the first argument to `saxon:index()`, you can get this by calling `saxon:find()` with a single argument.

See also: saxon:find().

saxon:in-summer-time()

The `$region` argument may either be an ISO two-letter country code (for example "de" or "es"), or an Olsen timezone name (such as "America/New_York" or "Europe/Lisbon"). The function returns true if the given date/time is in summer time (daylight savings time) in that country or timezone, as far as can be determined from the Java timezone database. If the information is not available the function either returns false or an empty sequence.

If a country code is specified for a country that spans different timezones with different daylight savings rules (for example, the US), then one timezone in that country is chosen arbitrarily.

For reliable results, the supplied date/time should include a timezone. This does not need to correspond to the timezone named in the second argument; it is there purely to ensure that `$date` represents a single point in time unambigously.

saxon:is-whole-number()

This function takes a number as input and returns true if the number has no fractional part, that is, if it is equal to some integer. If an empty sequence is supplied, the function returns false.

Example:

```
saxon:is-whole-number(12e0)
```

will return true.

saxon:item-at()

This function returns the item at a given position in a sequence. The index counts from one. If the index is an empty sequence, or less than one, or not a whole number, or greater than the length of the sequence, the result is an empty sequence.

Example:

```
saxon:item-at(10 to 20, 8)
```

will return 17.

saxon:last-modified()

> The specification of this function has changed in Saxon 9.2. To find the last-modified date of the file from which a given node was loaded, use `saxon:last-modified(document-uri($node))`

This function returns the date and time when a file was modified.

If the argument is an empty sequence, the result is an empty sequence.

Otherwise the argument must be a valid URI. If it is a relative URI, this will be resolved against the base URI from the static context. The function has been tested with URIs using the `file` and `http` schemes, and is not guaranteed to give a result with other URI schemes (or indeed with these, under all circumstances).

The function returns an `xs:dateTime` value which will usually be in a specific timezone. The result can therefore be formatted using the `format-dateTime` function, or input to arithmetic and comparisons against other dates and times.

If the URI is not syntactically valid, error FODC0005 is thrown. If the resource identified by the URI cannot be retrieved, error FODC0002 is thrown. If the resource is successfully retrieved but no date/ time information is available, the function returns an empty sequence.

Example:

```
format-dateTime(file-last-modified(resolve-uri('lookup.xml', static-base-uri())
```

saxon:leading()

This function returns a sequence containing all those nodes from `$input` up to and excluding the first one (in order of the input sequence) for which $test has an effective boolean value of false. A function may be obtained in XQuery 1.1 using any of the new mechanisms for constructing functions; in other environments, it may be obtained by calling saxon:function.

The `$test` argument defaults to the function `fn:data#1`, that is, it atomizes the item.

The `$test` function is evaluated for item in $input in turn, with that item supplied as the argument to the function. The result consists of that leading sequence of items for which the effective boolean value of the function is true.

XQuery 1.1 example:

```
saxon:leading(following-sibling::*, function($x){$x/self::para})
```

XSLT 2.0 example:

```
saxon:leading(following-sibling::*, saxon:function('f:is-para', 1)
```

where `f:is-para` is a user-defined function that tests whether the supplied item is a `<para>` element

This will return the `<para>` elements following the current node, stopping at the first element that is not a `<para>`.

saxon:line-number(node)

This function returns the line number of a selected node within the XML document (or external entity) that contains it. If the argument is supplied, it must be a node; if the argument is omitted, the context item is used, in which case the context item must be a node. If line numbers are not maintained for the current document, the function returns -1.

To ensure that line numbers are maintained, use the -l (letter ell) option on the command line.

Note that the value returned is dependent on information supplied by the XML parser. For an element node, SAX parsers generally report the line and column position of the ">" character at the end of the start tag. StAX parsers by contrast report the position of the "<" character at the start of the start tag. SAX parsers report line and column numbers only for element nodes, so for any other kind of node, the returned value will be -1.

From release 9.0, the -l option also causes line numbers to be copied from a source document to a result document when the `xsl:copy-of` instruction is applied to a document or element node. For elements created using other instructions, the line number will reflect the position of the instruction in the stylesheet or query that caused the element to be created.

See also saxon:column-number().

saxon:lowest()

This function returns the item or items from the input sequence that have the lowest value for the function `$key`. If the second argument is omitted, it defaults to the function `fn:data#1`, that is, it atomizes the item from the input sequence.

In XQuery 3.0, the function may be obtained in a variety of ways. In other environments, it may be obtained by calling the saxon:function() extension function.

The `$key` function is evaluated for each item in `$input` in turn, with that item supplied as the parameter to the function. Any NaN values are ignored. If the input sequence is empty, the result is an empty sequence. If several items have the lowest value, the result sequence contains them all.

Example: saxon:lowest(sale, function($x){$x/@price * $x/@qty'}) will evaluate price times quantity for each child `<sale>` element, and return the element or elements for which this has the lowest value.

saxon:namespace-node()

This function creates a new namespace node. The first argument gives the name of the namespace node (that is, the namespace prefix), while the second gives the namespace URI. The prefix may be "" to create a default namespace; otherwise it must be a valid NCName. The URI must not be the empty string.

The function serves the same role in XQuery as the `xsl:namespace` instruction in XSLT 2.0: it allows a namespace in the result document to be computed dynamically.

The namespace node is typically used as part of the computed content of a constructed element node. For example:

```
declare namespace saxon="http://saxon.sf.net/";
<a xsi:type="my:decimal">
  { saxon:namespace-node("my", "http://my.uri/"), 12.4 }
</a>
```

produces the output:

```
<a xmlns:my="http://my.uri/"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="my:decimal">12.4</a>
```

In this case the namespace node could equally well be created by adding the namespace declaration `xmlns:my="http://my.uri/"` to the direct element constructor for the <a> element. But this is not always possible, for example (a) if the element name is not known statically, or (b) if the namespace URI is not known statically, or (c) if the decision whether or not to add the namespace node depends on input data.

saxon:stream()

Conceptually, this function returns a copy of its input. The intent, however, is to evaluate the supplied argument in "streaming mode", which allows an input document to be processed without building a tree represention of the whole document in memory. This allows much larger documents to be processed using Saxon than would otherwise be the case.

For further details see Streaming of Large Documents.

saxon:octets-to-base64Binary()

This function takes a sequence of integers as input, and treats them as octets. The integers must therefore all be in the range 0-255. The result of the function is the `xs:base64Binary` value representing this sequence of octets.

saxon:octets-to-hexBinary()

This function takes a sequence of integers as input, and treats them as octets. The integers must therefore all be in the range 0-255. The result of the function is the `xs:hexBinary` value representing this sequence of octets.

saxon:parse()

This function takes a single argument, a string containing the source text of a well-formed XML document. It returns the document node (root node) that results from parsing this text. It throws an error if the text is not well-formed XML. Applications should not rely on the identity of the returned document node (at present, if the function is called twice with the same arguments, it returns a new document node each time, but this may change in future).

This function is useful where one XML document is embedded inside another using CDATA, or as an alternative way of passing an XML document as a parameter to a stylesheet.

A dynamic error is reported if the supplied string is not well-formed (and namespace-well-formed) XML. An XML declaration may be included, but is not required: if it is present, the encoding declaration will be ignored, since the document is supplied in character form rather than in binary.

The XML parser that is used will be the one nominated to the Saxon `Configuration` as the parser for source documents. Validation against schemas or DTDs, and whitespace stripping, will take place according the settings in the `Configuration` object; if called from XSLT, the `xsl:strip-space` and `xsl:preserve-space` settings in the stylesheet are also taken into account.

saxon:parse-html()

This function takes a single argument, a string containing the source text of an HTML document. It returns the document node (root node) that results from parsing this text using the TagSoup parser.

On the Java platform, the TagSoup jar file must be on the classpath. It may be downloaded from http://home.ccil.org/~cowan/XML/tagsoup/tagsoup-1.2.jar.

On the .NET platform, the code of TagSoup 1.2 is available automatically: it has been compiled into the `saxon9pe.dll` and `saxon9ee.dll` assemblies.

This function is useful where an HTML document is embedded inside another using CDATA. It can also be used in conjunction with the `unparsed-text()` function to read HTML from filestore. Note that the base URI of the document is not retained in this case.

saxon:path()

The second version of the function (with no arguments) is equivalent to supplying "." as the argument. In this case there must be a context item and it must be a node.

The function returns a string whose value is an XPath expression identifying the selected node in the source tree. This can be useful for diagnostics, or to create an XPointer value, or when generating another stylesheet to process the same document. The resulting string can be used as input to the `evaluate()` function, provided that any namespace prefixes it uses are declared.

The generated path will use lexical QNames as written in the original source document. In documents that use multiple namespaces, this may not be the most suitable representation if there is a requirement to evaluate the XPath expression later, as any prefixes it uses will need to be declared.

saxon:print-stack()

This function takes no arguments. It returns a string whose value is a formatted representation of the current state of the execution stack. This is suitable for display using `xsl:message` or the `trace()` function, or it can be inserted as a comment into the result document.

saxon:query()

This function takes as input a compiled XQuery query, and runs the query, returning the result of evaluating the query. The first argument will generally be the result of calling the saxon:compile-query() extension function.

If the first argument is an empty sequence, the result is an empty sequence.

If only one argument is supplied, the context item for evaluating the query will be the same as the context item in the environment where the function is called, that is, the implicit second argument is ".". If there is no context item, however, no failure occurs unless the query attempts to reference the context item.

If the second argument is present it can be any item, which is used as the context item for the query. It can also be the empty sequence, in which case the query runs with no context item.

If the optional third argument is present, it is used to supply parameters (external variables) to the query. The value is a sequence of nodes. Each node must be an element node, attribute node, or document node; supplying a document node is equivalent to supplying all its element children. The name of the node must match an external variable name declared in the query prolog, and the atomized value of the node is used as the value of the parameter. If this is `untypedAtomic` then it is converted to the required type declared in the query.

The function is available both in XQuery and in XSLT.

The compiled stylesheet can be used repeatedly with different inputs.

Here is an example of how to use the function from XQuery:

```
declare namespace saxon = "http://saxon.sf.net/";

<out>{
let $q1 := "declare variable $x external; declare variable $y external; <z>{$x
return saxon:query(saxon:compile-query($q1), 4, (<x>3</x>, <y>2</y>))
}</out>
```

The result of the query is a sequence containing the single integer 9.

saxon:result-document()

This function takes three arguments:

1. $href is a URI identifying the destination of the serialized result document. Generally speaking, the `file:/` URI scheme works and other schemes usually don't. If the URI is relative, it is interpreted relative to the which is the destination of the principal output of the query, and which defaults to the current directory (not the stylesheet directory). When the -t option is used on the command line, the actual destination of output files is reported as a message on System.err.
2. $content is a sequence of items which makes up the content of the new document, it is processed in the same way as the content sequence passed to the `document{}` constructor in XQuery, or the `xsl:document` instruction in XSLT.
3. $format is used to define serialization properties; its value is an `xsl:output` element conforming to the rules defined in the XSLT specification. This element may be constructed dynamically, and may therefore be used to decide all the serialization properties dynamically.

For example, the function may be used as follows in XQuery. This example has the effect of writing each `<country>` element to a separate file:

```
declare namespace saxon="http://saxon.sf.net/";
declare namespace xsl="http://www.w3.org/1999/XSL/Transform";
let $input :=
  <data>
    <country name="Austria">Vienna</country>
    <country name="France">Paris</country>
    <country name="Germany">Berlin</country>
  </data>
return
<log> {
  for $c in $input/country return
    let $href := concat($c/@name, '.xml')
    return (
      saxon:result-document($href, $c,
          <xsl:output method="xml" indent="yes"/>),
      <done href="{$href}"/>
    )
} </log>
}</out>
```

The function returns no result (technically, it returns an empty sequence). Because it is called only for its side-effects, some care is needed in how it is used. Generally, the safe approach is to call it in a position where, if it did produce output, the output would form part of the result of the query. It is implemented internally using the `xsl:result-document` instruction from XSLT, and must follow the same constraints. For example, an error will be reported if it is called while evaluating a variable or a function, or if you try to write and read the same document within a single query.

saxon:serialize()

This function takes two arguments: the first is a node (generally a document or element node) to be serialized. The second argument defines the serialization properties. The second argument takes several possible forms:

- When called within an XSLT stylesheet, the second argument may be the name of an `xsl:output` element in the stylesheet, written as a string literal (it must be a literal, or at any rate, an expression that is evaluated at compile time).
- In non-XSLT environments, the second argument may be the name of the output method (xml, html, xhtml, text), written as a string literal. In this case the other serialization parameters are defaulted.
- In all environments, the second argument may be an `xsl:output` element conforming to the rules defined in the XSLT specification. This element may be constructed dynamically, and may therefore be used to decide all the serialization properties dynamically.

For example, the function may be used as follows in XQuery:

```
declare namespace saxon="http://saxon.sf.net/";
declare namespace xsl="http://www.w3.org/1999/XSL/Transform";

<out>{
let $x := <a><b/><c>content</c><?pi?><!--comment--></a>
return saxon:serialize($x, <xsl:output method="xml"
                             omit-xml-declaration="yes"
                             indent="yes"
                             saxon:indent-spaces="1"/>)
}</out>
```

The function serializes the specified document, or the subtree rooted at the specified element, according to the parameters specified, and returns the serialized document as a string.

This function is useful where the stylesheet or query wants to manipulate the serialized output, for example by embedding it as CDATA inside another XML document, or prefixing it with a DOCTYPE declaration, or inserting it into a non-XML output file.

Note that because the output is a string, the encoding parameter has no effect on the actual encoding, though it does affect what is written to the XML declaration.

saxon:sort()

The `saxon:sort` function is provided primarily for use in XPath, which has no built-in sorting capability. In XSLT it is preferable to use `xsl:sort`, in XQuery to use a FLWOR expression with an `order by` clause.

This form of the function sorts a sequence of nodes and/or atomic values. For atomic values, the value itself is used as the sort key. For nodes, the atomized value is used as the sort key. The atomized value must be a single atomic value. The values must all be comparable. Strings are sorted using codepoint collation.

This form of the function sorts a sequence of nodes and/or atomic values, using the supplied stored expression to compute the sort key for each item in the sequence. The computed sort key must either be a single atomic value, or a node that atomizes to a single atomic value, and the sort keys must all be comparable. Strings are sorted using codepoint collation.

A stored expression may be obtained as the result of calling the saxon:expression function.

The stored expression is evaluated for each item in $seq in turn, with that item as the context node, with the context position equal to the position of that item in $seq, and with the context size equal to the size of $seq.

Example: saxon:sort(sale, saxon:expression('@price * @qty')) will evaluate price times quantity for each child <sale> element, and return the `sale` elements in ascending numeric order of this value.

saxon:string-to-base64Binary()

This function takes as input a string and the name of a character encoding (for example "UTF8"). It encodes the contents of the string value as a sequence of bytes using a particular encoding, and returns the `xs:base64Binary` representation of this sequence of bytes.

For example, the call `saxon:string-to-base64Binary("Dassel", "UTF8")` returns the `xs:base64Binary` value whose lexical representation is `"RGFzc2Vs"`.

saxon:string-to-hexBinary()

This function takes as input a string and the name of a character encoding (for example "UTF8"). It encodes the contents of the string value as a sequence of bytes using a particular encoding, and returns the `xs:hexBinary` representation of this sequence of bytes.

saxon:string-to-utf8()

This function takes a string as input, and returns a sequence of integers representing the octets in the UTF8 encoding of the string. The integers are all in the range 0-255.

saxon:system-id()

This returns the system identifier (URI) of the entity in the source document that contains the context node. There are no arguments.

saxon:transform()

This function takes as input a compiled XSLT stylesheet, and uses it to transform a source document into a result document. The first argument will generally be the result of calling the saxon:compile-stylesheet() extension function. The second argument can be any node (usually a document node or element node), either read from external filestore using the `doc()` or `document()` function, or constructed programmatically.

The function is available both in XQuery and in XSLT. It can thus be used for example in XQuery to pre-process the input using a stylesheet, or to post-process the output using a stylesheet. It can also be used to chain multiple XSLT transformations together.

The compiled stylesheet can be used repeatedly to transform multiple source documents.

If the optional third argument is present, it is used to supply parameters to the transformation. The value is a sequence of nodes. Each node must be an element node, attribute node, or document node; supplying a document node is equivalent to supplying all its element children. The name of the node must match the parameter name declared in an `xsl:param` element in the stylesheet, and the atomized value of the node is used as the value of the parameter. If this is `untypedAtomic` then it is converted to the required type declared in the stylesheet.

The stylesheet may contain calls on `xsl:result-document`. This allows the output of the stylesheet to be serialized directly to filestore rather than being returned to the calling transformation or query.

Here is an example of how to use the function from XQuery:

```
let $results :=
  <customers>{ //customer[location="Scotland"] }</customers>
let $rendition := saxon:compile-stylesheet(doc('show-customers.xsl'))
return saxon:transform($rendition, $results)
```

The following example uses XSLT's ability to create multiple output files:

```
let $splitter := saxon:compile-stylesheet(
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                  version="2.0">
    <xsl:template match="customer">
      <xsl:result-document href="{{@id}}.xml">
        <xsl:copy-of select="."/>
      </xsl:result-document>
    </xsl:template>
  </xsl:stylesheet>)
let $results :=
  document {
     <customers>{ //customer[location="Scotland"] }</customers>
  }
return saxon:transform($splitter, $results)
```

Note (a) the need to double the curly braces to ensure that the contained expression is expanded when the stylesheet is executed by XSLT, not when it is created by XQuery, and (b) the fact that the stylesheet and source document are supplied as document nodes, not element nodes.

saxon:try()

This function provides a simple way of recovering from dynamic errors (including type errors).

This returns the value of the first argument, unless evaluation of the first argument fails. If a failure occurs, then the second argument is evaluated and its value is returned.

Here is an example showing the simpler form of call: the expression `saxon:try(1 div 0, "divide by zero")` returns the string "divide by zero".

To recover from failures occurring in XSLT instructions, for example schema validation errors, wrap the instructions in a stylesheet function (`xsl:function`) and call this function within a call of `saxon:try()`.

> The ability to supply a function value as the second argument is withdrawn in Saxon 9.2.

saxon:type-annotation()

This function takes an item as argument.

If the argument is an element or attribute node, the function returns the type annotation of the node, as a QName. If the type is anonymous, this will be a system-generated internal name.

If the argument is a document node, the function returns `xs:anyType` if the document has been schema-validated, or `xs:untyped` otherwise.

For a text node, the function returns `xs:untypedAtomic`, and for comment, processing-instruction, and namespace nodes, it returns `xs:string`.

If the argument is an atomic value, the function returns the type label of the atomic value, again as a QName.

The prefix of the returned QName should not be treated as significant.

saxon:unparsed-entities()

This function returns a list of the names of the unparsed entities declared with the document node supplied as $doc. These names can then be used as arguments to the standard XSLT functions `unparsed-entity-uri()` and `unparsed-entity-public-id()` to determine the system and public identifiers of the unparsed entity.

The Map Extension

This is a family of extension functions, `map:new()`, `map:put()`, `map:get()`, and `map:keys()` that can be used to maintain a general purpose map from atomic values to arbitrary XDM sequences. The functions are in namespace `http://ns.saxonica.com/map`, and are available in Saxon-PE and Saxon-EE only.

The map itself is an object of type `javatype:com.saxonica.functions.map.ImmutableMap` [Javadoc: `com.saxonica.functions.map.ImmutableMap`], where the prefix `javatype` corresponds to the namespace URI `http://saxon.sf.net/java-type`.

The map is immutable: adding an entry to a map creates a new map, leaving the original map unchanged. These are therefore pure functions. Under the hood, the implementation avoids copying data whereever possible to minimise the use of memory when a map is built incrementally.

The individual methods are described below:

Creating a new map

This method creates a new empty map

Adding a value to the map

This method creates and returns a new map that differs from the supplied map by adding or replacing a single entry. The key for the new entry is an atomic value supplied as `$key`, the value is supplied as `$value`. The new entry is added to the map, replacing any existing entry for the same key. Adding an entry whose value is the empty sequence is equivalent to removing the entry from the map.

Getting a value from the map

This method locates the entry in the map for the given key, if there is one, and returns it. If there is no entry, it returns the empty sequence. Keys are compared using the XPath `eq` operator, except that no error occurs in the case of incomparable types; the collation used is the Unicode codepoint collation.

Example

This example creates a map reflecting the contents of an input file, and then uses it to perform a look-up.

```
<xsl:stylesheet ... xmlns:map="http://ns.saxonica.com/map">

<xsl:variable name="transaction-map" as="javatype:com.saxonica.functions.map.Imm
    xmlns:javatype="http://saxon.sf.net/java-type">
  <xsl:param name="transactions" as="element(transaction)*"/>
  <xsl:iterate select="doc('transactions.xml')/*/transaction">
    <xsl:param name="map" select="map:new()"/>
    <xsl:next-iteration>
        <xsl:with-param name="map" select="map:put($map, @date, @value)"/>
    </xsl:next-iteration>
    <xsl:on-completion>
        <xsl:sequence select="$map"/>
    </xsl:on-completion>
  </xsl:iterate>
</xsl:variable>

<xsl:variable name="latest-transaction" select="map:get($transaction-map, strin

</xsl:stylesheet>
```

Extension instructions

A Saxon extension instruction is invoked using a name such as `<saxon:localname>`.

The `saxon` prefix (or whatever prefix you choose to use) must be associated with the Saxon namespace URI `http://saxon.sf.net/`. The prefix must also be designated as an extension element prefix by including it in the `extension-element-prefixes` attribute on the `xsl:stylesheet` element, or the `xsl:extension-element-prefixes` attribute on any enclosing literal result element or extension element.

However, top-level elements such as `saxon:collation` and `saxon:script` can be used without designating the prefix as an extension element prefix.

The extension instructions and declarations are:

- saxon:assign: assigns a new value to a global variable

- saxon:break: breaks out of a saxon:iterate loop
- saxon:call-template: calls a template whose name is decided at run-time
- saxon:catch: used within saxon:try to catch dynamic errors
- saxon:collation: names and describes a collating sequence
- saxon:continue: continues execution of a saxon:iterate loop
- saxon:doctype: constructs a serialized DOCTYPE declaration
- saxon:entity-ref: creates an entity reference in the serialized output
- saxon:finally: continues execution of a saxon:iterate loop
- saxon:import-query: imports functions from an XQuery library module
- saxon:iterate: iterates over a sequence in order, allowing parameters to be set
- saxon:mode: declares properties of a mode
- saxon:script: declares an extension function
- saxon:try: evaluates an expression with recovery from dynamic errors
- saxon:while: iterates until a condition becomes false

saxon:assign

The `saxon:assign` instruction is used to change the value of a global variable that has previously been declared using `xsl:variable` (or `xsl:param`). The variable or parameter must be marked as assignable by including the extra attribute `saxon:assignable="yes"`

As with `xsl:variable`, the name of the variable is given in the mandatory attribute, and the new value may be given either by an expression in the `select` attribute, or by expanding the content of the `xsl:assign` element.

If the `xsl:variable` element has an `as` attribute, then the value is converted to the required type of the variable in the usual way.

Example:

```
<xsl:variable name="i" select="0" saxon:assignable="yes"/>
<xsl:template name="loop">
  <saxon:while test="$i &lt; 10">
    The value of i is <xsl:value-of select="$i"/>
    <saxon:assign name="i" select="$i+1"/>
  </saxon:while>
</xsl:template>
```

The `saxon:assign` element itself does not allow an `as` attribute. Instead, it calculates the value of the variable as if `as="item()*"` were specified. This means that the result of the construct:

```
<saxon:assign name="a">London</saxon:assign>
```

is a single text node, not a document node containing a text node. If you want to create a document node, use `xsl:document`.

Using `saxon:assign` is cheating. XSLT is designed as a language that is free of side-effects, which is why variables are not assignable. Once assignment to variables is allowed, certain optimizations become impossible. At present this doesn't affect Saxon, which generally executes the stylesheet sequentially. However, there are some circumstances in which the order of execution may not be quite what you expect, in which case `saxon:assign` may show anomalous behavior. In principle the `saxon:assignable` attribute is designed to stop Saxon doing optimizations that cause such anomalies, but you can't always rely on this.

saxon:break

A `saxon:break` instruction causes early exit from a loop defined using saxon:iterate. The instruction must appear lexically within the `saxon:iterate` element.

If the `saxon:iterate` instruction takes its input from streamed document (using the `saxon:stream()` function) then executing `saxon:break` will cause the parsing and analysis of that document to terminate. This can be very useful when you only want to examine a small amount of data near the start of a large document. Note however that it means that any well-formedness or validity errors appearing later in the source document will not be detected.

For further details and examples see saxon:iterate.

saxon:call-template

The `saxon:call-template` instruction is identical to `xsl:call-template` except that the template name can be written as an attribute value template, allowing the actual template that is called to be selected at run time.

Typical usage is:

The `saxon:call-template` instruction allows `xsl:fallback` as a child element, so that fallback behaviour can be defined for other XSLT processors when they encounter this instruction.

saxon:catch

This instruction is always used together with `saxon:try`. It is used to catch dynamic errors occurring within a `saxon:try` instruction, and to return an alternative result when an error occurs.

The parent instruction must be `saxon:try`. A `saxon:try` element must have at least one `saxon:catch` child element (there may be several), and the `saxon:catch` children must come last, except perhaps for any `xsl:fallback` children.

There is a mandatory attribute `errors` defining which errors are caught. Every error code is identified by a QName. The attribute is a whitespace-separated list of `NameTests` which are used to match this QName. Each `NameTest` may be in the form `*` (match all names), `*:local` (match `local` within any namespace), `err:*` (match all names in namespace `err`, or `err:ABCD9867` (match a specific error code). The most common namespace, used by all system-defined errors, is `http://www.w3.org/2005/xqt-errors`, for which the conventional prefix is `err:`.

The value to be returned by the `saxon:catch` element may be calculated in an XPath expression in the `select` attribute, or in instructions forming the body of the `saxon:catch` element.

Within the `select` expression, or within the body of the `saxon:catch` element, the following variables are available. The namespace prefix `err` in this names must be bound to the namespace URI `http://www.w3.org/2005/xqt-errors`.

Table 12.3.

Variable	Type	Value
err:code	xs:QName	The error code

Variable	Type	Value
err:description	xs:string	A description of the error condition
err:value	item()*	Value associated with the error. For an error raised by calling the `error` function, this is the value of the third argument (if supplied). For an error raised by evaluating `xsl:message` with `terminate="yes"`, this is the document node at the root of the tree containing the XML message body.
err:module	xs:string?	The URI (or system ID) of the stylesheet module containing the instruction where the error occurred; an empty sequence if the information is not available.
err:line-number	xs:integer?	The line number within the stylesheet module of the instruction where the error occurred; an empty sequence if the information is not available.
err:column-number	xs:integer?	The column number within the stylesheet module of the instruction where the error occurred; an empty sequence if the information is not available.

The catch block can throw a new error by calling the `error()` function.

Variables declared within the `saxon:try` block are not visible within the `saxon:catch` block.

For examples, see saxon:try.

saxon:collation

Instead, you can define collations in a configuration file: see The collations element. It is also possible to specify a collation directly by using a URI of the form `http://saxon.sf.net/collation?keyword=value;keyword=value;...`. For details see Collations.

The `saxon:collation` element is a top-level element used to define collating sequences that may be used in sort keys and in functions such as `compare()`.

The collation name is a URI, and is defined in the mandatory `name` attribute. This must be a valid URI. If it is a relative URI, it is interpreted as being relative to the base URI of the `saxon:collation` element itself.

The attribute was used in earlier Saxon releases to identify the default collation. From Saxon 8.8 this attribute is ignored, with a warning. To specify the default collation, use the standard `[xsl:]default-collation` attribute.

The other attributes control how the collation is defined. These attributes have the same effect as the corresponding query parameters in a URI starting with `http://saxon.sf.net/collation?`, and are described here.

Specifically, these attributes are:

Table 12.4.

class	fully-qualified Java class name of a class that implements `java.util.Comparator`.	This parameter should not be combined with any other parameter. An instance of the requested class is created, and is used to perform the comparisons. Note that if the collation is to be used in functions such as contains() and starts-with(), this class must also be a `java.text.RuleBasedCollator`. This approach allows a user-defined collation to be implemented in Java.This option is also available on the .NET platform, but the class must implement the Java interface java.util.Comparator.
rules	details of the ordering required, using the syntax of the Java `RuleBasedCollator`	This defines exactly how individual characters are collated.This option is also available on the .NET platform, and if used will select a collation provided using the GNU Classpath implementation of `RuleBasedCollator`. At the time of writing, this is not 100% compatible with the Sun JDK implementation.
lang	any value allowed for xml:lang, for example `en-US` for US English	This is used to find the collation appropriate to a Java locale or .NET culture. The collation may be further tailored using the parameters described below.
ignore-case	yes, no	Indicates whether the upper and lower case letters are considered equivalent. Note that even when ignore-case is set to "no", case is less significant than the actual letter value, so that "XPath" and "Xpath" will appear next to each other in the sorted sequence.On the Java platform, setting ignore-case sets the collation strength to secondary.
ignore-modifiers	yes, no	Indicates whether non-spacing combining characters (such as accents and diacritical marks) are considered significant. Note that even when ignore-modifiers is set to "no", modifiers are less significant than the actual

		letter value, so that "Hofen" and "Höfen" will appear next to each other in the sorted sequence.On the Java platform, setting ignore-case sets the collation strength to primary.
ignore-symbols	yes, no	Indicates whether symbols such as whitespace characters and punctuation marks are to be ignored. This option currently has no effect on the Java platform, where such symbols are in most cases ignored by default.
ignore-width	yes, no	Indicates whether characters that differ only in width should be considered equivalent.On the Java platform, setting ignore-width sets the collation strength to tertiary.
strength	primary, secondary, tertiary, or identical	Indicates the differences that are considered significant when comparing two strings. A/B is a primary difference; A/a is a secondary difference; a/ä is a tertiary difference (though this varies by language). So if strength=primary then A=a is true; with strength=secondary then A=a is false but a=ä is true; with strength=tertiary then a=ä is false.This option should not be combined with the ignore-XXX options. The setting "primary" is equivalent to ignoring case, modifiers, and width; "secondary" is equivalent to ignoring case and width; "tertiary" ignores width only; and "identical" ignores nothing.
decomposition	none, standard, full	Indicates how the collator handles Unicode composed characters. See the JDK documentation for details. This option is ignored on the .NET platform.
alphanumeric	yes, no	If set to yes, the string is split into a sequence of alphabetic and numeric parts (a numeric part is any consecutive sequence of ASCII digits; anything else is considered alphabetic). Eacn numeric part is considered to be preceded by an alphabetic part even if it is zero-length.

		The parts are then compared pairwise: alphabetic parts using the collation implied by the other query parameters, numeric parts using their numeric value. The result is that, for example, AD985 collates before AD1066.Note that an alphanumeric collation cannot be used in conjunction with functions such as contains() and substring-before().
case-order	upper-first, lower-first	Indicates whether upper case letters collate before or after lower case letters.

Sorting and comparison according to Unicode codepoints can be achieved by setting up a collator as `<saxon:collation name="unicode" class="net.sf.saxon.sort.CodepointCollator"/>`

saxon:continue

A `saxon:continue` instruction causes continuation of a loop defined using saxon:iterate. The instruction must appear lexically within the `saxon:iterate` element.

Typically the instruction will contain a number of `xsl:with-param` elements defining new values for the loop iteration parameters.

For further details and examples see saxon:iterate.

saxon:doctype

The `saxon:doctype` instruction is used to insert a document type declaration into the current output file. It should be instantiated before the first element in the output file is written. It must be used only when writing a final result tree (not a temporary tree) and only when writing text nodes. The reason for these restrictions is that saxon:doctype writes directly to the serialized output stream (internally it uses disable-output-escaping to achieve this). It is not possible to represent a doctype declaration as a node on a temporary tree.

The `saxon:doctype` instruction takes no attributes. The content of the element is a template-body that is instantiated to create an XML document that represents the DTD to be generated; this XML document is then serialized using a special output method that produces DTD syntax rather than XML syntax.

If this element is present the `doctype-system` and `doctype-public` attributes of `xsl:output` should not be present. Also, the `standalone` attribute of `xsl:output` should not be used. This is because the DOCTYPE declaration generated by `saxon:doctype` is output as a text node using disable-output-escaping, and thus appears to the serializer as a document that is not well-formed; the use of `standalone` with such documents is prohibited by the W3C serialization specification.

The generated XML document uses the following elements, where the namespace prefix "dtd" is used for the namespace URI "http://saxon.sf.net/dtd":

Table 12.5.

dtd:doctype	Represents the document type declaration. This is always the top-level element. The element may

	contain dtd:element, dtd:attlist, dtd:entity, and dtd:notation elements. It may have the following attributes: (mandatory) The name of the document type The system ID The public ID
dtd:element	Represents an element type declaration. This is always a child of dtd:doctype. The element is always empty. It may have the following attributes: (mandatory) The name of the element type (mandatory) The content model, exactly as it appears in a DTD, for example content="(#PCDATA)" or content="(a | b | c)*"
dtd:attlist	Represents an attribute list declaration. This is always a child of dtd:doctype. The element will generally have one or more dtd:attribute children. It may have the following attributes: (mandatory) The name of the element type
dtd:attribute	Represents an attribute declaration within an attribute list. This is always a child of dtd:attlist. The element will always be empty. It may have the following attributes: (mandatory) The name of the attribute (mandatory) The type of the attribute, exactly as it appears in a DTD, for example type="ID" or type="(red | green | blue)" (mandatory) The default value of the attribute, exactly as it appears in a DTD, for example value="#REQUIRED" or value="#FIXED 'blue'"
dtd:entity	Represents an entity declaration. This is always a child of dtd:doctype. The element may be empty, or it may have content. The content is a template body, which is instantiated to define the value of an internal parsed entity. Note that this value includes the delimiting quotes. The xsl:entity element may have the following attributes: (mandatory) The name of the entity The system identifier The public identifier Set to "yes" for a parameter entity The name of a notation, for an unparsed entity
dtd:notation	Represents a notation declaration. This is always a child of dtd:doctype. The element will always be empty. It may have the following attributes: (mandatory) The name of the notation The system identifier The public identifier

Note that Saxon will perform only minimal validation on the DTD being generated; it will output the components requested but will not check that this generates well-formed XML, let alone that the output document instance is valid according to this DTD.

Example:

```
<xsl:template match="/">
  <saxon:doctype xsl:extension-element-prefixes="saxon">
  <dtd:doctype name="booklist"
        xmlns:dtd="http://saxon.sf.net/dtd" xsl:exclude-result-prefixes="dtd">
    <dtd:element name="booklist" content="(book)*"/>
    <dtd:element name="book" content="EMPTY"/>
    <dtd:attlist element="book">
      <dtd:attribute name="isbn" type="ID" value="#REQUIRED"/>
      <dtd:attribute name="title" type="CDATA" value="#IMPLIED"/>
    </dtd:attlist>
    <dtd:entity name="blurb">'A <i>cool</i> book with &gt; 200 pictures!'</dtd:
    <dtd:entity name="cover" system="cover.gif" notation="GIF"/>
    <dtd:notation name="GIF" system="http://gif.org/"/>
    <dtd:entity name="ISOEntities"
        public="ISO 8879-1986//ENTITIES ISO Character Entities 20030531//EN//XM
        system="D:\ent\ISOEntities"
        parameter="yes">
    <xsl:text>%ISOEntities;</xsl:text>
  </dtd:doctype>
  </saxon:doctype>
  <xsl:apply-templates/>
</xsl:template>
```

Although not shown in this example, there is nothing to stop the DTD being generated as the output of a transformation, using instructions such as `xsl:value-of` and `xsl:call-template`. It is also possible to use `xsl:text` to output DTD constructs not covered by this syntax, for example conditional sections and references to parameter entities. Such text nodes will always be output with escaping disabled.

saxon:entity-ref

The saxon:entity-ref element is useful to generate entities such as &nbsp; in HTML output. To do this, write:

```
<saxon:entity-ref name="nbsp"/>
```

the preferred way to produce a non-breaking space character in the output is simply to write `&#160;` or `&#xa0;` in the stylesheet. By default, with HTML output, this will be serialized as `&nbsp;`, though the way it is serialized doesn't actually matter as far as the HTML browser is concerned.

`saxon:entity-ref` is permitted only in contexts where `disable-output-escaping` would be permitted: that is, when writing to a serialized output destination.

saxon:finally

A `saxon:finally` instruction may be included as the last child of `saxon:iterate`; it is executed when the end of the input sequence is reached. It is evaluated if the loop is terminated using `saxon:break`. The focus (context node, position, and size) for this instruction is undefined; however, variables declared within the loop including the loop parameters are available for reference. The loop parameters have their values as set by the `saxon:continue` instruction that ended the last normal iteration.

For further details and examples see saxon:iterate.

saxon:import-query

The saxon:import-query element is a top-level declaration that causes an XQuery library module to be imported into the stylesheet.

The effect is that the functions defined in the library module become available for calling from any XPath expression in the stylesheet, as extension functions. They are available in all modules of the stylesheet.

Only the functions actually defined in the given XQuery module are imported. Functions that the specified module imports from other XQuery modules are not imported. This follows the semantics of XQuery's `import module` declaration. Variables defined in an imported module are not (currently) imported into the stylesheet.

The imported functions do not have any specific import precedence. If a stylesheet contains two `saxon:import-query` declarations importing the same namespace, then they are assumed to refer to the same library module, and all but the first are ignored. As with other extension functions, the `override="yes|no"` attribute on `xsl:function` can be used to determine whether a stylesheet function overrides an imported XQuery function of the same name.

The `saxon:import-xquery` declaration has two optional attributes The `href` attribute is the (absolute or relative) URI of the XQuery module. The `namespace` attribute identifies the module namespace of the imported module. If `href` alone is specified, then the module is loaded from the given location. If namespace alone is specified, then the module must already be present in Saxon's `Configuration` object (you can share a Configuration between multiple stylesheets, which means that imported XQuery modules will not need to be recompiled for each one). If both attributes are specified, then Saxon uses an already-loaded module for the namespace if it can, otherwise it fetches it from the specified location, and checks that the namespace is correct. I would recommend specifying both attributes.

saxon:iterate

The `saxon:iterate` element is used to iterate over a sequence. In simple cases the effect is exactly the same as `xsl:for-each`. However, `saxon:iterate` offers a guarantee that the evaluation will be sequential. It allows each iteration to set parameters which will be available during the next iteration, and it allows early exit from the loop when some condition is true.

This extension was new in Saxon 9.1. Some changes have been made in 9.2, and it should still be regarded as experimental. The primary motivation for introducing it is to enable streamed processing of input files where there is a need to "remember" data read at the beginning of the document for use when processing elements encountered later. Streaming is available only in Saxon-EE, but the basic functionality of `saxon:iterate` is also available in Saxon-B

The following example shows how a shopping basket may be displayed with the cumulative values of the items:

```
<saxon:iterate select="//ITEM" xmlns:saxon="http://saxon.sf.net/" xsl:extensi
  <xsl:param name="basketCost" as="xs:decimal" select="0"/>
  <item cost="{$basketCost}"><xsl:copy-of select="TITLE"/></item>
  <saxon:continue>
    <xsl:with-param name="basketCost" select="$basketCost + (xs:decimal(PRICE
  </saxon:continue>
</saxon:iterate>
```

The initial values of the parameters are taken from the `select` attribute of the `xsl:param` elements; values for subsequent iterations are taken from the `xsl:with-param` in the `saxon:continue` instruction. If any declared parameters are not given a value in the `saxon:continue` instruction, they retain their values from the previous iteration.

The following example shows early exit from the loop when the cumulative value reaches 25.00:

```
<saxon:iterate select="//ITEM" xmlns:saxon="http://saxon.sf.net/" xsl:extensi
  <xsl:param name="basketCost" as="xs:decimal" select="0"/>
  <xsl:choose>
```

```
      <xsl:when test="$basketCost gt 25.00">
        <saxon:break/>
      </xsl:when>
      <xsl:otherwise>
        <item cost="{$basketCost}"><xsl:copy-of select="TITLE"/></item>
        <saxon:continue>
          <xsl:with-param name="basketCost" select="$basketCost + (xs:decimal(PI
        </saxon:continue>
      </xsl:otherwise>
    </xsl:choose>
  </saxon:iterate>
```

The instructions `saxon:continue` and `saxon:break` must be lexically within the `saxon:iterate` instruction and must not be followed by further instructions other than `xsl:fallback`. They may appear within `xsl:if` or `xsl:choose` (nested to any depth) so long as none of the intermediate instructions have a following sibling instruction. They must not appear with a nested `xsl:for-each`, a nested literal result element, or nested inside any other instruction.

Within the body of the instruction, the context item, position, and size are available just as in `xsl:for-each`. The value of `last()` reflects the size of the input sequence, which may be greater than the number of iterations if `saxon:break` is used for early exit.

A `saxon:finally` instruction may be included as the last child of `saxon:iterate`; it is executed when the end of the input sequence is reached. It is evaluated if the loop is terminated using `saxon:break`. The focus (context node, position, and size) for this instruction is undefined; however, variables declared within the loop including the loop parameters are available for reference. The loop parameters have their values as set by the `saxon:continue` instruction that ended the last normal iteration.

saxon:mode

The `saxon:mode` element is a top-level element. It is used to define properties of a mode, in particular whether the mode is streamable.

The `name` attribute indicates the name of the mode (which will normally match the `mode` attribute of one or more template rules. Specifying `name="#default"` is equivalent to omitting the attribute, and indicates that the declaration applies to the default (unnamed) mode.

The `streamable` attribute takes the value `yes` or `no`, default `no`. If `yes` is specified, all template rules in this mode must be streamable templates, and all processing using these template rules is done using streaming.

This declaration is available only in Saxon-EE.

saxon:script

The `saxon:script` element is a top-level element. It is used to define an implementation for an extension function that will be used by Saxon. With other processors, a different implementation of the same function can be selected, using mechanisms defined by that processor (for example, `xalan:script`).

The attributes for `saxon:script` are the same as the attributes of the `xsl:script` element defined in the (now withdrawn) XSLT 1.1 working draft.

The `language` attribute is mandatory, and must take the value "java". The values "javascript", "ecmascript", or a QName are also permitted, but in this case Saxon ignores the `saxon:script` element.

The `implements-prefix` attribute is mandatory, its value must be a namespace prefix that maps to the same namespace URI as the prefix used in the extension function call.

The `src` attribute is mandatory for language="java", its value must take the form "java:fully.qualified.class.Name", for example "java:java.util.Date". It defines the class containing the implementation of extension functions that use this prefix.

The `archive` attribute is optional, its value is a space-separated list of URLs of folders or JAR files that will be searched to find the named class. If the attribute is omitted, the class is sought on the classpath.

saxon:try

The `saxon:try` instruction evaluates an XPath expression in its `select` attribute, or a sequence of child instructions. If any dynamic error occurs while performing this evaluation, the error may be caught by an `xsl:catch` instruction that is written as a child of the `xsl:try`. For example, the following code catches a failure occurring while executing the `document()` function, and returns an `<error-document/>` element if this occurs.

```
<xsl:variable name="doc" as="document-node()">
  <saxon:try select="document($input-uri)">
    <saxon:catch errors="*">
      <xsl:document>
        <error-document/>
      </xsl:document>
    </saxon:catch>
  </saxon:try>
</xsl:variable>
```

The `saxon:try` element must contain at least one `saxon:catch` child, and it may contain several. Each one specifies which errors it is intended to catch in its `errors` attribute; the value `*` catches all errors. The first `saxon:catch` to match the error code is the one that is evaluated. The `saxon:catch` children must come after all other children, with the exception of any `xsl:fallback` children (which can be included to define fallback behaviour by processors that do not recognize the `saxon:try` instruction.

The `saxon:try` element may have either a `select` attribute, or a sequence of child instructions (preceding the first `xsl:catch`); it must not have both.

See also saxon:catch.

saxon:while

The `saxon:while` element is used to iterate while some condition is true.

The condition is given as a boolean expression in the mandatory test attribute. Because this expression must change its value if the loop is to terminate, the condition will generally reference a variable that is updated somewhere in the loop using an `saxon:assign` element. Alternatively, it may test a condition that is changed by means of a call on an extension function that has side-effects.

Example:

```
<xsl:variable name="i" select="0" saxon:assignable="yes"/>
<xsl:template name="loop">
  <saxon:while test="$i &lt; 10">
    The value of i is <xsl:value-of select="$i"/>
    <saxon:assign name="i" select="$i+1"/>
  </saxon:while>
</xsl:template>
```

Chapter 13. Sample Saxon Applications

Introduction

Several sample applications are available. They can be downloaded as part of the `saxon-resources` file available from SourceForge [http://sourceforge.net/project/showfiles.php?group_id=29872] or from Saxonica [http://www.saxonica.com/download/download_page_fs.html].

Knight's Tour

This program is available in two forms: as an XSLT stylesheet `tour.xsl` and as an XQuery `tour.xq`.

This is a program whose output is a knight's tour of the chessboard (the knight can start on any square, and has to visit each square exactly once). The XSLT version was published as an example stylesheet in my book (Wrox Press [http://www.wrox.com/]) but has been completely reworked so it now makes extensive use of features in XSLT 2.0, XPath 2.0 and XQuery. It is worth studying the stylesheet and query as an introduction to the use of the new features in these languages. Comparing the two versions, it can be seen that they are very similar: the only differences are in the surface syntax of the two languages.

The stylesheet can be found in the file `samples/styles/tour.xsl`, the query in `samples/query/tour.xq`. No source document is required.

You can run this example with Saxon on the Java platform using a command of the form:

java -jar saxon9.jar -it:main samples\styles\tour.xsl start=e5 >tour.html

or

java -cp saxon9.jar net.sf.saxon.Query samples\query\tour.xq start=e5 >tour.html

On the .NET platform, the equivalent commands are:

Transform -it:main samples\styles\tour.xsl start=e5 >tour.html

or

Query samples\query\tour.xq start=e5 >tour.html

When you display the resulting HTML file in your browser it should look like this:

(Graphic not available)

JAXP Transformation Examples

Saxon supports the Java JAXP Transformation API, originally known as TrAX (package javax.xml.transform) for invoking the XSLT stylesheet processor. This API is useful when you want to write your own Java applications that invoke Saxon XSLT transformations.

A sample program that illustrates many features of the TrAX interface (including Saxon-specific extensions) is included in the distribution as . Source XML and XSLT files for use with this program are included in the directory. To run the program, use the command:

cd $saxonhome/samples/trax java TraxExamples

You can supply an argument to indicate which of the examples you want to run; see the source code for details. By default, they are all executed in turn.

SaxonServlet

This is a general-purpose servlet that takes the name of a source document and XSLT stylesheet as URL parameters, and uses the stylesheet to process the source document, creating a result document which is sent back to the browser for display. The result document may have any media type, though HTML and XML are the most likely.

The servlet maintains a cache of prepared stylesheets; if the same stylesheet is used repeatedly, it is only parsed and validated once, which will often greatly improve performance. Prepared style sheets are thread-safe so they can be used to serve documents to several users concurrently.

The URLs for the source document and the stylesheet document are supplied in the URL, which will typically take the form:

http://server.com/servlets/SaxonServlet?source=doc.xml&style=sheet.xsl

The source and style parameters identify the source document and stylesheet by URL. These are interpreted relative to the servlet context. This means that specifying say "style=/styles/styleone.xsl" in the URL will locate the stylesheet in this file relative to the root directory for the web server.

The stylesheet is prepared the first time it is used, and held in memory in a cache. The cache may be cleared (for example, if a stylesheet has been changed) using a URL such as:

http://server.com/servlets/SaxonServlet?clear-stylesheet-cache=yes

This code is provided purely as a sample, in the expectation that you will customise it to your particular site requirements.

The Book List Stylesheet

This is a very simple sample stylesheet to illustrate several Saxon extensions. It uses the XML file (derived from a file issued originally by Microsoft). You will find this in the samples\data directory. The DTD is in

There is a style sheet that can be used to display the data: run this as follows, with the samples directory as the current directory:

java net.sf.saxon.Transform data\books.xml styles\books.xsl >

Transform data\books.xml styles\books.xsl >

This produces an HTML page showing the data. (The output isn't very pretty, if you have time to write an improved version, please send it in).

The stylesheet takes a parameter named "top-author". This is the name of the "author of the week", and the default value is "Bonner". To run the stylesheet with a different top author, try adding to the end of the command line:

..... top-author=Danzig >

It is possible (under those operating systems I know of) to write the author's name in quotes if it contains spaces, e.g. top-author="A. A. Milne".

There is another style sheet, books-csv.xsl, which converts the data into a comma-separated-values file.

A query that runs with this data is also supplied. The command to use on the Java platform is:

java net.sf.saxon.Query -s:data\books.xml query\books.xq >

The equivalent on .NET is:

Query -s:data\books.xml query\books.xq >

Shakespeare Example

This example works on an input file containing a Shakespeare play. You can use any of the Shakespeare plays in Jon Bosak's distribution at http://www.ibiblio.org/bosak/xml/eg/shaks200.zip, but for convenience one of them, , is included in the Saxon distribution (in the samples\data directory).

Shakespeare stylesheet

There is an XSLT stylesheet, `play.xsl`, which processes an input play in XML and generates a set of linked HTML files (one for the play and one for each scene) in an output directory. To run this on the Java platform, create a directory (say playhtml) and execute the following from the command line:

cd samples java net.sf.saxon.Transform data\othello.xml styles\play.xsl dir=playhtml

The equivalent on .NET is:

cd samples Transform data\othello.xml styles\play.xsl dir=playhtml

The last parameter sets the value of the constant to the value ; this constant is referenced from the style sheet when creating output files.

Shakespeare XPath Sample Application

In the `samples/java` directory is an example application called `XPathExample.java`. This is designed to illustrate the use of Saxon's implementation of the JAXP 1.3 XPath API from a Java application. It searches a Shakespeare play for all occurrences of a word entered from the console, displaying the lines containing that word, and the context where they appear.

To run this example, first ensure that it is on your classpath, and find the location of the `othello.xml` file in the `samples/data` directory. Open a command-line console, and run the application using the command:

cd samples java XPathExample data\othello.xml

The application prompts for a word. Enter the word (try "castle" or "handkerchief"). The lines containing the chosen word are displayed on the console. Exit the application by entering ".".

The Bible

The stylesheet takes as input an XML file containing the text of the Old or New Testament. These files are not included in the Saxon distribution for space reasons, but can be downloaded from http://www.ibiblio.org/bosak/xml/eg/rel200.zip or from various mirror sites. They were prepared by Jon Bosak.

The output of the stylesheet is a set of 292 HTML files in a single directory, which together provide a frames-oriented rendition of the text. The application also works with the Old Testament text, but not with the other religious texts included in Jon Bosak's distribution.

To run the stylesheet on the Java platform first create an output directory (say htmldir), then execute the following from the command line :

java net.sf.saxon.Transform data\nt.xml styles\bible.xsl dir=htmldir

The equivalent on .NET is:

Transform data\nt.xml styles\bible.xsl dir=htmldir

The final parameter sets the value of the XSLT parameter "dir" to the value "htmldir", which is referenced within the stylesheet to select a directory for output files.

JDOM Example

Saxon includes an adapter that allows the source tree to be a JDOM document.

To use this facility:

- The JAR file saxon-jdom.jar must be on the classpath
- JDOM must be installed and on the classpath
- You must be using JDK 1.4 or later.

The sample application JDOMExample.java illustrates how a JDOM tree can be used with Saxon. It combines two scenarios: running XPath expressions directly against a JDOM tree, from the Java application; and invoking a transformation with the JDOM document supplied as the Source object.

The application is designed to take two arguments, the `books.xml` file in the samples/data directory, and the `total.xsl` file in the samples/styles directory. The application builds a JDOM tree, modifies it to add extra attributes, and then references these attributes from the stylesheet.

Example applications for .NET

The `/samples/cs` directory contains somer sample applications written in C#, designed to illustrate use of the Saxon API available in the `Saxon.Api` namespace. In particular there are three programs `ExamplesHE.cs`, `ExamplesPE.cs`, and `ExamplesEE.cs` relevant to Saxon-HE, Saxon-PE, and Saxon-EE respectively; each shows various ways of invoking XSLT, XQuery, XPath, and schema validation where appropriate. The idea is that you should be able to cut and paste a suitable fragment of code into your own application as a quick way of getting started.

The main value of these samples is in reading the source code rather than in actually running them; however, running them is also a good check that your installation has been set up correctly. a quick look at the comments in the source should enable you to see how they are executed. Note that they all require the environment variable `SAXON_HOME` to be set to the directory where Saxon is installed: this is used to find the sample data files.

The example applications can be run from the command line. They take the following arguments:

- - provides the name of the samples directory from the saxon-resources download. The default uses the SAXON_HOME environment variable - but it's normally best to supply this parameter explicitly.
- - with ask:yes (the default), the application prompts after each test to ask if you want to continue.
- - supplies the name of the test you want to run, or "all" (the default) to indicate all tests.

Also in the same directory is an application `TestRunnerProgram.cs` which is a test harness for running the XSLT, XQuery, and XSD test suites published by W3C (in the case of XSLT, the test data is available to W3C members only).

Chapter 14. The Saxon SQL Extension

Introduction

The Saxon distribution includes a set of extension elements providing access to SQL databases. These are not intended as being necessarily a production-quality piece of software (there are many limitations in the design), but more as an illustration of how extension elements can be used to enhance the capability of the processor.

To use the SQL extension elements in a stylesheet, you need to define a namespace prefix (for example "sql") in the extension-element-prefixes attribute of the xsl:stylesheet element, and to map this prefix to a namespace URI, conventionally `http://saxon.sf.net/sql`. This namespace must be bound in the `Configuration [Javadoc: net.sf.saxon.Configuration]` to the class `net.sf.saxon.option.sql.SQLElementFactory`. This binding can be done either by calling `Configuration.setExtensionElementNamespace() [Javadoc: net.sf.saxon.Configuration#setExtensionElementNamespace]`, or by means of an entry in the configuration file.

This extension defines eight new stylesheet elements: `sql:connect`, `sql:query`, `sql:insert`, `sql:column`, `sql:update`, `sql:delete`, `sql:execute`, and `sql:close`, described in the following sections.

sql:connect

`sql:connect` creates a database connection. It has attributes `driver`, `database`, `user`, `password`, and `auto-commit` all of which are attribute value templates (so the values can be passed in as parameters).

The `driver` attribute names the JDBC driver class to be used. The database name must be a name that JDBC can associate with an actual database.

The `auto-commit` attribute, if present, should take the value "yes" or "no". This causes a call of `connection.setAutoCommit()` on the underlying JDBC connection.

The `sql:connect` instruction returns a database connection as a value, specifically a value of type "external object", which can be referred to using the type `java:java.sql.Connection`. Typically the value will be assigned to a variable using the construct:

```
<xsl:variable  name="connection"
               as="java:java.sql.Connection"
               xmlns:java="http://saxon.sf.net/java-type">
  <sql:connect database="jdbc:odbc:testdb"
               driver="sun.jdbc.odbc.JdbcOdbcDriver"
               xsl:extension-element-prefixes="sql"/>
</xsl:variable>
```

This can be a global variable or a local variable; if local, it can be passed to other templates as a parameter in the normal way. The connection is used on instructions such as `sql:insert` and `sql:query` with an attribute such as `connection="$connection"`; the value of the `connection` attribute is an expression that returns a database connection object.

sql:query

`sql:query` performs a query, and writes the results of the query to the result tree, using elements to represent the rows and columns. If you want to process the results in the stylesheet, you can write the results to a temporary tree by using the `sql:query` instruction as a child of `xsl:variable`. The attributes are as follows:

Table 14.1.

connection	The database connection. This is mandatory, the value is an expression, which must evaluate to a database connection object as returned by `sql:connect`.
table	The table to be queried (the contents of the FROM clause of the select statement). This is mandatory, the value is an attribute value template.
column	The columns to be retrieved (the contents of the SELECT clause of the select statement). May be "*" to retrieve all columns. This is mandatory, the value is an attribute value template.
where	The conditions to be applied (the contents of the WHERE clause of the select statement). This is optional, if present the value is an attribute value template.
row-tag	The element name to be used to contain each row. Must be a simple name (no colon allowed). Default is "row".
column-tag	The element name to be used to contain each column. Must be a simple name (no colon allowed). Default is "col".
disable-output-escaping	Allowed values are "yes" or "no", default is "no". The value "yes" causes the content of all rows/ columns to be output as is, without converting special characters such as "<" to "&lt;". This is useful where the database contains XML or HTML markup that you want to be copied into the result document. Use this option with care, however, since it applies to all the columns retrieved, not only to those that contain XML or HTML. An alternative is to use the saxon:parse extension function to process the contents of an XML column.

sql:insert and sql:column

`sql:insert` performs an SQL INSERT statement. This causes a row to be added to the table identified by the `table` attribute. The `table` attribute holds the table name as a constant string; it is not possible to set the name of the table (or of the columns) at run-time.

There is a mandatory `connection` attribute, used as in the `sql:query` instruction described above.

`sql:column` is used as a child element of `sql:insert`, and identifies the name and value of a column to be included in the INSERT statement. The name of the column is identified by the `name` attribute, the value may be indicated either by evaluating the expression contained in the `select` attribute, or as the expanded contents of the `sql:column` element. The value is always interpreted as a string. (Remember this is purely a demonstration of extensibility, in a real system there would be a need to cater for SQL columns of other data types).

sql:update and sql:column

`sql:update` performs an SQL UPDATE statement. This causes a set of rows to be selected from a given table; columns identified in a child `sql:column` element are then updated with new values.

The attributes are as follows:

Table 14.2.

connection	The database connection. This is mandatory, the value is an expression, which must evaluate to a database connection object as returned by `sql:connect`.
table	The table to be updated. This is mandatory, and the value must be known statically (it cannot be defined as an attribute value template).
where	The conditions to be applied (the contents of the WHERE clause of the select statement). This is optional, if present the value is an attribute value template whose value is a SQL conditional expression. If omitted, all rows in the table are updated.

`sql:column` is used as a child element of `sql:update`, and identifies the name and value of a column to be included in the UPDATE statement. The name of the column is identified by the `name` attribute, the value may be indicated either by evaluating the expression contained in the `select` attribute, or as the expanded contents of the `sql:column` element. The value is always interpreted as a String. (Remember this is purely a demonstration of extensibility, in a real system there would be a need to cater for SQL columns of other data types).

sql:delete

`sql:delete` performs an SQL DELETE statement. This causes a set of rows to be selected from a given table and deleted.

The attributes are as follows:

Table 14.3.

connection	The database connection. This is mandatory, the value is an expression, which must evaluate to a database connection object as returned by `sql:connect`.
table	The table containing the rows to be deleted. This is mandatory, and the value must be known statically (it cannot be defined as an attribute value template).
where	The conditions to be applied (the contents of the WHERE clause of the select statement). This is optional, if present the value is an attribute value template whose value is a SQL conditional

	expression. If omitted, all rows in the table are deleted.

sql:close

`sql:close` closes the database connection. There is a mandatory `connection` attribute, used as in the `sql:query` instruction described above.

Note that the JDBC documentation advises calling "commit" or "rollback" before closing the connection; the effects of not doing so are said to be implementation-defined. It is possible to issue a commit or rollback request using the `sql:execute` instruction: see sql:execute.

sql:execute

The `sql:execute` instruction allows arbitrary SQL statements to be executed. There are two attributes (and no content). The `connection` attribute is an XPath expression whose value is the SQL connection created using `sql:connection`. The `statement` attribute is an attribute value template whose effective value is the SQL statement to be executed. No result is returned.

The statements `COMMIT WORK` and `ROLLBACK WORK` (spelt exactly like that) are recognized specially, and cause calls on the JDBC `connection.commit()` and `connection.rollback()` methods respectively.

Example

A specimen stylesheet that uses these XSL extension is . This loads the contents of the books.xml file into a database table, To use it, you need to create a database database containing a table "Book" with three character columns, "Title", "Author", and "Category"

Here is the stylesheet:

```
<xsl:stylesheet
 xmlns:sql="http://saxon.sf.net/sql"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
 xmlns:saxon="http://saxon.sf.net/"
  extension-element-prefixes="saxon sql">

<!-- insert your database details here, or supply them in parameters -->
<xsl:param name="driver" select="'sun.jdbc.odbc.JdbcOdbcDriver'"/>
<xsl:param name="database" select="'jdbc:odbc:test'"/>
<xsl:param name="user"/>
<xsl:param name="password"/>

<!-- This stylesheet writes the book list to a SQL database -->

<xsl:variable name="count" select="0" saxon:assignable="yes"/>

<xsl:output method="xml" indent="yes"/>

<xsl:template match="BOOKLIST">
    <xsl:if test="not(element-available('sql:connect'))">
        <xsl:message>sql:connect is not available</xsl:message>
    </xsl:if>

    <xsl:message>Connecting to <xsl:value-of select="$database"/>...</xsl:messa
    <xsl:variable name="connection"
         as="java:java.sql.Connection" xmlns:java="http://saxon.sf.net/java-typ
      <sql:connect driver="{$driver}" database="{$database}"
                 user="{$user}" password="{$password}">
        <xsl:fallback>
          <xsl:message terminate="yes">SQL extensions are not installed</xsl:me
        </xsl:fallback>
      </sql:connect>
    </xsl:variable>

    <xsl:message>Connected...</xsl:message>

    <xsl:apply-templates select="BOOKS">
      <xsl:with-param name="connection" select="$connection"/>
    </xsl:apply-templates>

    <xsl:message>Inserted <xsl:value-of select="$count"/> records.</xsl:message

    <xsl:variable name="book-table">
      <sql:query connection="$connection" table="Book" column="*" row-tag="book
    </xsl:variable>

    <xsl:message>There are now <xsl:value-of select="count($book-table//Book)"/
    <new-book-table>
        <xsl:copy-of select="$book-table"/>
    </new-book-table>

    <sql:close connection="$connection"/>

</xsl:template>

<xsl:template match="BOOKS">
    <xsl:param name="connection"/>
    <xsl:for-each select="ITEM">
     <sql:insert connection="$connection" table="Book">
         <sql:column name="title" select="TITLE"/>
            <sql:column name="author" select="AUTHOR"/>
            <sql:column name="category" select="@CAT"/>
     </sql:insert>
     <saxon:assign name="count" select="$count+1"/>
    </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Running the example using Microsoft Access

To run this stylesheet you will need to do the following:

1. Create a database (e.g. Microsoft Access) containing a table "Book" with three character columns, "Title", "Author", and "Category".

2. Register this database as a JDBC data source. (If you use Microsoft Access, register it as an ODBC data source called, say, Books, and then it will automatically be available under JDBC as "jdbc:odbc:Books".

3. Modify the `<sql:connect>` element in the stylesheet to specify the correct JDBC connection name for the database, and if necessary to supply a username and password. Alternatively you can

supply the driver class, database name, username, and password as parameters on the command line.

4. Execute the stylesheet from the command line, as follows:

   java net.sf.saxon.Transform data\books.xml style\books-sql.xsl

The database will be populated with data from the `books.xml` document.

Running the example using MySQL

The following instructions illustrates how to run the stylesheet using MySQL, under a UNIX platform:

1. Create the database, using MySQL.

2. Register the database as a JDBC data source (same as 2. above). However, change the `xsl:param` elements with attribute names "driver" and "database" as follows (we assume machine is localhost):

```
<xsl:stylesheet
               <xsl:param name="driver" select="'com.mysql.jdbc.Driver'"/>
               <xsl:param name="database" select="'jdbc:mysql://localhost:330
```

3. Download the MySQL connector jar file, if missing.

4. Execute the stylesheet from the command line (same as 4. above). The kind of problems that might occur are as follows:

   - "JDBC Connection Failure: com.mysql.jdbc.Driver": Ensure the MySQL connector jar file is downloaded and in the classpath

   - "SQL extensions are not installed": The binding of the namespace for the SQL extension must be present in the `Configuration` object and must match the URI in the stylesheet. For execution of the stylesheet from the command line, the following is required:

     java net.sf.saxon.Transform -config:data/config.xml data/books.xml style/books-mysql.xsl

     where the configuration file `data/config.xml` includes the entry

```
<configuration xmlns="http://saxon.sf.net/ns/configuration"
               edition="EE">
  <xslt>
    <extensionElement namespace="http://saxon.sf.net/sql"
                      factory="net.sf.saxon.option.sql.SQLElementFactory"/>
  </xslt>
</configuration>
```

A Warning about Side-Effects

XSLT does not guarantee the order of execution of instructions. In principle, the `sql:close` instruction could be evaluated before the `sql:query` instruction, which would obviously be disastrous.

In practice, Saxon's order of execution for XSLT instructions is reasonably predictable unless you use variables, or unless you invoke stylesheet functions from within an XPath expression. Using the SQL extension instructions within templates that are directly invoked is thus fairly safe, but it is not a good idea to invoke them as a side-effect of computing a variable or invoking a function. The exceptions are `sql:connect` and `sql:query`: if you use these instructions to evaluate the content of a variable, then of course they will be executed before any instruction that uses the value of the variable.

A Warning about Security (SQL injection)

The instructions in the SQL extension make no attempt to verify that the SQL being executed is correct and benign. No checks are made against injection attacks; indeed the `sql:execute` instruction explicitly allows any SQL statement to be executed.

Therefore, the extension should be enabled only if (a) the stylesheet itself is trusted, and (b) any text inserted into the stylesheet to construct dynamic SQL statements is also trusted.

Chapter 15. XSLT Elements

Introduction

This section of the Saxon documentation lists the standard XSLT elements, all of which are supported in Saxon stylesheets, and gives brief descriptions of their function. In some cases the text gives information specific to the Saxon implementation. For extension elements provided with the Saxon product, see Extensions.

Saxon implements the XSLT version 2.0 Recommendation from the World Wide Web Consortium: see Conformance. The information here is designed to give a summary of the features: for the full specification, consult the official standard.

This section of the documentation also describes new elements in the draft XSLT 3.0 specifications, with notes on the extent to which they are implemented in Saxon.

xsl:analyze-string

The `xsl:analyze-string` element is new in XSLT 2.0. It applies a regular expression to a supplied string value. The string is split into a sequence of substrings, each of which is classified as either a matching substring (if it matches the regular expression) or a non-matching substring (if it doesn't). The substrings are then processed individually: the matching substrings by a `xsl:matching-substring` element that appears as a child of the `xsl:analyze-string` instruction, the non-matching substrings by a similar `xsl:non-matching-substring` element. If either of these is omitted, the relevant substrings are not processed.

The element has three attributes: `select` is an XPath expression whose value is the string to be analyzed; `regex` is the regular expression (which may be given as an attribute value template), and `flags` provides one or more Perl-like flags to control the way in which regular expression matching is performed, for example the value "m" indicates multi-line mode.

When processing matching substrings, it is possible to call the `regex-group()` function to find the parts of the matching substring that matched particular parenthesized groups within the regular expression.

There are examples [http://www.w3.org/TR/xslt20/#regex-examples] of this element in the XSLT 2.0 Working Draft.

xsl:apply-imports

The `xsl:apply-imports` element is used in conjunction with imported stylesheets. There are no attributes. The element may contain zero or more `xsl:with-param` elements (as permitted in XSLT 2.0).

At run-time, there must be a . A current template is established when a template is activated as a result of a call on `xsl:apply-templates`. Calling `xsl:call-template` does not change the current template. Calling xsl:for-each or xsl:for-each-group causes the current template to become null.

The effect is to search for a template that matches the current node and that is defined in a stylesheet that was imported (directly or indirectly, possibly via `xsl:include`) from the stylesheet containing the current template, and whose mode matches the current mode. If there is such a template, it is activated using the current node. If not, the call on `xsl:apply-imports` has no effect.

To supply parameters to the called template, one or more xsl:with-param elements may be included. The values of these parameters are available to the called template. If the `xsl:with-param` element specifies `tunnel="yes"`, then the parameter is passed transparently through to templates called at any depth, but it can only be referenced by an `xsl:param` element that also specifies `tunnel="yes"`. If the default value, `tunnel="no"` is used, then the parameter value is available only in the immediately called template, and only if the `xsl:param` element specifies `tunnel="no"` (explicitly or by defaulting the attribute).

xsl:apply-templates

The `xsl:apply-templates` element causes navigation from the current element, usually but not necessarily to process its children. Each selected node is processed using the best-match `xsl:template` defined for that node.

The `xsl:apply-templates` element takes an optional attribute, `mode`, which identifies the processing mode. If this attribute is present, only templates with a matching `mode` parameter will be considered when searching for the rule to apply to the selected elements.

It also takes an optional attribute, `select`.

If the `select` attribute is , apply-templates causes all the immediate children of the current node to be processed: that is, child elements and character content, in the order in which it appears. Character content must be processed by a template whose match pattern will be something like `*/text()`. Child elements similarly are processed using the appropriate template, selected according to the rules given below under xsl:template.

If the `select` attribute is , the result must be a sequence of nodes. All nodes selected by the expression are processed.

The `xsl:apply-templates` element is usually empty, in which case the selected nodes are processed in the order they are selected (this will usually be document order, but this depends on the `select` expression that is used). However the element may include `xsl:sort` and/or `xsl:param` elements:

- For sorted processing, one or more child xsl:sort elements may be included. These define the sort order to be applied to the selection. The sort keys are listed in major-to-minor order.
- To supply parameters to the called template, one or more xsl:with-param elements may be included. The values of these parameters are available to the called template. If the `xsl:with-param` element specifies `tunnel="yes"`, then the parameter is passed transparently through to templates called at any depth, but it can only be referenced by an `xsl:param` element that also specifies `tunnel="yes"`. If the default value, `tunnel="no"` is used, then the parameter value is available only in the immediately called template, and only if the `xsl:param` element specifies `tunnel="no"` (explicitly or by defaulting the attribute).

The selected nodes are processed in a particular . This context includes:

- A current node: the node being processed
- A current node list: the list of nodes being processed, in the order they are processed (this affects the value of the position() and last() functions)
- A set of variables, which initially is those variable defined as parameters

Some examples of the most useful forms of select expression are listed below:

Table 15.1.

XXX	Process all immediate child elements with tag XXX
*	Process all immediate child elements (but not character data within the element)
../TITLE	Process the TITLE children of the parent element

XXX[@AAA]	Process all XXX child elements having an attribute named AAA
@*	Process all attributes of the current element
*/ZZZ	Process all grandchild ZZZ elements
XXX[ZZZ]	Process all child XXX elements that have a child ZZZ
XXX[@WIDTH and not(@width="20")]	Process all child XXX elements that have a WIDTH attribute whose value is not "20"
AUTHOR[1]	Process the first child AUTHOR element
APPENDIX[@NUMBER][last()]	Process the last child APPENDIX element having a NUMBER attribute
APPENDIX[last()][@NUMBER]	Process the last child APPENDIX element provided it has a NUMBER attribute

The full syntax of select expressions is outlined in XPath Expression Syntax.

In XSLT 3.0, the `xsl:apply-templates` instruction can select atomic values as well as nodes, and the match pattern syntax of `xsl:template` is extended to allow atomic values as well as nodes to be matched. As of Saxon 9.4, not all the extensions to match pattern syntax are implemented, but some are, including in particular the construct `match="~itemtype"` which matches any item of a specified type, for example `match="~xs:integer[. mod 2 = 0]"` matches any even integer.

xsl:attribute

The `xsl:attribute` element is used to add an attribute value to an `xsl:element` element or general formatting element, or to an element created using `xsl:copy`. The attribute must be output immediately after the element, with no intervening character data. The name of the attribute is indicated by the `name` attribute and the value by the content of the `xsl:attribute` element.

The attribute name is interpreted as an , so it may contain string expressions within curly braces. The full syntax of string expressions is outlined in XPath Expression Syntax.

The attribute value may be given either by a `select` attribute or by an enclosed sequence constructor. If the `select` attribute is used and the value is a sequence, then the items in the sequence are output space-separated.

The `separator` attribute can be used to specify an alternative separator.

For example, the following code creates a <FONT> element with several attributes:

```
<xsl:element name="FONT">
    <xsl:attribute name="SIZE">4</xsl:attribute>
    <xsl:attribute name="FACE">Courier New</xsl:attribute>
Some output text
</xsl:element>
```

A new attribute `type` was added in XSLT 2.0. This indicates the data type of the value of the attribute. The value must either be a built-in type defined in XML Schema, for example `xs:integer` or `xs:date`, or a user-defined type defined in a schema imported using `xsl:import-schema`. Type annotations are only accessible if the attribute is added to a temporary tree that specifies `validation="preserve"`. The value given to the attribute must be a string that conforms to the rules for the data type, as defined in XML Schema.

There are two main uses for the `xsl:attribute` element:

- It is the only way to set attributes on an element generated dynamically using xsl:element
- It allows attributes of a literal result element to be calculated using xsl:value-of.

The `xsl:attribute` must be output immediately after the relevant element is generated: there must be no intervening character data (other than white space which is ignored). Saxon outputs the closing ">" of the element start tag as soon as something other than an attribute is written to the output stream, and rejects an attempt to output an attribute if there is no currently-open start tag. Any special characters within the attribute value will automatically be escaped (for example, "<" will be output as "&lt;")

If two attributes are output with the same name, the second one takes precedence.

xsl:attribute-set

The `xsl:attribute-set` element is used to declare a named collection of attributes, which will often be used together to define an output style. It is declared at the top level (subordinate to `xsl:stylesheet`).

An attribute-set contains a collection of `xsl:attribute` elements.

The attributes in an attribute-set can be used in several ways:

- They can be added to a literal result element by specifying `xsl:use-attribute-sets` in the list of attributes for the element. The value is a space-separated list of attribute-set names. Attributes specified explicitly on the literal result element, or added using `xsl:attribute`, override any that are specified in the attribute-set definition.
- They can be added to an element created using `xsl:element`, by specifying use-attribute-sets in the list of attributes for the xsl:element element. The value is a space-separated list of attribute-set names. Attributes specified explicitly on the literal result element, or added using `xsl:attribute`, override any that are specified in the attribute-set definition.
- One attribute set can be based on another by specifying `use-attribute-sets` in the list of attributes for the `xsl:attribute-set` element. Again, attributes defined explicitly in the attribute set override any that are included implicitly from another attribute set.

Attribute sets named in the `xsl:use-attribute-sets` or `use-attribute-sets` attribute are applied in the order given: if the same attribute is generated more than once, the later value always takes precedence.

xsl:break

The `xsl:break` instruction is new in XSLT 3.0; it occurs within `xsl:iterate`. For details see xsl:iterate.

xsl:call-template

The `xsl:call-template` element is used to invoke a named template.

The `name` attribute is mandatory and must match the name defined on an `xsl:template` element.

Saxon supports an alternative instruction `saxon:call-template`. This has the same effect as `xsl:call-template`, except that the `name` attribute may be written as an attribute value template, allowing the called template to be decided at run-time. The string result of evaluating the attribute value template must be a valid QName that identifies a named template somewhere in the stylesheet.

To supply parameters to the called template, one or more xsl:with-param elements may be included. The values of these parameters are available to the called template. If the `xsl:with-param` element specifies `tunnel="yes"`, then the parameter is passed transparently through to templates called at any depth, but it can only be referenced by an `xsl:param` element that also specifies `tunnel="yes"`. If the default value, `tunnel="no"` is used, then the parameter value is available only in the immediately called template, and only if the `xsl:param` element specifies `tunnel="no"` (explicitly or by defaulting the attribute).

The context of the called template (for example the current node and current node list) is the same as that for the calling template; however the variables defined in the calling template are not accessible in the called template.

xsl:character-map

The `xsl:character-map` declaration defines a named character map for use during serialization. The `name` attribute gives the name of the character map, which can be referenced from the `use-character-maps` attribute of `xsl:output`. The `xsl:character-map` element contains a set of `xsl:output-character` elements each of which defines the output representation of a given Unicode character. The character is specified using the `character` attribute, the string which is to replace this character on serialization is specified using the `string` attribute. Both attributes are mandatory.

The replacement string is output , even if it contains special (markup) characters. So, for example, you can define <xsl:output-character character="&#xa0;" string="&nbsp;"/> to ensure that NBSP characters are output using the entity reference `&nbsp;`.

Character maps allow you to produce output that is not well-formed XML, and they thus provide a replacement facility for `disable-output-escaping`. A useful technique is to use characters in the Unicode private use area (xE000 to xF8FF) as characters which, if present in the result tree, will be mapped to special strings on output. For example, if you want to generate a proprietary XML-like format that uses tags such as <!IF>, <!THEN>, and <!ELSE>, then you could map these to the three characters xE000, xE001, xE002 (which you could in turn define as entities so they can be written symbolically in your stylesheet or source document).

Character maps are preferred to `disable-output-escaping` because they do not rely on an intimate interface between the transformation engine and the serializer, and they do not distort the data model. The special characters can happily be stored in a DOM, passed across the SAX interface, or manipulated in any other way, before finally being rendered by the serializer.

Character maps may be assembled from other character maps using the `use-character-maps` attribute. This contains a space-separated list of the names of other character maps that are to be included in this character map.

Using character maps may be expensive at run-time. Saxon currently makes no special attempts to optimize their use: if character maps are used, then every character that is output will be looked up in a hash table to see if there is a replacement string.

xsl:choose

The `xsl:choose` element is used to choose one of a number of alternative outputs. The element typically contains a number of xsl:when elements, each with a separate test condition. The first `xsl:when` element whose condition matches the current element in the source document is expanded, the others are ignored. If none of the conditions is satisfied, the xsl:otherwise child element, if any, is expanded.

The test condition in the `xsl:when` element is a boolean expression. The full syntax of expressions is outlined in XPath Expression Syntax.

Example:

```
<xsl:choose>
    <xsl:when test="@cat='F'">Fiction</xsl:when>
    <xsl:when test="@cat='C'">Crime</xsl:when>
    <xsl:when test="@cat='R'">Reference</xsl:when>
    <xsl:otherwise>General</xsl:otherwise>
</xsl:choose>
```

xsl:comment

The `xsl:comment` element can appear anywhere within an `xsl:template`. It indicates text that is to be output to the current output stream in the form of an XML or HTML comment.

The content of the comment may be given either by a `select` attribute or by an enclosed sequence constructor. If the `select` attribute is used and the value is a sequence, then the items in the sequence are output space-separated.

For example, the text below inserts some JavaScript into a generated HTML document:

```
<script language="JavaScript">
    <xsl:comment>
        function bk(n) {
            parent.frames['content'].location="chap" + n + ".1.html";
        }
    //</xsl:comment>
</script>
```

Note that special characters occurring within the comment text will be escaped.

The `xsl:comment` element will normally contain text only but it may contain other elements such as xsl:if or xsl:value-of. However, it should not contain literal result elements.

> Tip: the `xsl:comment` element can be very useful for debugging your stylesheet. Use comments in the generated output as a way of tracking which rules in the stylesheet were invoked to produce the output.

xsl:copy

The `xsl:copy` element causes the current XML node in the source document to be copied to the output. The actual effect depends on whether the node is an element, an attribute, or a text node.

For an element, the start and end element tags are copied; the attributes, character content and child elements are copied only if `xsl:apply-templates` is used within `xsl:copy`.

Attributes of the generated element can be defined by reference to a named attribute set. The optional use-attribute-sets attribute contains a white-space-separated list of attribute set names. They are applied in the order given: if the same attribute is generated more than once, the later value always takes precedence.

The following example is a template that copies the input element to the output, together with all its child elements, character content, and attributes:

```
<xsl:template match="*|text()|@*">
    <xsl:copy>
        <xsl:apply-templates select="@*"/>
        <xsl:apply-templates/>
    </xsl:copy>
</xsl:template>
```

In XSLT 3.0, a new `select` attribute is added to `xsl:copy`, allowing a node other than the context node to be copied. This is useful when the instruction appears inside `xsl:function`.

xsl:copy-of

The `xsl:copy-of` element copies the value obtained by evaluating the mandatory `select` attribute. It makes an exact copy.

If this expression is a string, a number, or a boolean, the effect is the same as using `xsl:sequence`.

There is an optional attribute `copy-namespaces` whose value is "yes" or "no". The default is "yes". This controls whether the in-scope namespaces of any element nodes copied by this instruction are automatically copied to the result. If the value is "no", then namespaces will only be copied if they are actually used in the names of elements or attributes. This allows you, for example, to copy the contents of a SOAP message without copying the namespaces declared in its envelope.

xsl:decimal-format

The `xsl:decimal-format` element is used at the top level of a stylesheet to indicate a set of localisation parameters. If the `xsl:decimal-format` element has a `name` attribute, it identifies a named format; if not, it identifies the default format.

In practice decimal formats are used only for formatting numbers using the `format-number()` function in XPath expressions. For details of the attributes available, see the XSLT specification.

xsl:document

The `xsl:document` instruction creates a new document node. The content of the new document node is created using the contained instructions (in the same way as `xsl:result-document`), and the new document node is added to the result sequence. The instruction is useful mainly if you want to validate the document: the element allows attributes `validation` and `type` which perform document-level validation in the same way as the corresponding attributes on `xsl:result-document`.

The instruction also allows a function or template to create a temporary tree without the need to create a variable and then return the value of the variable.

This instruction should not be confused with the instruction of the same name in the withdrawn XSLT 1.1 draft, (which is supported in Saxon 6.5.x). That instruction was a precursor to `xsl:result-document`.

xsl:element

The `xsl:element` instruction is used to create an output element whose name might be calculated at run-time.

The element has a mandatory attribute, `name`, which is the name of the generated element. The name attribute is an , so it may contain string expressions inside curly braces.

The attributes of the generated element are defined by subsequent `xsl:attribute` elements. The content of the generated element is whatever is generated between the `<xsl:element>` and `</xsl:element>` tags.

Additionally, attributes of the generated element can be defined by reference to a named attribute set. The optional use-attribute-sets attribute contains a white-space-separated list of attribute set names. They are applied in the order given: if the same attribute is generated more than once, the later value always takes precedence.

For example, the following code creates a <FONT> element with several attributes:

```
<xsl:element name="FONT">
    <xsl:attribute name="SIZE">4</xsl:attribute>
    <xsl:attribute name="FACE">Courier New</xsl:attribute>
Some output text
</xsl:element>
```

A new attribute `type` was added in XSLT 2.0. This indicates the data type of the value of the element. The value may be a built-in type defined in XML Schema, for example `xs:integer` or `xs:date`, or a user-defined type defined in a schema imported using `xsl:import-schema`. Type annotations are only accessible if the attribute is added to a temporary tree that specifies `type-information="preserve"`. The attribute causes the content of the element to be validated against the schema definition of the type, and will cause a fatal dynamic error if validation fails.

xsl:evaluate

The `xsl:evaluate` instruction is new in XSLT 3.0. It allows dynamic evaluation of XPath expressions constructed as a string, in the same way as the `saxon:evaluate()` extension function that has been available in Saxon for many years.

The following example sorts product elements according to a sort key supplied (in the form of an XPath expression) as a parameter to the stylesheet.

```
<xsl:apply-templates select="product">
  <xsl:sort>
    <xsl:evaluate select="$product-sort-key"/>
  </xsl:sort>
</xsl:apply-templates>
```

The functionality is available as an XSLT instruction, rather than a function, to allow more flexibility in the syntax, in particular the ability to define parameters using `xsl:with-param` child elements.

The instruction is fully implemented in Saxon 9.3 with the following exceptions:

- Functions available only in XSLT, such as key(), cannot be used in the target XPath expression.

The instruction may take an `xsl:fallback` to define fallback behaviour when using an XSLT 2.0 processor.

Attributes:

- xpath: an expression, which is evaluated to return the target expression as a string.
- base-uri: a string (as an AVT), gives the base URI for the target expression. Defaults to the base URI of the stylesheet instruction.
- namespace-context: an expression returning a node; the in-scope namespaces of this node define the namespace context for the XPath expression. Defaults to the namespace context of the `xsl:evaluate` instruction in the stylesheet
- as: a SequenceType: defines the required type of the result of the XPath expression. Defaults to `item()*`
- schema-aware: "yes" or "no", as an AVT: if "yes", the XPath expression has access to the schema components imported into the stylesheet.

Children:

- `xsl:with-param`: defines variables that the target expression can use.
- `xsl:fallback`: defines fallback behaviour when using an XSLT 2.0 (or 1.0) processor.

Before using `xsl:evaluate`, consider whether higher-order functions (also new in XSLT 3.0) would provide a better solution to the problem. Also think carefully about the possibility of injection attacks if the expression to be evaluated is formed by string concatenation.

xsl:fallback

The `xsl:fallback` element is used to define recovery action to be taken when an instruction element is used in the stylesheet and no implementation of that element is available. An element is an instruction element if its namespace URI is the standard URI for XSLT elements or if its namespace is identified in the `xsl:extension-element-prefixes` attribute of a containing literal result element, or in the `extension-element-prefixes` attribute of the `xsl:stylesheet` element.

If the `xsl:fallback` element appears in any other context, it is ignored, together with all its child and descendant elements.

There are no attributes.

If the parent element can be instantiated and processed, the `xsl:fallback` element and its descendants are ignored. If the parent element is not recognised of if any failure occurs instantiating it, all its xsl:fallback children are processed in turn. If there are no xsl:fallback children, an error is reported.

xsl:for-each

The `xsl:for-each` element causes iteration over the nodes selected by a node-set expression. It can be used as an alternative to `xsl:apply-templates` where the child nodes of the current node are known in advance. There is a mandatory attribute, `select`, which defines the nodes over which the statement will iterate. The XSLT statements subordinate to the `xsl:for-each` element are applied to each source node selected by the node-set expression in turn.

The full syntax of expressions is outlined in XPath Expression Syntax.

The `xsl:for-each` element may have one or more `xsl:sort` child elements to define the order of sorting. The sort keys are specified in major-to-minor order.

The expression used for sorting can be any string expressions. The following are particularly useful:

- element-name, e.g. TITLE: sorts on the value of a child element
- attribute-name, e.g. @CODE: sorts on the value of an attribute
- ".": sorts on the character content of the element
- "qname(.)": sorts on the name of the element

Example 1:

```
<xsl:template match="BOOKLIST">
    <TABLE>
    <xsl:for-each select="BOOK">
        <TR>
        <TD><xsl:value-of select="TITLE"/></TD>
        <TD><xsl:value-of select="AUTHOR"/></TD>
        <TD><xsl:value-of select="ISBN"/></TD>
        </TR>
    </xsl:for-each>
    </TABLE>
</xsl:template>
```

Example 2: sorting with xsl:for-each. This example also shows a template for a BOOKLIST element which processes all the child BOOK elements in order of their child AUTHOR elements.

```
<xsl:template match="BOOKLIST">
```

```
<h2>
    <xsl:for-each select="BOOK">
        <xsl:sort select="AUTHOR"/>
        <p>AUTHOR: <xsl:value-of select="AUTHOR"/></p>
        <p>TITLE: <xsl:value-of select="TITLE"/></p>
        <hr/>
    </xsl:for-each>
</h2>
</xsl:template>
```

Saxon-EE offers an extension to the `xsl:for-each` instruction: the `saxon:threads` attribute allows the items in the input sequence to be processed in parallel. For details see saxon:threads.

xsl:for-each-group

The `xsl:for-each-group` element selects a sequence of nodes and/or atomic values and organizes them into subsets called groups. There are four possible ways of defining the grouping:

- This groups together all items having the same value for a grouping key. The grouping key may have multiple values (a sequence of values) in which case the item is added to more than one group.
- This groups together all items having the same value for a grouping key, provided that they are also adjacent in the input sequence. This is useful when you need to wrap a new element around a sequence of related elements in the source documents, for example a consecutive sequence of `<bullet>` elements. In this case the grouping key must be single-valued.
- This processes the items in the supplied sequence in turn, starting a new group whenever one of the items matches a specified pattern. This is useful, for example, when matching an `<h2>` element and its following `<p>` elements.
- This processes the items in the supplied sequence in turn, closing the current group whenever one of the items matches a specified pattern. This is useful when matching a sequence of items in which the last item in the group carries some distinguishing attribute such as `continued="no"`.

Saxon implements the `xsl:for-each-group` instruction in full. For examples of using the instruction, see the XSLT 2.0 specification [http://www.w3.org/TR/xslt20/#grouping].

In XSLT 3.0, the capabilities of the `xsl:for-each-group` instruction are extended by virtue of the fact that the pattern used in `group-starting-with` or `group-ending-with` can now match atomic values as well as nodes.

xsl:function

The `xsl:function` element defines a function within a stylesheet. The function is written in XSLT but it may be called from any XPath expression in the stylesheet. It must have a non-default namespace prefix.

Example:

```
<xsl:function name="my:factorial" as="xs:integer">
<xsl:param name="number" as="xs:integer"/>
<xsl:sequence
          select="if ($number=0) then 1 else $number * my:factorial($number-1
```

In limited circumstances, stylesheet functions (`xsl:function`) optimise tail-recursion. The circumstances are that the `select` expression of the `xsl:result` instruction must contain a call on the same function as the `then` or `else` part of a conditional expression (which may be nested in further conditional expressions). It may require a little care to write functions to exploit this. The example above is not tail-recursive, because the recursive call is within an arithmetic expression: the multiplication takes place on return from the recursive call. It can be recast in tail-recursive form by adding an extra parameter (which should be set to 1 on the initial call):

```
<xsl:function name="x:factorial">
    <xsl:param name="acc" as="xs:integer?"/>
    <xsl:param name="n" as="xs:integer"/>
    <xsl:sequence as="xs:integer"
         select="if ($n = 1)
                 then $acc
                 else x:factorial($acc*$n, $n - 1)" />
</xsl:function>
```

The call `x:factorial(1, 5)` returns 120.

Saxon defines an extra attribute on `xsl:function`: `saxon:memo-function="yes"` indicates that Saxon should remember the results of calling the function in a cache, and if the function is called again with the same arguments, the result is retrieved from the cache rather than being recalculated. Further details: see saxon:memo-function.

xsl:if

The `xsl:if` element is used for conditional processing. It takes a mandatory `test` attribute, whose value is a boolean expression. The contents of the xsl:if element are expanded only of the expression is true.

The full syntax of boolean expressions is outlined in XPath Expression Syntax.

Example:

```
<xsl:if test="@preface">
        <a href="preface.html">Preface</a>
</xsl:if>
```

This includes a hyperlink in the output only if the current element has a `preface` attribute.

xsl:include

The `xsl:include` element is always used at the top level of the stylesheet. It has a mandatory `href` attribute, which is a URL (absolute or relative) of another stylesheet to be textually included within this one. The top-level elements of the included stylesheet effectively replace the xsl:include element.

`xsl:include` may also be used at the top level of the included stylesheet, and so on recursively.

To customize the way in which the `href` attribute is handled, a user-written `URIResolver` can be supplied. See also Using XML Catalogs.

xsl:import

The `xsl:import` element is always used at the top level of the stylesheet, and it must appear before all other elements at the top level. It has a mandatory `href` attribute, which is a URL (absolute or relative) of another stylesheet to be textually included within this one. The top-level elements of the included stylesheet effectively replace the xsl:import element.

The `xsl:import` element may also be used at the top level of the included stylesheet, and so on recursively.

The elements in the imported stylesheet have lower precedence than the elements in the importing stylesheet. The main effect of this is on selection of a template when xsl:apply-templates is used: if there is a matching template with precedence X, all templates with precedence less than X are ignored, regardless of their priority.

To customize the way in which the `href` attribute is handled, a user-written `URIResolver` can be supplied. See also Using XML Catalogs.

xsl:import-schema

Saxon implements the `xsl:import-schema` declaration in the enterprise edition product Saxon-EE only.

The `namespace` attribute specifies the target namespace of the schema to be imported. The attribute should be omitted when importing a schema with no target namespace.

The `schema-location` attribute specifies where the schema document can be found. This URI is passed through the `URIResolver` in the same way as the URIs used on `xsl:include` and `xsl:import`. The attribute can be omitted only if a schema for the required namespace has already been loaded in the `Configuration [Javadoc: net.sf.saxon.Configuration]`, for example if it has already been imported from another stylesheet module.

For further information see Schema Processing.

xsl:iterate

The `xsl:iterate` instruction is new in XSLT 3.0. It is similar to `xsl:for-each`, except that the items in the input sequence are processed sequentially, and after processing each item in the input sequence it is possible to set parameters for use in the next iteration. It can therefore be used to solve problems than in XSLT 2.0 require recursive functions or templates.

Here is an example that computes the running balance of a sequence of financial transactions:

```
<xsl:iterate select="transactions/transaction">
  <xsl:param name="balance" select="0.00" as="xs:decimal"/>
  <xsl:variable name="newBalance"
                select="$balance + xs:decimal(@value)"/>
  <balance date="{@date}" value="{$newBalance}"/>
  <xsl:next-iteration>
    <xsl:with-param name="balance" select="$newBalance"/>
  </xsl:next-iteration>
  </xsl:iterate>
```

As well as `xsl:next-iteration`, the instruction allows a child element `xsl:break` which causes premature completion before the entire input sequence has been processed, and a child element `xsl:on-completion` which defines processing to be carried out when the input sequence is exhausted. The instructions within `xsl:on-completion` have access to the final values of the parameters declared in the `xsl:next-iteration` instruction set while processing the last item in the sequence.

Here is an example that copies the input sequence up to the first `br` element:

```
<xsl:iterate select="*">
  <xsl:choose>
    <xsl:when test="self::br">
      <xsl:break/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:copy-of select="."/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:iterate>
```

xsl:key

The `xsl:key` element is used at the top level of the stylesheet to declare an attribute, or other value, that may be used as a key to identify nodes using the `key()` function within an expression. Each xsl:key definition declares a named key, which must match the name of the key used in the `key()` function.

The set of nodes to which the key applies is defined by a pattern in the match attribute: for example, if `match="ACT|SCENE"` then every `ACT` element and every `SCENE` element is indexed by this key.

The value of the key, for each of these matched elements, is determined by the `use` attribute. This is an expression, which is evaluated for each matched element. If the expression returns a node-set, the typed value of each node in this node-set acts as a key value. For example, if `use="AUTHOR"`, then each AUTHOR child of the matched element supplies one key value.

Note that:

1. Keys are not unique: the same value may identify many different nodes
2. Keys are multi-valued: each matched node may have several (zero or more) values of the key, any one of which may be used to locate that node
3. Keys can only be used to identify nodes within a single XML document: the key() function will return nodes that are in the same document as the current node.

All three attributes, `name`, `match`, and `use`, are mandatory.

The optional `collation` attribute can be used when comparing strings.

Saxon does not yet allow the `xsl:key` element to contain a sequence constructor in place of the `use` attribute.

xsl:matching-substring

The `xsl:matching-substring` element is used within an `xsl:analyze-string` element to indicate the default action to be taken with substrings that match a regular expression.

See xsl:analyze-string.

xsl:merge

The `xsl:merge` instruction is new in XSLT 3.0, and is first implemented in Saxon-EE 9.4. The purpose of the instruction is to allow streamed merging of two or more pre-sorted input files, but the current implementation in Saxon is unstreamed.

Each kind of input source is described in an `xsl:merge-source` child element of the `xsl:merge` instruction; and the instances of that kind of input source are selected in a `xsl:merge-input` child of the `xsl:merge-source` element. The processing to be carried out on each group of input items sharing a value for the merge key is defined in a `xsl:merge-action` element.

The following example merges a homogenous collection of log files, each already sorted by timestamp:

```
<xsl:merge>
  <xsl:merge-source select="collection('log-collection')">
    <xsl:merge-input select="events/event"/>
    <xsl:merge-key select="@timestamp" order="ascending"/>
  </xsl:merge-source>
  <xsl:merge-action>
    <xsl:sequence select="current-group()"/>
  </xsl:merge-action>
</xsl:merge>
```

The following example merges two log files with different internal structure:

```
<xsl:merge>
  <xsl:merge-source select="doc('log1.xml')">
    <xsl:merge-input select="transactions/transaction"/>
    <xsl:merge-key select="xs:dateTime(@date, @time)" order="ascending"/>
  </xsl:merge-source>
  <xsl:merge-source select="doc('log2.xml')">
    <xsl:merge-input select="eventdata/transfer"/>
    <xsl:merge-key select="@timestamp" order="ascending"/>
  </xsl:merge-source>
  <xsl:merge-action>
    <xsl:apply-templates select="current-group()"/>
  </xsl:merge-action>
  </xsl:merge>
```

The function `current-merge-inputs()` is not yet implemented.

xsl:merge-action

See xsl:merge

xsl:merge-input

See xsl:merge

xsl:merge-source

See xsl:merge

xsl:message

The `xsl:message` element causes a message to be displayed. The message is the contents of the `xsl:message` element.

There is an optional attribute `terminate` with permitted values `yes` and `no`; the default is no. If the value is set to yes, processing of the stylesheet is terminated after issuing the message. This attribute may be supplied as an attribute value template.

By default the message is displayed on the standard error output stream. You can supply your own message receiver if you want it handled differently. This must be a class that implements the `Receiver [Javadoc: net.sf.saxon.event.Receiver]` interface. The content of the message is in general an XML fragment. You can supply the Receiver using the -m option on the command line, or the setMessageEmitter() method of the `Controller [Javadoc: net.sf.saxon.Controller]` class.

The sequence of calls to this Receiver is as follows: there is a single `open()` call at the start of the transformation, and a single `close()` call at the end; and each evaluation of an `xsl:message` instruction starts with a `startDocument()` call and ends with `endDocument()`. The `startDocument()` event has a `properties` argument indicating whether `terminate="yes"` was specified, and the `locationId` on calls such as `startElement()` and `characters()` can be used to identify the location in the stylesheet where the message data originated (this is achieved by passing the supplied `locationId` in a call to `getPipelineConfiguration().getLocator().getSystemId(locationId)`, or to `getLineNumber()` on the same object).

Select the class `MessageWarner [Javadoc: net.sf.saxon.serialize.MessageWarner]` to have `xsl:message` output notified to the JAXP `ErrorListener`, as described in the JAXP documentation.

xsl:mode

The `xsl:mode` declaration is new in XSLT 3.0. Previously, modes were declared implicitly by referring to them in the `mode` attribute of `xsl:template` or `xsl:apply-templates`. XSLT 3.0 introduces an `xsl:mode` declaration to allow properties of the mode to be defined.

The element always appears as a child of `xsl:stylesheet` (or `xsl:transform`), and it is empty (has no children).

The `name` attribute identifies the name of this mode; if omitted, the element describes the properties of the unnamed mode.

The attribute `streamable="yes"` indicates that template rules using this mode must be capable of being evaluated in a streaming manner. This imposes restrictions on the content of the template rules. For details, see Streaming of Large Documents. This option is available in Saxon-EE only.

The attribute `on-multiple-match` indicates what action is taken when a node being processed by `xsl:apply-templates` in this mode matches more than one template rule (with the same precedence and priority). The values are `fail` indicating that a dynamic error is reported, or `use-last` indicating that the template rule appearing last in document order is chosen.

The attribute `on-no-match` indicates what action is taken when a node being processed by `xsl:apply-templates` in this mode matches no template rule. The default value is `text-only-copy`. The permitted values are:

- text-only-copy: the XSLT 2.0 behaviour (for elements: apply-templates to the children; for text nodes: copy the text node to the output)
- shallow-copy: invoke the "identity template", which copies an element node and does apply-templates to its children
- deep-copy: invoke `xsl:copy-of`
- shallow-skip: ignores this node, does apply-templates to its children
- deep-skip: ignores this node and all its descendants
- fail: reports a dynamic error

The attribute `warning-on-multiple-match="yes"` causes a run-time warning when a node is matched by multiple template rules.

The attribute `warning-on-no-match="yes"` causes a run-time warning when a node is matched by no template rules.

xsl:namespace

The `xsl:namespace` instruction creates a namespace node. The `name` attribute defines the name of the namespace node (that is, the namespace prefix) while the content of the instruction defines the string value of the namespace node (that is, the namespace URI). The semantics thus parallel `xsl:attribute` which creates attribute nodes.

It is rarely necessary to use this instruction explicitly. The only cases it is needed are where the namespaces to be included in the result document are not known statically, and are not present in the source document.

xsl:namespace-alias

The `xsl:namespace-alias` element is a top-level element that is used to control the mapping between a namespace URI used in the stylesheet and the corresponding namespace URI used in the result document.

Normally when a literal result element is encountered in a template, the namespace used for the element name and attribute names in the result document is the same as the namespace used in the stylesheet. If a different namespace is wanted (e.g. because the result document is a stylesheet using the XSLT namespace), then xsl:namespace-alias can be used to define the mapping.

Example: This example allows the prefix outxsl to be used for output elements that are to be associated with the XSLT namespace. It assumes that both namespaces xsl and outxsl have been declared and are in scope.

```
<xsl:namespace-alias stylesheet-prefix="outxsl" result-prefix="xsl"/>
```

xsl:next-iteration

The `xsl:next-iteration` instruction is new in XSLT 3.0; it occurs within `xsl:iterate`. For details see xsl:iterate. The contents are a set of `xsl:with-param` elements defining the values of the iteration parameters to be used on the next iteration.

xsl:next-match

The instruction was introduced in XSLT 2.0. It is very similar to `xsl:apply-imports`, but with a different algorithm for choosing the next template to execute. It chooses the template rule that matches the current node and that would have been chosen if the current template rule and all higher precedence/ priority rules were not there.

xsl:non-matching-substring

The `xsl:non-matching-substring` element is used within an `xsl:analyze-string` element to indicate the default action to be taken with substrings that do not match a regular expression.

See xsl:analyze-string.

xsl:number

The `xsl:number` element outputs the sequential number of a node in the source document.

If the `select` attribute is present, this is an expression that selects the node to be numbered. The default is `.`, the context node.

The instruction takes an attribute `count` whose value is a pattern indicating which nodes to count; the default is to match all nodes of the same type and name as the current node.

The `level` attribute may take three values: "single", "any", or "multiple". The default is "single". This defines the rules for calculating a number.

There is also an optional `from` attribute, which is also a pattern. The exact meaning of this depends on the level.

The calculation is as follows:

Table 15.2.

level=single	If the current node matches the pattern, the counted node is the current node. Otherwise the counted node is the innermost ancestor of the current node that matches the pattern. If no ancestor matches the pattern, the result is zero. If the from attribute is present, the counted node must be a descendant of a node that matches the "from" pattern.The result is one plus the number of elder siblings of the counted node that match the count pattern.
level=any	The result is the number of nodes in the document that match the count pattern, that are at or before the current node in document order, and that follow in document order the most recent node that matches the "from" pattern, if any. Typically this is used to number, say, the diagrams or equations in a document, or in some section or chapter of a document, regardless of where the diagrams or equations appear in the hierarchic structure.
level=multiple	The result of this is not a single number, but a list of numbers. There is one number in the list for each ancestor of the current element that matches the count pattern and that is a descendant of the anchor element. Each number is one plus the number of elder siblings of the relevant element that match the count pattern. The order of the numbers is "outwards-in".

There is an optional `format` attribute which controls the output format. This contains an alternating sequence of format-tokens and punctuation-tokens. A format-token is any sequence of alphanumeric characters, a punctuation-token is any other sequence. The following values (among others) are supported for the format-token:

Table 15.3.

1	Sequence 1, 2, 3, ... 10, 11, 12, ...
001	Sequence 001, 002, 003, ... 010, 011, 012, ... (any number of leading zeroes)
a	Sequence a, b, c, ... aa, ab, ac, ...
A	Sequence A, B, C, ... AA, AB, AC, ...
i	Sequence i, ii, iii, iv, ... x, xi, xii, ...
I	Sequence I, II, III, IV, ... X, XI, XII, ...

There is also support for various Japanese sequences (Hiragana, Katakana, and Kanji) using the format tokens &#x3042, &#x30a2, &#x3044, &#x30a4, &#x4e00, and for Greek and Hebrew sequences.

The format token "w" gives the sequence "one", "two", "three", ... , while "W" gives the same in upper-case, and "Ww" in title case. The language is determined by the `lang` attribute, for example `lang="en"` for English. Saxon currently supports Belgian French (fr-BE), Danish (da), Dutch (nl), English (en), Flemish (nl-BE), French (fr), German (de), Italian (it), Swedish (sw). Additional languages can be achieved by writing a `Numberer` class, as described in Localizing numbers and dates.

The default format is "1".

A sequence of Unicode digits other than ASCII digits (for exaple, Tibetan digits) can be used, and will result in decimal numbering using those digits.

Similarly, any other character classified as a letter can be used, and will result in "numbering" using all consecutive Unicode letters following the one provided. For example, specifying "x" will give the sequence x, y, z, xx, xy, xz, yx, yy, yz, etc. Specifying the Greek letter alpha (&#178;) will cause "numbering" using the Greek letters up to "Greek letter omega with tonos" (&#206;). Only "i" and "I" (for roman numbering), and the Japanese characters listed above, are exceptions to this rule.

Successive format-tokens in the format are used to process successive numbers in the list. If there are more format-tokens in the format than numbers in the list, the excess format-tokens and punctuation-tokens are ignored. If there are fewer format-tokens in the format than numbers in the list, the last format-token and the punctuation-token that precedes it are used to format all excess numbers, with the final punctuation-token being used only at the end.

Examples:

Table 15.4.

Number(s)	Format	Result
3	(1)	(3)
12	I	XII
2,3	1.1	2.3
2,3	1(i)	2(iii)
2,3	1.	2.3.
2,3	A.1.1	B.3.
2,3,4,5	1.1	2.3.4.5

This character may be preceded or followed by arbitrary punctuation (anything other than these characters or XML special characters such as "<") which is copied to the output verbatim. For example, the value 3 with format "(a)" produces output "(c)".

It is also possible to use `xsl:number` to format a number obtained from an expression. This is achieved using the value attribute of the `xsl:number` element. If this attribute is present, the count, level, and from attributes are ignored.

With large numbers, the digits may be split into groups. For example, specify grouping-size="3" and grouping-separator="/" to have the number 3000000 displayed as "3/000/000".

Negative numbers are always output in conventional decimal notation, regardless of the format specified.

Example: This example outputs the title child of an H2 element preceded by a composite number formed from the sequential number of the containing H1 element and the number of the containing H2 element.

```
<xsl:template match="H2/TITLE">
        <xsl:number count="H1">.<xsl:number count="H2">
        <xsl:text> </xsl:text>
        <xsl:apply-templates/>
</xsl:template>
```

xsl:on-completion

The `xsl:on-completion` instruction is new in XSLT 3.0; it occurs within `xsl:iterate`. For details see xsl:iterate. During execution of `xsl:on-completion` there is no context item, position or size; the instruction has access to the iteration parameters with the values given on the last iteration (or the initial values of the `xsl:param` elements if the input sequence was empty).

xsl:otherwise

The `xsl:otherwise` element is used within an `xsl:choose` element to indicate the default action to be taken if none of the other choices matches.

See xsl:choose.

xsl:output

The `xsl:output` element is used to control the format of serial output files resulting from the transformation. It is always a top-level element immediately below the`xsl:stylesheet` element. There may be multiple `xsl:output` elements; their values are accumulated as described in the XSLT specification.

The following standard attributes may be specified:

Table 15.5.

name	This provides a name for this output format, which may be referenced in the `xsl:result-document` element. By default, the unnamed output format is used.
method	This indicates the format or destination of the output. The value "xml" indicates XML output (though if disable-output-escaping or character maps are used there is no guarantee that it is well-formed). A value of "html" is used for HTML output, and "xhtml" for XHTML. The value "text" indicates plain text output: in this case no markup may be written to the file using constructs such as literal result elements, xsl:element, xsl:attribute, or xsl:comment.Alternatively output can be directed to a user-defined Java program by specifying the name of the class as the value of the method attribute, prefixed by a namespace prefix, for example "xx:com.me.myjava.MyEmitter". The class must be on the classpath, and must implement either the `org.xml.sax.ContentHandler` interface, or the `Receiver [Javadoc: net.sf.saxon.event.Receiver]` interface. The last of these, though proprietary, is a richer interface that gives access to additional information.
cdata-section-elements	This is used only for XML output. It is a whitespace-separated list of element names. Character data belonging to these output elements will be written within CDATA sections.
doctype-system	This is used only for XML output: it is copied into the DOCTYPE declaration as the system identifier. If the value is an empty string, Saxon interprets this as if the attribute were omitted, which can be useful it you want to override an actual value with "absent".
doctype-public	This is used only for XML output: it is copied into the DOCTYPE declaration as the public

	identifier. It is ignored if there is no system identifier. If the value is an empty string, Saxon interprets this as if the attribute were omitted, which can be useful it you want to override an actual value with "absent".
encoding	A character encoding, e.g. iso-8859-1 or utf-8. The value must be one recognised both by the Java run-time system and by Saxon itself: the encoding names that Saxon recognises are ASCII, US-ASCII, iso-8859-1, utf-8, utf8, KOI8R, cp1251. It is used for three distinct purposes: to control character conversion by the Java I/O routines; to determine which characters will be represented as character entities; and to document the encoding in the output file itself. The default (and fallback) is utf-8.
escape-uri-attributes	New in XSLT 2.0: values "yes" or "no" are accepted. This affects HTML output only. It controls whether non-ASCII characters in HTML URI-valued attributes (for example, `href`) are escaped using the %HH convention. The default is "yes".
include-content-type	New in XSLT 2.0: values "yes" or "no" are accepted. This affects HTML output only. It controls whether a `meta` tag is inserted into the HTML `head` element. The default is "yes".
indent	as in the XSLT spec: values "yes" or "no" are accepted. The indentation algorithm is different for HTML and XML. For HTML it avoids outputting extra space before or after an inline element, but will indent text as well as tags, except in elements such as PRE and SCRIPT. For XML, it avoids outputting extra whitespace except between two tags. The emphasis is on conformance rather than aesthetics!
suppress-indentation	This is a new property in XSLT 3.0 (it was previously available in Saxon as an extension). The value is a whitespace-separated list of element names, and it typically identifies "inline" elements that should not cause indentation; in XHTML, for example, these would be `b`, `i`, `span`, and the like.
media-type	For example, "text/xml" or "text/html". This is largely documentary. However, the value assigned is passed back to the calling application in the OutputDetails object, where is can be accessed using the getMediaType() method. The supplied servlet application SaxonServlet uses this to set the media type in the HTTP header.
omit-xml-declaration	The values are "yes" or "no". For XML output this controls whether an xml declaration should be output; the default is "no".

standalone	This is used only for XML output: if it is present, a standalone attribute is included in the XML declaration, with the value "yes" or "no".
use-character-maps	A space-separated list of the names of character maps (see xsl:character-map) which will be applied to transform individual characters during serialization.
version	Determines the version of XML or HTML to be output. This is largely documentary. However, for XML the distinction between "1.0" and "1.1" determines whether or not namespace undeclarations will be output; and for HTML, the value "5" can be used to force the HTML5 style of DOCTYPE declaration.

See Additional Serialization Parameters for descriptions of additional attributes supported by Saxon on the `xsl:output` declaration.

xsl:output-character

This element defines one entry in a character map. See xsl:character-map for further details.

xsl:param

The `xsl:param` element is used to define a formal parameter to a template, or to the stylesheet.

As a template parameter, it must be used as an immediate child of the `xsl:template` element. As a stylesheet parameter, it must be used as an immediate child of the `xsl:stylesheet` element.

There is a mandatory attribute, `name`, to define the name of the parameter. The default value of the parameter may be defined either by a `select` attribute, or by the contents of the `xsl:param` element, in the same way as for `xsl:variable`. The default value is ignored if an actual parameter is supplied with the same name.

There is an optional attribute, `as`, to define the type of the parameter. The actual supplied parameter will be converted to this type if required. If the parameter is omitted, the default value must conform to the type. Note that if no default is specified, the default is a zero-length string, which may conflict with the required type.

The `type-information` attribute is removed at Saxon 7.5

The `required` attribute can take the values "yes" or "no". This isn't allowed for function parameters, which are always required. If the parameter is required, no default value may be specified. Failure to supply a value for a required parameter gives a run-time error (the specification says that in the case of call-template, it should be a static error).

In XSLT 3.0, `xsl:param` can also appear as a child of `xsl:iterate`.

xsl:perform-sort

The `xsl:perform-sort` instruction takes a sequence as its input and produces a sorted sequence as its output.

The input sequence may be specified either using the `select` attribute, or using the instructions contained within the `xsl:perform-sort` instruction.

The sort criteria are specified using `xsl:sort` elements as children of `xsl:perform-sort`, in the usual way.

For example:

```
<xsl:perform-sort select="//BOOK">
    <xsl:sort select="author/last-name"/>
    <xsl:sort select="author/first-name"/>
</xsl:perform-sort>
```

It's often useful to use `xsl:perform-sort` inside a stylesheet function; the function can return the sorted sequence as its result, and can be invoked directly from an XPath expression.

xsl:preserve-space

The `xsl:preserve-space` element is used at the top level of the stylesheet to define elements in the source document for which white-space nodes are significant and should be retained.

The `elements` attribute is mandatory, and defines a space-separated list of element names. The value "*" may be used to mean "all elements"; in this case any elements where whitespace is not to be preserved may be indicated by an xsl:strip-space element.

xsl:processing-instruction

The `xsl:processing-instruction` element can appear anywhere within an `xsl:template`. It causes an XML processing instruction to be output.

There is a mandatory `name` attribute which gives the name of the PI. This attribute is interpreted as an , so it may contain string expressions within curly braces.

The data part of the PI may be given either by a `select` attribute or by an enclosed sequence constructor. If the `select` attribute is used and the value is a sequence, then the items in the sequence are output space-separated.

For example:

```
<xsl:processing-instruction name="submit-invoice">version="1.0"</xsl:processing
```

Note that special characters occurring within the PI text will be escaped.

xsl:result-document

The `xsl:result-document` element is new in XSLT 2.0, and replaces the previous extension element `saxon:output`. It is used to direct output to a secondary output destination.

The `format` attribute is optional. If present, it gives the name of an `xsl:output` element that describes the serialization format for this output document; if absent, the unnamed `xsl:output` declaration is used.

The `href` attribute gives the URI for the result document. If this is a relative URI, it is interpreted relative to the base output URI. The base output URI is the systemID of the Result object supplied as the destination for the transformation, or if you are using the command line, the value of the `-o` flag. If the `href` attribute is omitted, the document is written to the location identified by the base output URI: this will only work if all the output produced by the stylesheet is within the scope of an `xsl:result-document` instruction.

If the base output URI is not known, then the current directory is used, unless the configuration disables calling of extension functions, in which case it is assumed that the stylesheet is not trusted to overwrite files relative to the current directory, and an error is then reported.

This base output URI must be a writable location. Usually it will therefore be a URI that uses the "file:" scheme. However, Saxon attempts to open a connection whatever URI scheme is used, and it should therefore work with any URI where the Java VM has the capability to open a writable connection. Users have reported success in using "ftp:" and "mailto:" URIs.

The optional `validation` and `type` attributes determine what happens to any type annotations on element or attribute nodes. These values must not be used in the basic Saxon product.

The `xsl:result-document` instruction may also take serialization attributes such as `method`, `indent`, or `saxon:indent-spaces`. These attributes may be AVTs, so the values can be decided at run-time. Any values specified on the `xsl:result-document` instruction override the values specified on the `xsl:output` declaration.

Here is an example that uses xsl:result-document:

```
<xsl:template match="preface">
    <xsl:result-document href="{$dir}/preface.html" method="html">
        <html><body bgcolor="#00eeee"><center>
            <xsl:apply-templates/>
        </center><hr/></body></html>
    </xsl:result-document>
    <a href="{$dir}/preface.html">Preface</a>
</xsl:template>
```

Here the body of the preface is directed to a file called preface.html (prefixed by a constant that supplies the directory name). Output then reverts to the previous destination, where an HTML hyperlink to the newly created file is inserted.

xsl:sequence

The `xsl:sequence` element is new in XSLT 2.0; it used to construct arbitrary sequences. It may select any sequence of nodes and/or atomic values, and essentially adds these to the result sequence. The input may be specified either by a `select` attribute, or by the instructions contained in the `xsl:sequence` instruction, or both (the `select` attribute is processed first). Nodes and atomic values are included in the result sequence directly. Unlike `xsl:copy-of`, no copy is made.

There are two main usage scenarios. The first is copying atomic values into a tree. For example:

```
<e>
    <xsl:sequence select="1 to 5"/>
    <br/>
    <xsl:sequence select="6 to 10"/>
</e>
```

which produces the output `<e>1 2 3 4 5<br/>6 7 8 9 10</e>`.

The second, more important, is constructing a sequence-valued variable. A variable is sequence-valued if the variable binding element (e.g. `xsl:variable` has non-empty content, an `as` attribute, and no `select` attribute. For example:

```
<xsl:variable name="seq" as="xs:integer *">
    <xsl:for-each select="1 to 5">>
       <xsl:sequence select=". * ."/>
    </xsl:for-each/>
</xsl:variable>
```

This produces the sequence (1, 4, 9, 16, 25) as the value of the variable.

The `xsl:sequence` instruction may be used to produce any sequence of nodes and/or atomic values.

If nodes are constructed within a sequence-valued variable, they will be . For example, the following code creates a variable whose value is a sequence of three parentless attributes:

```
<xsl:variable name="seq" as="attribute() *">
    <xsl:attribute name="a">10</xsl:attribute>
    <xsl:attribute name="b">20</xsl:attribute>
    <xsl:attribute name="a">30</xsl:attribute>
</xsl:variable>
```

It is quite legitimate to have two attributes in the sequence with the same name; there is no conflict until an attempt is made to add them both to the same element. The attributes can be added to an element by using `<xsl:copy-of select="$seq"/>` within an `xsl:element` instruction or within a literal result element. At this stage the usual rule applies: if there are duplicate attributes, the last one wins.

xsl:sort

The `xsl:sort` element is used within an `xsl:for-each` or `xsl:apply-templates` or `saxon:group` element to indicate the order in which the selected elements are processed.

The `select` attribute (default value ".") is a string expression that calculates the sort key.

The `order` attribute (values "ascending" or "descending", default "ascending") determines the sort order. There is no control over language, collating sequence, or data type.

The `data-type` attribute determines whether collating is based on alphabetic sequence or numeric sequence. The permitted values are either "text" or "number", or a built-in type in XML Schema, such as `xs:date` or `xs:decimal`.

The `collation` attribute is the name of a collating sequence. If present it must be a collation URI recognized by Saxon: see Implementing a collating sequence.

The `case-order` attribute (values "upper-first" and "lower-first") is relevant only for data-type="text"; it determines whether uppercase letters are sorted before their lowercase equivalents, or vice-versa.

The value of the `lang` attribute can be an ISO language code such as "en" (English) or "de" (German). It determines the algorithm used for alphabetic collating. The default is based on the Java system locale. The value is used to select a collating sequence associated with the Java Locale for that language.

Several sort keys are allowed: they are written in major-to-minor order.

Example 1: sorting with xsl:apply-templates. This example shows a template for a BOOKLIST element which processes all the child BOOK elements in order of their child AUTHOR elements; books with the same author are in descending order of the DATE attribute.

```
<xsl:template match="BOOKLIST">
    <h2>
        <xsl:apply-templates select="BOOK">
            <xsl:sort select="AUTHOR"/>
            <xsl:sort select="@DATE" order="descending" lang="GregorianDate"/>
        </xsl:apply-templates>
    </h2>
</xsl:template>
```

Example 2: sorting with xsl:for-each. This example also shows a template for a BOOKLIST element which processes all the child BOOK elements in order of their child AUTHOR elements.

```
<xsl:template match="BOOKLIST">
    <h2>
```

```
        <xsl:for-each select="BOOK">
            <xsl:sort select="AUTHOR"/>
            <p>AUTHOR: <xsl:value-of select="AUTHOR"></p>
            <p>TITLE: <xsl:value-of select="TITLE"></p>
            <hr/>
        </xsl:for-each>
    </h2>
</xsl:template>
```

xsl:strip-space

The `xsl:strip-space` element is used at the top level of the stylesheet to define elements in the source document for which white-space nodes are insignificant and should be removed from the tree before processing.

The `elements` attribute is mandatory, and defines a space-separated list of element names. The value "*" may be used to mean "all elements"; in this case any elements where whitespace is not to be stripped may be indicated by an xsl:preserve-space element.

xsl:stylesheet

The `xsl:stylesheet` element is always the top-level element of an XSLT stylesheet. The name `xsl:transform` may be used as a synonym.

The following attributes may be specified:

Table 15.6.

version	Mandatory. A value other than "1.0" invokes forwards compatibility mode.
saxon:trace	Value "yes" or "no": default no. If set to "yes", causes activation of templates to be traced on System.err for diagnostic purposes. The value may be overridden by specifying a saxon:trace attribute on the individual template.

xsl:template

The `xsl:template` element defines a processing rule for source elements or other nodes of a particular type.

The type of node to be processed is identified by a pattern, written in the mandatory `match` attribute. The most common form of pattern is simply an element name. However, more complex patterns may also be used: The syntax of patterns is given in more detail in XSLT Pattern Syntax

The following examples show some of the possibilities:

Table 15.7.

XXX	Matches any element whose name (tag) is XXX
	Matches any element
XXX/YYY	Matches any YYY element whose parent is an XXX
XXX//YYY	Matches any YYY element that has an ancestor named XXX

/*/XXX	Matches any XXX element that is immediately below the root (document) element
*[@ID]	Matches any element with an ID attribute
XXX[1]	Matches any XXX element that is the first XXX child of its parent element. (Note that this kind of pattern can be very inefficient: it is better to match all XXX elements with a single template, and then use xsl:if to distinguish them)
SECTION[TITLE="Contents"]	Matches any SECTION element whose first TITLE child element has the value "Contents"
A/TITLE | B/TITLE | C/TITLE	Matches any TITLE element whose parent is of type A or B or C
text()	Matches any character data node
@*	Matches any attribute
/	Matches the document node

The `xsl:template` element has an optional `mode` attribute. If this is present, the template will only be matched when the same mode is used in the invoking `xsl:apply-templates` element. The value can be a list of mode names, indicating that the template matches more than one mode; this list can include the token `#default` to indicate that the template matches the default (unnamed) mode. Alternatively the `mode` attribute can be set to `#all`, to indicate that the template matches all modes. (This can be useful in conjunction with `xsl:next-match`: one can write a template rule that matches in all modes, and then call `xsl:next-match` to continue processing in the original mode.)

There is also an optional `name` attribute. If this is present, the template may be invoked directly using `xsl:call-template`. The match attribute then becomes optional.

If there are several `xsl:template` elements that all match the same node, the one that is chosen is determined by the optional `priority` attribute: the template with highest priority wins. The priority is written as a floating-point number; the default priority is 1. If two matching templates have the same priority, the one that appears last in the stylesheet is used.

Examples:

The following examples illustrate different kinds of template and match pattern.

: a simple XSLT template for a particular element. This example causes all <ptitle> elements in the source document to be output as HTML <h2> elements.

```
<xsl:template match="ptitle">
    <h2>
        <xsl:apply-templates/>
    </h2>
</xsl:template>
```

xsl:text

The `xsl:text` element causes its content to be output.

The main reason for enclosing text within an `xsl:text` element is to allow white space to be output. White space nodes in the stylesheet are ignored unless they appear immediately within an `xsl:text` element.

The optional `disable-output-escaping` attribute may be set to "yes" or "no"; the default is "no". If set to "yes", special characters such as "<" and "&" will be output as themselves, not as entities.

Be aware that in general this can produce non-well-formed XML or HTML. It is useful, however, when generating things such as ASP or JSP pages. Escaping may not be disabled when writing to a result tree fragment.

xsl:try

The `xsl:try` instruction is new in XSLT 3.0: in conjunction with `xsl:catch` it allows recovery from dynamic errors.

The following example shows how to recover from an error in evaluating an XPath expression (in this case, divide-by-zero):

```
<xsl:try select="salary div length-of-service">
  <xsl:catch errors="err:FAOR0001" select="()"/>
  </xsl:try>
```

The following example shows how to recover from an error in evaluating a sequence of XSLT instructions (in this case, a validation error):

```
<xsl:try>
    <xsl:copy-of select="$result" validation="strict"/>
    <xsl:catch>
      <xsl:message>Warning: validation of result document failed:
          Error code: <xsl:value-of select="$err:code"/>
          Reason: <xsl:value-of select="$err:description"/>
      </xsl:message>
      <xsl:sequence select="$result"/>
    </xsl:catch>
</xsl:try>
```

The `errors` attribute of `xsl:catch` indicates which error codes are caught. If absent, or it set to `*`, all errors are caught. The value can be a whitespace-separated list of QNames; the wildcards `*:local` or `prefix:*` can also be used.

It is possible to have more than one `xsl:catch` within an `xsl:try`; the first one that matches the error is used.

Within the `xsl:catch`, a number of variables are available in the namespace `http://www.w3.org/2005/xqt-errors`:

- err:code - the error code as a QName
- err:description - the error description (error message)
- err:value - the error object (if available)
- err:module - the URI of the stylesheet module in which the error occurred
- err:line-number - the line-number of the source stylesheet where the error occurred
- err:column-number - for Saxon this will generally be unknown (-1)

The error can be re-thrown by using the `error()` function.

xsl:value-of

The `xsl:value-of` element evaluates an expression as a string, and outputs its value to the current result tree.

The full syntax of expressions is outlined in XPath Expression Syntax.

The `select` attribute identifes the expression. If this is not specified, the value to be output is obtained by evaluating the sequence constructor contained within the `xsl:value-of` element.

The optional `disable-output-escaping` attribute may be set to "yes" or "no"; the default is "no". If set to "yes", special characters such as "<" and "&" will be output as themselves, not as entities. Be aware that in general this can produce non-well-formed XML or HTML. It is useful, however, when generating things such as ASP or JSP pages. Escaping may not be disabled when writing to a result tree fragment.

If the `select` expression evaluates to a sequence containing more than one item, the result depends on whether a `separator` attribute is present. If the `separator` is absent when running in 1.0 mode, then only the first item is considered. When running in 2.0 mode, all the items are output. The separator defaults to a single space if the `select` attribute is used, or to a zero-length string if a sequence constructor is used. The `separator` attribute may be specified as an .

Here are some examples of expressions that can be used in the select attribute:

Table 15.8.

TITLE	The character content of the first child TITLE element if there is one
@NAME	The value of the NAME attribute of the current element if there is one
.	The expanded character content of the current element
../TITLE	The expanded character content of the first TITLE child of the parent element, if there is one
ancestor::SECTION/TITLE	The expanded character content of the first TITLE child of the enclosing SECTION element, if there is one
ancestor::*/TITLE	The expanded character content of the first TITLE child of the nearest enclosing element that has a child element named TITLE
PERSON[@ID]	The content of the first child PERSON element having an ID attribute, if there is one
*[last()]/@ID	The value of the ID attribute of the last child element of any type, if there are any
.//TITLE	The content of the first descendant TITLE element if there is one
sum(*/@SALES)	The numeric total of the values of the SALES attributes of all child elements that have a SALES attribute

xsl:variable

The `xsl:variable` element is used to declare a variable and give it a value. If it appears at the top level (immediately within xsl:stylesheet) it declares a global variable, otherwise it declares a local variable that is visible only within the stylesheet element containing the xsl:variable declaration.

The mandatory `name` attribute defines the name of the variable.

The value of the variable may be defined either by an expression within the optional `select` attribute, or by the contents of the xsl:variable element. In the latter case the result is a temporary tree. A temporary tree can be used like a source document, for example it can be accessed using path expressions and processed using template rules.

There is an optional attribute, `as`, to define the type of the variable. The actual supplied value must be an instance of this type; it will not be converted.

In standard XSLT, variables once declared cannot be updated. Saxon however provides a saxon:assign extension element to circumvent this restriction.

The value of a variable can be referenced within an expression using the syntax `$name`.

Example:

```
<xsl:variable name="title">A really exciting document"</xsl:variable>
<xsl:variable name="backcolor" expr="'#FFFFCC'" />
<xsl:template match="/*">
    <HTML><TITLE<xsl:value-of select="$title"/></TITLE>
    <BODY BGCOLOR='{$backcolor}'>
    ...
    </BODY></HTML>
</xsl:template>
```

xsl:when

The `xsl:when` element is used within an `xsl:choose` element to indicate one of a number of choices. It takes a mandatory parameter, `test`, whose value is a match pattern. If this is the first xsl:when element within the enclosing xsl:choose whose test condition matches the current element, the content of the xsl:when element is expanded, otherwise it is ignored.

xsl:with-param

The `xsl:with-param` element is used to define an actual parameter to a template. It may be used within an `xsl:call-template` or an `xsl:apply-templates` or an `xsl:apply-imports` element. For an example, see the xsl:template section.

There is a mandatory attribute, `name`, to define the name of the parameter. The value of the parameter may be defined either by a select attribute, or by the contents of the `xsl:param` element, in the same way as for `xsl:variable`.

The attribute `tunnel="yes"` creates a tunnel parameter which is accessible to called templates at any depth, whether or not they are declared in intermediate templates. However, the value is only accessible if `tunnel="yes"` is also specified on the corresponding `xsl:param` element.

In XSLT 3.0, `xsl:with-param` can also appear as a child of `xsl:evaluate`, to define variables available for use within the dynamically-evaluated XPath expression, and as a child of `xsl:next-iteration`, to define values of iteration parameters to be used on the next iteration.

Literal Result Elements

Any elements in the style sheet other than those listed above are assumed to be literal result elements, and are copied to the current output stream at the position in which they occur.

Attribute values within literal result elements are treated as attribute value templates: they may contain string expressions enclosed between curly braces. For the syntax of string expressions, see above.

Where the output is HTML, certain formatting elements are recognised as empty elements: these are AREA, BASEFONT, BR, COL, FRAME, HR, IMG, INPUT, ISINDEX, LINK, META, and SYSTEM (in either upper or lower case, and optionally with attributes, of course). These should be written as empty XML elements in the stylesheet, and will be written to the HTML output stream without a closing tag.

With HTML output, if the attribute name is the same as its value, the abbreviated form of output is used: for example if <OPTION SELECTED="SELECTED"> appears in the stylesheet, it will be output as <OPTION SELECTED>.

A simple stylesheet may be created by using a literal result element as the top-level element of the stylesheet. This implicitly defines a single template with a match pattern of "/". In fact, an XHTML document constitutes a valid stylesheet which will be output as a copy of itself, regardless of the contents of the source XML document.

XSLT Patterns

This section gives an informal description of the syntax of XSLT patterns. For a formal specification, see the XSLT recommendation. Pattern syntax did not change significantly in XSLT 2.0, except by allowing any XPath 2.0 expression to be used within a predicate. XSLT 3.0 introduces extension, described below, to allow atomic values as well as nodes to be matched, and by removing a number of restrictions. Some of these changes (but not all) are implemented in Saxon 9.4 when XSLT 3.0 processing is enabled.

Patterns define a condition that a node may or may not satisfy: a node either matches the pattern, or it does not. The syntax of patterns is a subset of that for the XPath expressions, and formally, a node matches a pattern if it is a member of the node set selected by the corresponding expression, with some ancestor of the node acting as the current node for evaluating the expression. For example a TITLE node matches the pattern "TITLE" because it is a member of the node set selected by the expression "TITLE" when evaluated at the immediate parent node.

In XSLT stylesheets, patterns are used primarily in the `match` attribute of the `xsl:template` element. They are also used in the `count` and `from` attributes of `xsl:number`, the `match` attribute of `xsl:key`, and the `group-starting-with` and `group-ending-with` attributes of `xsl:for-each-group`.

The next page gives some examples of match patterns and their meaning. This is followed by a page that gives a summary of the XSLT 2.0 syntax, and another page that describes the extensions to patterns in XSLT 3.0.

- Examples of XSLT 2.0 Patterns
- Pattern syntax
- Patterns in XSLT 3.0

Examples of XSLT 2.0 Patterns

The table below gives some examples of patterns, and explains their meaning:

Table 15.9.

PARA	Matches any element whose name (tag) is PARA
	Matches any element
APPENDIX/PARA	Matches any PARA element whose parent is an APPENDIX
APPENDIX//PARA	Matches any PARA element that has an ancestor named APPENDIX

/*/SECTION	Matches any SECTION element that is an immediate child of the outermost element in the document
*[@NAME]	Matches any element with a NAME attribute
SECTION/PARA[1]	Matches any PARA element that is the first PARA child of a SECTION element
SECTION[TITLE="Contents"]	Matches any SECTION element whose first TITLE child element has the value "Contents"
A/TITLE | B/TITLE | C/TITLE	Matches any TITLE element whose parent is of type A or B or C (Note that this cannot be written "(A|B|C)/TITLE", although that is a valid XPath 2.0 path expression.)
/BOOK//*	Matches any element in a document provided the top-level element in the document is named "BOOK"
A/text()	Matches the character content of an A element
A/@*	Matches any attribute of an A element

In a schema-aware stylesheet, it can be useful to match elements by their schema-defined type, rather than by their name. The following table shows some examples of this.

Table 15.10.

schema-element(CHAPTER)	Matches any element that has been validated against the global element declaration named CHAPTER. More precisely, it matches an element named CHAPTER that has been validated against the type of the global element declaration named CHAPTER, and any element in the substitution group of the CHAPTER element.
element(*, ADDRESS-TYPE)	Matches any element that has been validated against the schema-defined global type definition ADDRESS-TYPE. The "*" indicates that there are no constraints on the element's name.
attribute(*, xs:date)	Matches any attribute that has been validated as an instance of xs:date, including types derived by restriction from xs:date.

Pattern syntax

Saxon supports the full XSLT syntax for patterns. The rules below describe a simplified form of this syntax (for example, it omits the legal but useless pattern '@comment()'):

```
pattern          ::= path ( '|' path )*
path             ::= anchor? remainder? (Note 1)

anchor           ::= '/' | '//' | id | key
id               ::= 'id' '(' value ')'
key              ::= 'key' '(' literal ',' value ')'
value            ::= literal | variable-reference

remainder        ::= path-part ( sep path-part )*
```

```
sep              ::= '/' | '//'
path-part        ::= node-match predicate+
node-match       ::= kind-match | type-match
kind-match       ::= element-match |
                     text-match |
      attribute-match |
      pi-match |
      any-node-match
element-match    ::= 'child::'? ( name | '*' )
text-match       ::= 'text' '(' ')'
attribute-match  ::= ('attribute::' | '@') ( name | '*' )
pi-match         ::= 'processing-instruction' '(' literal? ')'
any-node-match   ::= 'node' '(' ')'

type-match       ::= ('element'|'attribute')
                        '(' ('*'|node-name) (',' type-name) ')'

predicate        ::= '[' ( boolean-expression |
                           numeric-expression ) ']'
```

Note 1: not all combinations are allowed. If the anchor is '//' then the remainder is mandatory.

The form of a literal is as defined in expressions; and a predicate is itself a boolean or numeric expression. As with predicates in expressions, a numeric predicate `[P]` is shorthand for the boolean predicate `[position()=P]`.

Informally, a pattern consists of either a single path or a sequence of paths separated by vertical bars. An element matches the match-pattern if it matches any one of the paths.

A path consists of a sequence of path-parts separated by either "/" or "//". There is an optional separator ("/" or "//") at the start; a "//" has no effect and can be ignored. The last path-part may be an element-match, a text-match, an attribute-match, a pi-match, or a node-match; in practice, a path-part other than the last should be an element-match.

The axis syntax `child::` and `attribute::` may also be used in patterns, as described in the XSLT specification.

Patterns in XSLT 3.0

XSLT 3.0 extensions to patterns that are implemented in Saxon 9.4 include the following:

Pattern syntax in the form `~ItemType` is supported, for example `match="~xs:integer"` matches an integer. Predicates are allowed on such patterns, for example `~xs:integer[. gt 0]`.

The `intersect` and `except` operators are allowed at the top level: for example `match="* except br"`.

Parentheses may be used in conjunction with a predicate, for example `match="(//para)[1]"`

Any downwards axis may be used in a pattern, for example `descendant` or `descendant-or-self`.

Other XSLT 3.0 extensions to patterns have yet to be implemented in Saxon, for example patterns that start with (or consist exclusively of) a variable reference.

Chapter 16. XPath 2.0 Expression Syntax

Introduction

This document is an informal guide to the syntax of XPath 2.0 expressions, which are used in Saxon both within XSLT stylesheets, and in the Java API. XPath is also a subset of XQuery. For formal specifications, see the XPath 2.0 specification [http://www.w3.org/TR/xpath20/], except where differences are noted here.

There is also a summary of new features introduced in XPath 3.0 (originally published as XPath 2.1).

Saxon 9.4 implements XPath extensions to define maps: these extensions have been proposed by the XSL Working Group, but not yet accepted by the XQuery Working Group. Details are given here: Maps in XPath 3.0

XPath expressions may be used either in an XSLT stylesheet, or as a parameter to various Java interfaces. The syntax is the same in both cases. Saxon supports XPath either through the standard JAXP interface (which is somewhat limiting as it is designed for XPath 1.0 and is not well integrated with interfaces for XSLT and XQuery), or via Saxon's own s9api interface ("snappy"). In s9api, the steps are:

1. Create a `Processor` (which can also be used for XSLT, XQuery, and XSD processing)
2. User the `newXPathCompiler()` method to create an `XPathCompiler`, and use its methods to set the static context for the expression
3. User the `compile()` method to compile the expression, and the load() method to instantiate it for use (you can compile the method once and load it as many times as you like for execution)
4. Call methods on the resulting `XPathSelector` object to set the context item and the values of any external variables for evaluating the expression
5. Call another method to evaluate the expression. Because `XPathSelector` is a Java `Iterable`, you can simply use the Java for-each construct to iterate over the results of the expression.
6. If the results are sequences of nodes, as is often the case, they are returned as instances of the Saxon class `XdmNode` which provides methods to perform further processing of the results.

An important change in XPath 2.0 is that all values are now considered as sequences. A sequence consists of zero or more items; an item may be a node or a simple-value. Examples of simple-values are integers, strings, booleans, and dates. A single value such as a number is considered as a sequence of length 1. The empty sequence is written as `()`; a singleton sequence may be written as `"a"` or `("a")`, and a general sequence is written as `("a", "b", "c")`.

The node-sets of XPath 1.0 are replaced in XPath 2.0 by sequences of nodes. Path expressions will return node sequences whose nodes are in document order with no duplicates, but other kinds of expression may return sequences of nodes in any order, with duplicates permitted.

This section summarizes the syntactic constructs and operators provided in XPath 2.0. The functions provided in the function library are listed separately: see the Functions section.

Constants

are written as "London" or 'Paris'. In each case you can use the opposite kind of quotation mark within the string: 'He said "Boo"', or "That's rubbish". In a stylesheet XPath expressions always appear within

XML attributes, so it is usual to use one kind of delimiter for the attribute and the other kind for the literal. Anything else can be written using XML character entities. In XPath 2.0, string delimiters can be doubled within the string to represent the delimiter itself: for example `<xsl:value-of select='"He said, ""Go!"""'/>`

follow the Java rules for decimal literals: for example, `12` or `3.05`; a negative number can be written as (say) `-93.7`, though technically the minus sign is not part of the literal. (Also, note that you may need a space before the minus sign to avoid it being treated as a hyphen within a preceding name). The numeric literal is taken as a double precision floating point number if it uses scientific notation (e.g. `1.0e7`), as fixed point decimal if it includes a full stop, or as a integer otherwise. Decimal values (and integers) in Saxon have unlimited precision and range (they use the Java classes BigInteger and BigDecimal internally). Note that a value such as `3.5` was handled as a double-precision floating point number in XPath 1.0, but as a decimal number in XPath 2.0: this may affect the precision of arithmetic results.

There are no boolean constants as such: instead use the function calls `true()` and `false()`.

Constants of other data types can be written using constructors, which look like function calls but require a string literal as their argument. For example, `xs:float("10.7")` produces a single-precision floating point number. Saxon implements constructors for all of the built-in data types defined in XML Schema Part 2.

An example for and values: you can write constants for these data types as `xs:date("2002-04-30")` or `xs:dateTime("1966-07-31T15:00:00Z")`.

XPath 2.0 allows the argument to a constructor to contain whitespace, as determined by the whitespace facet for the target data type.

Variable References

The value of a variable (local or global variable, local or global parameter) may be referred to using the construct `$`, where is the variable name.

The variable is always bound at the textual place where the expression containing it appears; for example a variable used within an `xsl:attribute-set` must be in scope at the point where the attribute-set is defined, not the point where it is used.

A variable may be declared to be of a particular type, for example it may be constrained to be an integer, or a sequence of strings, or an attribute node. In a schema-aware environment, this may also be a reference to a user-defined type in a schema. If there is no type declared for the variable, it may take a value of any data type, and in general it is not possible to determine its data type statically.

It is an error to refer to a variable that has not been declared.

Starting with XPath 2.0, variables (known as range variables) may be declared within an XPath expression, not only using `xsl:variable` elements in an XSLT stylesheet. The expressions that declare variables are the `for`, `some`, and `every` expressions. In XQuery, variables can also be declared in a `let` or `typeswitch` expression, as well as in the signature of a user-written function.

Function Calls

A function call in XPath 2.0 takes the form `F ( arg1, arg2, ...)` . In general, the function name is a QName. A library of core functions is defined in the XPath 2.0 and XSLT 2.0 specifications. For details of these functions, including notes on their implementation in this Saxon release, see the Functions section. Additional functions are available (in a special namespace) as Saxon extensions: these are listed in the Extensions. Further functions may be implemented by the user, either as XSLT (see xsl:function), as XQuery functions, or as Java (see the Extensibility section).

Axis steps

The basic primitive for accessing a source document is the . Axis steps may be combined into path expressions using the path operators `/` and `//`, and they may be filtered using filter expressions in the same way as the result of any other expression.

An axis step has the basic form `axis :: node-test`, and selects nodes on a given axis that satisfy the node-test. The axes available are:

Table 16.1.

ancestor	Selects ancestor nodes starting with the current node and ending with the document node
ancestor-or-self	Selects the current node plus all ancestor nodes
attribute	Selects all attributes of the current node (if it is an element)
child	Selects the children of the current node, in documetn order
descendant	Selects the children of the current node and their children, recursively (in document order)
descendant-or-self	Selects the current node plus all descendant nodes
following	Selects the nodes that follow the current node in document order, other than its descendants
following-sibling	Selects all subsequent child nodes of the same parent node
namespace	Selects all the in-scope namespaces for an element (this axis is deprecated in XPath 2.0, but Saxon continues to support it)
parent	Selects the parent of the current node
preceding	Selects the nodes that precede the current node in document order, other than its ancestors
preceding-sibling	Selects all preceding child nodes of the same parent node
self	Selects the current node

When the child axis is used, `child::` may be omitted, and when the attribute axis is used, `attribute::` may be abbviated to `@`. The expression `parent::node()` may be shortened to `..`

The node-test may be, for example:

- a node name
- `prefix:*` to select nodes in a given namespace
- `*:localname` to select nodes with a given local name, regardless of namespace
- `text()` (to select text nodes)
- `node()` (to select any node)
- `processing-instruction()` (to select any processing instruction)
- `processing-instruction('literal')` to select processing instructions with the given name (target)

- `comment()` to select comment nodes
- `element()` or `element(*)` to select any element node
- `element(N)` to select any element node named N
- `element(*, T)` to select any element node whose type annotation is T, or a subtype of T
- `element(N, T)` to select any element node whose name is N and whose type annotation is T, or a subtype of T
- `schema-element(N)` to select any element node whose name is N, or an element in the substitution group of N, that conforms to the schema-defined type for a global element declaration named N in an imported schema
- `attribute` or `attribute(*)` to select any attribute node
- `attrbute(N)` to select any attribute node named N
- `attribute(*, T)` to select any attribute node whose type annotation is T, or a subtype of T
- `attribute(N, T)` to select any attribute node whose name is N and whose type annotation is T, or a subtype of T
- `schema-attribute(N)` to select any attribute node whose name is N, that conforms to the schema-defined type for a global attribute declaration named N in an imported schema

Parentheses and operator precedence

In general an expression may be enclosed in parentheses without changing its meaning.

If parentheses are not used, operator precedence follows the sequence below, starting with the operators that bind most tightly. Within each group the operators are evaluated left-to-right

Table 16.2.

[]	predicate
/, //	path operator
unary -, unary +	unary plus and minus
cast as	dynamic type conversion
castable as	type test
treat as	static type conversion
instance of	type test
except, intersect	set difference and intersection
|, union	union operation on sets
*, div, idiv, mod	multiply, divide, integer divide, modulo
+, -	plus, minus
to	range expression
=, !=, is, <, <=;, >, >=, eq, ne, lt, le, gt, ge	comparisons
and	Boolean and
or	Boolean or
if	conditional expressions
some, every	quantified expressions

for	iteration (mapping) over a sequence
, (comma)	Sequence concatenation

The various operators are described, in roughly the above order, in the sections that follow.

Filter expressions

The notation `E[P]` is used to select items from the sequence obtained by evaluating `E`. If the predicate `P` is numeric, the predicate selects an item if its position (counting from 1) is equal to `P`; otherwise, the of `P` determines whether an item is selected or not. The effective boolean value of a sequence is false if the sequence is empty, or if it contains a single item that is one of: the boolean value false, the zero-length string, or a numeric zero or NaN value. If the first item of the sequence is a node, or if the sequence is a singleton boolean, number or string other than those listed above, the effective boolean value is true. In other cases (for example, if the sequence contains two booleans or a date), evaluating the effective boolean value causes an error.

In XPath 2.0, `E` may be any sequence, it is not restricted to a node sequence. Within the predicate, the expression `.` (dot) refers to the context item, that is, the item currently being tested. The XPath 1.0 concept of context node has thus been generalized, for example `.` can refer to a string or a number.

Generally the order of items in the result preserves the order of items in `E`. As a special case, however, if `E` is a step using a reverse axis (e.g. preceding-sibling), the position of nodes for the purpose of evaluating the predicate is in reverse document order, but the result of the filter expression is in forwards document order.

Path expressions

A path expression is a sequence of steps separated by the `/` or `//` operator. For example, `../@desc` selects the `desc` attribute of the parent of the context node.

In XPath 2.0, path expressions have been generalized so that any expression can be used as an operand of `/`, (both on the left and the right), so long as its value is a sequence of nodes. For example, it is possible to use a union expression (in parentheses) or a call to the `id()` or `key()` functions. The right-hand operand is evaluated once for each node in the sequence that results from evaluating the left-hand operand, with that node as the context item. In the result of the path expression, nodes are sorted in document order, and duplicates are eliminated.

In practice, it only makes sense to use expressions on the right of `/` if they depend on the context item. It is legal to write `$x/$y` provided both `$x` and `$y` are sequences of nodes, but the result is exactly the same as writing `./$y`.

Note that the expressions `./$X` or `$X/.` can be used to remove duplicates from `$X` and sort the results into document order. The same effect can be achieved by writing `$X|()`

The operator `//` is an abbreviation for `/descendant-or-self::node()/`. An expression of the form `/E` is shorthand for `root(.)/E`, and the expression `/` on its own is shorthand for `root(.)`.

The expression on the left of the `/` operator must return a node or sequence of nodes. The expression on the right can return either a sequence of nodes or a sequence of atomic values (but not a mixture of the two). This allow constructs such as `$x/number()`, which returns the sequence obtained by converting each item in $x to a number.

Cast as, Treat as

The expression `E cast as T` converts the value of expression `E` to type `T`. Whether T is a built-in type or a user-defined type, the effect is exactly the same as using the constructor function `T (E)`.

The expression `E treat as T` is designed for environments that perform static type checking. Saxon doesn't do static type checking, so this expression has very little use, except to document an assertion that the expression `E` is of a particular type. A run-time failure will be reported if the value of `E` is not of type `T`; no attempt is made to convert the value to this type.

Set difference and intersection

These operators are new in XPath 2.0.

The expression `E1 except E2` selects all nodes that are in `E1` unless they are also in `E2`. Both expressions must return sequences of nodes. The results are returned in document order. For example, `@* except @note` returns all attributes except the `note` attribute.

The expression `E1 intersect E2` selects all nodes that are in both `E1` and `E2`. Both expressions must return sequences of nodes. The results are returned in document order. For example, `preceding::fig intersect ancestor::chapter//fig` returns all preceding `fig` elements within the current chapter.

Union

The `|` operator was available in XPath 1.0; the keyword `union` has been added in XPath 2.0 as a synonym, because it is familiar to SQL users.

The expression `E1 union E2` selects all nodes that are in either `E1` or `E2` or both. Both expressions must return sequences of nodes. The results are returned in document order. For example, `/book/(chapter | appendix)/sections` returns all `section` elements within a `chapter` or `appendix` of the selected `book` element.

Arithmetic expressions

- Unary plus and minus
- Multiplication and division
- Addition and subtraction

Unary plus and minus

The unary minus operator changes the sign of a number. For example `-1` is minus one, and `-0e0` is the double value negative zero.

Unary plus has very little effect: the value of `+1` is the same as the value of `1`. It does, however, provide a quick way of forcing untyped values to be numeric, for example you can write `<xsl:sort select="+@price"/>` to force a numeric sort, if you find `<xsl:sort select="number(@price)"/>` too verbose for your tastes.

Multiplication and division

The operator `*` multiplies two numbers. If the operands are of different types, one of them is promoted to the type of the other (for example, an integer is promoted to a decimal, a decimal to a double). The result is the same type as the operands after promotion.

The operator `div` divides two numbers. Dividing two integers produces a double; in other cases the result is the same type as the operands, after promotion. In the case of decimal division, the precision is the sum of the precisions of the two operands, plus six.

The operator `idiv` performs integer division. For example, the result of `10 idiv 3` is `3`.

The `mod` operator returns the modulus (or remainder) after division. See the XPath 2.0 specification for details of the way that negative numbers are handled.

The operators `*` and `div` may also be used to multiply or divide a duration by a number. For example, `fn:dayTimeDuration('PT12H') * 4` returns the duration two days.

Addition and subtraction

The operators `+` and `-` perform addition and subtraction of numbers, in the usual way. If the operands are of different types, one of them is promoted, and the result is the same type as the operands after promotion. For example, adding two integers produces an integer; adding an integer to a double produces a double.

Note that the `-` operator may need to be preceded by a space to prevent it being parsed as part of the preceding name.

XPath 2.0 also allows these operators to be used for adding durations to durations or to dates and times.

Range expressions

The expression `E1 to E2` returns a sequence of integers. For example, `1 to 5` returns the sequence `1, 2, 3, 4, 5`. This is useful in `for` expressions, for example the first five nodes of a node sequence can be processed by writing `for $i in 1 to 5 return (//x)[$i]`.

If you prefer, you can write this as `(//x)[position() = 1 to 5]`. This works because comparison of a single integer (`position()`) to a sequence of integers (`1 to 5`) is true if the integer on the left is equal to any integer in the sequence.

Comparisons

The simplest comparison operators are `eq`, `ne`, `lt` `le`, `gt`, `ge`. These compare two atomic values of the same type, for example two integers, two dates, or two strings. In the case of strings, the default collation is used (see saxon:collation). If the operands are not atomic values, an error is raised.

The operators =, `!=`, <, <=, >, and >= can compare arbitrary sequences. The result is true if any pair of items from the two sequences has the specified relationship, for example `$A = $B` is true if there is an item in `$A` that is equal to some item in `$B`. If an argument is a node, the effect depends on whether the source document has been validated against a schema. In Saxon-EE, with a validated source document, Saxon will use the typed value of the node in the comparison. Without schema validation, the type of the node is `untypedAtomic`, and the effect is that the value is converted to the type of the other operand.

The operator `is` tests whether the operands represent the same (identical) node. For example, `title[1] is *[@note][1]` is true if the first `title` child is the first child element that has a `@note` attribute. If either operand is an empty sequence the result is an empty sequence (which will usually be treated as false).

The operators << and >> test whether one node precedes or follows another in document order.

Instance of and Castable as

The expression `E instance of T` tests whether the value of expression `E` is an instance of type T, or of a subtype of T. For example, `$p instance of attribute+` is true if the value of `$p` is a sequence of one or more attribute nodes. It returns false if the sequence is empty or if it contains an item that is not an attribute node. The detailed rules for defining types, and for matching values against a type, are given in the XPath 2.0 specification.

Saxon also allows testing of the type annotation of an element or attribute node using tests of the form `element(*, T)`, `attribute(*, T)`. This is primarily useful with a schema-aware query or stylesheet, since the only way a node can acquire a type annotation (other than the special values `xs:untyped` and `xs:untypedAtomic`) is by validating a document against a schema.

The expression `E castable as T` tests whether the expression `E cast as T` would succeed. It is useful, for example, for testing whether a string contains a valid date before attempting to cast it to a date. This is because XPath and XSLT currently provide no way of trapping the error if the cast is attempted and fails.

Conditional Expressions

XPath 2.0 allows a conditional expression of the form `if ( E1 ) then E2 else E3`. For example, `if (@discount) then @discount else 0` returns the value of the `discount` attribute if it is present, or zero otherwise.

Quantified Expressions

The expression `some $x in E1 satisfies E2` returns true if there is an item in the sequence `E1` for which the of `E2` is true. Note that `E2` must use the range variable `$x` to refer to the item being tested; it does not become the context item. For example, `some $x in @* satisfies $x eq ""` is true if the context item is an element that has at least one zero-length attribute value.

Similarly, the expression `every $x in E1 satisfies E2` returns true if every item in the sequence given by `E1` satisfies the condition.

For Expressions

The expression `for $x in E1 return E2` returns the sequence that results from evaluating `E2` once for every item in the sequence `E1`. Note that `E2` must use the range variable `$x` to refer to the item being tested; it does not become the context item. For example, `sum(for $v in order-item return $v/price * $v/quantity)` returns the total value of (price times quantity) for all the selected `order-item` elements.

Boolean expressions: AND and OR

The expression `E1 and E2` returns true if the of `E1` and `E2` are both true.

The expression `E1 or E2` returns true if the of either or both of `E1` and `E2` are true.

Sequence expressions

The expression `E1 , E2` returns the sequence obtained by concatenating the sequences `E1` and `E2`.

For example, `$x = ("London", "Paris", "Tokyo")` returns true if the value of `$x` is one of the strings listed.

New features in XPath 3.0

These features are available in Saxon only if explicitly enabled, either directly, or by requesting support for XSLT 3.0 or XQuery 3.0. In both cases, this requires Saxon-PE or above.

Some of the new features in XPath 3.0 are as follows. For full details, see the W3C specifications.

1. The concatenation operator `||` is available (as in SQL). For example, `('$' || 12.5)` returns the string '$12.5'.

2. A new simple mapping operator is available, `!`. This works rather like `/`, except there is no restriction that the left hand operand must be a node-set, and there is no sorting of results into document order. For example, `(1 to 7)!(.*.)` returns the sequence `(1, 4, 9, 16, 25, 36, 49)`.

3. Local variables can be declared in a `let` expression, for example `let $x := /*/@version return //e[@version = $x]`

4. Inline function items can be declared, and used as arguments to higher-order functions. For example `map(//employee, function($e){$e/salary + $e/bonus})`. A is a third kind of item, alongside nodes and atomic values. The function represented by a function item `$f` can be invoked using a dynamic function call `$f(args)`.

5. Maps are available (for more details see Maps. They provide a similar capability to "objects" in Javascript, or "associative arrays" in some other languages. But as befits a function language like XPath, they are immutable. A collection of functions is available to operate on maps (see XSLT 2.0 and XPath 2.0 Functions), and in addition there is new syntax for a map constructor (of the form `map{ key := value, key := value }` where both the keys and values are arbitrary expressions. There is a sequenceType for maps: `map(K, V)` defining the types of the key and value parts. Maps are functions, so given a map $M, the entry for a key $K can be obtained as the result of the function call `$M($K)`.

6. Expanded QNames can be written in the notation `"uri":local`, allowing XPath expressions to be written that do not depend on an externally-supplied namespace context.

A number of new functions are available, including `head`, `tail`, `map`, `filter`, `map-pairs`, `pi`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sqrt`, `format-integer`, and others. For details see XSLT 2.0 and XPath 2.0 Functions.

Maps in XPath 3.0

The XSL Working Group has proposed extensions to XPath to handle maps. These extensions have not yet been accepted into the XPath 3.0 working drafts, but they are implemented in Saxon 9.4.

A map is a new kind of XDM item (alongside nodes and atomic values). In fact, a map is a kind of function: you can think of it as a function defined extensionally (by tabulating the value of the function for all possible arguments) rather than intensionally (by means of an algorithm).

A map is a set of entries. Each entry is a key-value pair. The key is always an atomic value. The "value" is any XDM value: a sequence of nodes, atomic values, functions, or maps.

Maps, like sequences, are immutable. When you add an entry to a map, you get a new map; the original is unchanged. Saxon provides an efficient implementation of maps that achieves this without copying the whole map every time an entry is added.

Also like sequences, maps do not have an intrinsic type of their own, but rather have a type that can be inferred from what they contain. A map conforms to the type `map(K, V)` if all the keys are of type K and all the values are of type V. For example if the keys are all strings, and the values are all employee elements, then the map conforms to the type `map(xs:string, element(employee))`.

There are several ways to create a map:

- If the number of entries is known, you can use the constructor syntax `map { key := value; key := value; ... }`. Here the keys and values can be any "simple expression" (an expression not containing a top-level comma). If the keys and values are all known statically, you might write: `map { "a" := 1; "e: := 2; "i" := 3; "o" := 4; "u" := 5 }`. You can use this construct anywhere an XPath expression can be used, for example in the `select` attribute of an `xsl:variable` element.

- The function `map:new()` takes a number of maps as input, and combines them into a single map. This can be used to construct a map where the number of entries is not known statically: for example `for $i in 1 to 10 return map{$i := codepoints-to-string($i)}`.

- A single-entry map can also be constructed using the function `map:entry($key, $value)`

Given a map `$M`, the value corresponding to a given key `$K` can be found either by invoking the map as a function: `$M($K)`, or by calling `map:get($M, $K)`.

The full list of functions that operate on maps is as follows. The prefix `map` represents the namespace URI `http://www.w3.org/2005/xpath-functions/map`

- map:new($maps as map(*)) as map(*): takes a sequence of maps as input and combines them into a single map.

- map:new($maps as map(*), $collation as xs:string) as map(*): takes a sequence of maps as input and combines them into a single map, using the specified collation to compare key values.

- map:collation($map as map(*)) as xs:string: returns the collation of a map.

- map:keys($map as map(*)) as xs:anyAtomicType*: returns the keys that are present in a map, in unpredictable order.

- map:contains($map as map(*), $key as xs:anyAtomicType) as xs:boolean: returns true if the given key is present in the map.

- map:get($map as map(*), $key as xs:anyAtomicType) as xs:item()*: returns the value associated with the given key if present, or the empty sequence otherwise. Equivalent to calling `$map($key)`.

- map:entry($key as xs:anyAtomicType, $value as item()*): creates a singleton map. Useful as an argument to `map:new()`

- map:remove($map as map(*), $key as xs:anyAtomicType) as map(*): removes an entry from a map (if it was present), returning a new map; if not present, returns the existing map unchanged.

Chapter 17. XSLT 2.0 and XPath 2.0 Functions

Index of Functions

The information in this section indicates which functions are implemented in this Saxon release, and any restrictions in the current implementation.

It includes both the core functions defined in XPath, and the additional functions defined in the XSLT specification.

Table 17.1.

abs acos adjust-dateTime-to-timezone adjust-date-to-timezone adjust-time-to-timezone analyze-string asin atan available-environment-variables avg base-uri boolean ceiling codepoint-equal codepoints-to-string collection compare concat contains cos count current current-date current-dateTime current-group current-grouping-key current-time data dateTime day-from-date day-from-dateTime days-from-duration deep-equal default-collation distinct-values doc doc-available document document-uri element-available element-with-id empty encode-for-uri ends-with environment-variable error escape-html-uri exactly-one exists exp exp10 false filter floor fold-left fold-right format-date format-dateTime format-integer format-number format-time function-arity function-available function-lookup function-name generate-id has-children head hours-from-dateTime hours-from-duration hours-from-time id idref implicit-timezone index-of innermost in-scope-prefixes insert-before iri-to-uri lang last local-name local-name-from-QName log	log10 lower-case map map-pairs matches max min minutes-from-dateTime minutes-from-duration minutes-from-time month-from-date month-from-dateTime months-from-duration name namespace-uri namespace-uri-for-prefix namespace-uri-from-QName nilled node-name normalize-space normalize-unicode not number one-or-more outermost parse-json parse-xml path pi position pow prefix-from-QName put QName regex-group remove replace resolve-QName resolve-uri reverse root round round-half-to-even seconds-from-dateTime seconds-from-duration seconds-from-time serialize serialize-json sin sqrt starts-with static-base-uri string string-join string-length string-to-codepoints subsequence substring substring-after substring-before sum system-property tail tan timezone-from-date timezone-from-dateTime timezone-from-time tokenize trace translate true type-available unordered unparsed-entity-public-id unparsed-entity-uri unparsed-text unparsed-text-available unparsed-text-lines upper-case uri-collection year-from-date year-from-dateTime years-from-duration zero-or-one

abs

Returns the absolute value of a given number. Returns the same type as the supplied argument.

abs($arg as numeric?) # numeric?

Table 17.2.

	$arg	numeric?	The input value
	numeric?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-abs]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-abs]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

acos

Returns the arc cosine of the argument, the result being in the range zero to +# radians.

acos($arg as xs:double?) # xs:double?

Table 17.3.

	$arg	xs:double?	The supplied angle in radians
	xs:double?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions/math

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-acos]

Notes on the Saxon implementation

Implemented since Saxon 9.3; available whether or not support for XPath 3.0 is enabled

adjust-dateTime-to-timezone

adjust-dateTime-to-timezone($arg as xs:dateTime?) # xs:dateTime

Returns a dateTime value equivalent to the original dateTime, but with the timezone removed.

Table 17.4.

	$arg	xs:dateTime?	The input date/time
	xs:dateTime		

adjust-dateTime-to-timezone($arg as xs:dateTime?, $timezone as xs:dayTimeDuration?) # xs:dateTime

Returns a dateTime value equivalent to the original dateTime, but adjusted to a different timezone.

Table 17.5.

	$arg	xs:dateTime?	The input date/time
	$timezone	xs:dayTimeDuration?	The new time zone, as an offset from UTC, or () to denote the implicit timezone
	xs:dateTime		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-adjust-dateTime-to-timezone]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-adjust-dateTime-to-timezone]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

adjust-date-to-timezone

adjust-date-to-timezone($arg as xs:date?) # xs:date?

Returns a date value equivalent to the original date, but with the timezone removed.

Table 17.6.

	$arg	xs:date?	The input date
	xs:date?		

adjust-date-to-timezone($arg as xs:date?, $timezone as xs:dayTimeDuration?) # xs:date?

Returns a date value equivalent to the original date, but adjusted to a different timezone.

Table 17.7.

	$arg	xs:date?	The input date
	$timezone	xs:dayTimeDuration?	The new time zone, as an offset from UTC, or () to denote the implicit timezone
	xs:date?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-adjust-date-to-timezone]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-adjust-date-to-timezone]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

adjust-time-to-timezone

adjust-time-to-timezone($arg as xs:time?) # xs:time?

Returns a time value equivalent to the original time, but with the timezone removed.

Table 17.8.

	$arg	xs:time?	The input time
	xs:time?		

adjust-time-to-timezone($arg as xs:time?, $timezone as xs:dayTimeDuration?) # xs:time?

Returns a time value equivalent to the original time, but adjusted to a different timezone.

Table 17.9.

	$arg	xs:time?	The input time
	$timezone	xs:dayTimeDuration?	The new time zone, as an offset from UTC, or () to denote the implicit timezone
	xs:time?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-adjust-time-to-timezone]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-adjust-time-to-timezone]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

analyze-string

Analyzes a string using a regular expression, returning an XML structure that identifies which parts of the input string matched or failed to match the regular expression, and in the case of matched substrings, which substrings matched each capturing group in the regular expression.

analyze-string($input as xs:string?, $pattern as xs:string) # element(fn:analyze-string-result)

Table 17.10.

	$input	xs:string?	The input string
	$pattern	xs:string	A regular expression
	element(fn:analyze-string-result)		

analyze-string($input as xs:string?, $pattern as xs:string, $flags as xs:string) # element(fn:analyze-string-result)

Table 17.11.

	$input	xs:string?	The input string
	$pattern	xs:string	A regular expression
	$flags	xs:string	Flags controlling the interpretation of the regular expression
	element(fn:analyze-string-result)		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-analyze-string]

Notes on the Saxon implementation

New in Saxon 9.3

asin

Returns the arc sine of the argument, the result being in the range -#/2 to +#/2 radians.

asin($arg as xs:double?) # xs:double?

Table 17.12.

	$arg	xs:double?	The supplied angle in radians
	xs:double?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions/math

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-asin]

Notes on the Saxon implementation

Implemented since Saxon 9.3; available whether or not support for XPath 3.0 is enabled

atan

Returns the arc tangent of the argument, the result being in the range -#/2 to +#/2 radians.

atan($arg as xs:double?) # xs:double?

Table 17.13.

	$arg	xs:double?	The supplied angle in radians
	xs:double?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions/math

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-atan]

Notes on the Saxon implementation

Implemented in Saxon 9.3; available whether or not support for XPath 3.0 is enabled

available-environment-variables

Returns a list of environment variable names that are suitable for passing to `environment-variable()`, as a (possibly empty) sequence of strings.

available-environment-variables() # xs:string*

Table 17.14.

	xs:string*

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-available-environment-variables]

Notes on the Saxon implementation

Implemented in Saxon 9.4, but without enforcing the rule that the result must be deterministic.

avg

Returns the average of a set of numbers or durations

avg($arg as xs:anyAtomicType*) # xs:anyAtomicType?

Table 17.15.

	$arg	xs:anyAtomicType*	The input sequence of numbers or durations
	xs:anyAtomicType?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-avg]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-avg]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

base-uri

base-uri() # xs:anyURI?

Returns the base URI of the context node

Table 17.16.

	xs:anyURI?

base-uri($arg as node()?) # xs:anyURI?

Returns the base URI of a specified node

Table 17.17.

	$arg	node()?	The node whose base URI is required
	xs:anyURI?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-base-uri]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-base-uri]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

boolean

Obtains the effective boolean value of the supplied argument. The result is true if the argument value is a sequence starting with a node, or the singleton boolean true(), a singleton non-zero number, or a singleton non-zero-length string (or untypedAtomic). The result is false if the argument is an empty sequence, the singleton boolean false, a singleton number zero or NaN, or a singleton zero-length string (or untypedAtomic). In all other cases the result is an error.

boolean($arg as item()*) # xs:boolean

Table 17.18.

	$arg	item()*	The sequence whose effective boolean value is required
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-boolean]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-boolean]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

ceiling

Rounds a value towards positive infinity

ceiling($arg as numeric?) # numeric?

Table 17.19.

	$arg	numeric?	The supplied number
	numeric?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-ceiling]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-ceiling]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

codepoint-equal

Compares two strings using the Unicode codepoint collation

codepoint-equal($comparand1 as xs:string?, $comparand2 as xs:string?) # xs:boolean?

Table 17.20.

	$comparand1	xs:string?	The first value to be compared
	$comparand2	xs:string?	The second value to be compared
	xs:boolean?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-codepoint-equal]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-codepoint-equal]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

codepoints-to-string

Converts a sequence of integers representing Unicode characters to the corresponding string.

codepoints-to-string($arg as xs:integer*) # xs:string

Table 17.21.

	$arg	xs:integer*	A sequence of Unicode codepoint values
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-codepoints-to-string]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-codepoints-to-string]

Notes on the Saxon implementation

The function throws an error if any of the integers doesn't represent a valid XML character. With Saxon, the set of valid XML characters depends on whether XML 1.0 or 1.1 is the chosen version in the `Configuration`.

collection

collection() # node()*

Returns the nodes making up the default collection

Table 17.22.

	node()*

collection($arg as xs:string?) # node()*

Returns the nodes making up the collection whose URI is supplied

Table 17.23.

	$arg	xs:string?	The supplied collection URI
	node()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-collection]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-collection]

Notes on the Saxon implementation

Saxon implements the zero-argument function by passing a null URI to the registered `CollectionURIResolver`. The default `CollectionURIResolver` implementation supplied with the product handles this by returning an empty sequence, but a user-supplied `CollectionURIResolver` is free to handle this case any way it wishes.

If a user-defined `CollectionURIResolver` has been registered, the action of this function is entirely user-defined. A resolver may be registered using the `setCollectionResolver()` method on the `Configuration` object, or (in XSLT) using `setAttribute()` on the `TransformerFactory`. The `CollectionURIResolver` may also be nominated using the -cr option on the command line.

For details of the behavior of the standard `CollectionURIResolver`, see Collections.

compare

compare($comparand1 as xs:string?, $comparand2 as xs:string?) # xs:integer?

Compares two strings using the default collation

Table 17.24.

	$comparand1	xs:string?	The first string to be compared
	$comparand2	xs:string?	The second string to be compared
	xs:integer?		

compare($comparand1 as xs:string?, $comparand2 as xs:string?, $collation as xs:string) # xs:integer?

Compares two strings using the specified collation

Table 17.25.

	$comparand1	xs:string?	The first string to be compared
	$comparand2	xs:string?	The second string to be compared
	$collation	xs:string	The collation to be used for the comparison
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-compare]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-compare]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

concat

Concatenates the string-values of the arguments into a single string. There must be at least two arguments.

concat($arg1 as xs:anyAtomicType?, $arg2 as xs:anyAtomicType?, $etc... as xs:anyAtomicType?) # xs:string

Table 17.26.

	$arg1	xs:anyAtomicType?	The first string
	$arg2	xs:anyAtomicType?	The second string
	$etc...	xs:anyAtomicType?	The third and subsequent strings (as many as required)

	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-concat]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-concat]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

contains

Returns true if the second string is a substring of the first

contains($arg1 as xs:string?, $arg2 as xs:string?) # xs:boolean

Table 17.27.

	$arg1	xs:string?	The containing string
	$arg2	xs:string?	The contained string
	xs:boolean		

contains($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) # xs:boolean

Table 17.28.

	$arg1	xs:string?	The containing string
	$arg2	xs:string?	The contained string
	$collation	xs:string	The collation to be used for comparing the strings
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-contains]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-contains]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

cos

Returns the cosine of the argument, expressed in radians.

cos($# as xs:double?) # xs:double?

Table 17.29.

	$#	xs:double?	The supplied angle, in radians
	xs:double?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions/math

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-cos]

Notes on the Saxon implementation

Implemented since Saxon 9.3; available whether or not support for XPath 3.0 is enabled

count

Counts the number of items in a sequence

count($arg as item()*) # xs:integer

Table 17.30.

	$arg	item()*	The sequence whose items are to be counted
	xs:integer		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-count]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-count]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

current

Returns the item that was the current item supplied on entry to the XPath expression

current() # item()

Table 17.31.

	item()

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 2.0 and later versions

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-current]

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-current]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

current-date

Returns the current date

current-date() # xs:date

Table 17.32.

	xs:date

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-current-date]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-current-date]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

current-dateTime

Returns the current date and time. Note that this does not change during the execution of the query or transformation.

current-dateTime() # xs:dateTimeStamp

Table 17.33.

	xs:dateTimeStamp

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-current-dateTime]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-current-dateTime]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

current-group

Returns the contents of the current group selected by `xsl:for-each-group`

current-group() # item()

Table 17.34.

	item()

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 2.0 and later versions

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-current-group]

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-current-group]

Notes on the Saxon implementation

Saxon 9.3 and earlier releases implement the XSLT 2.0 definition of this function. The additional behaviour associated with the `xsl:merge` instruction is implemented in Saxon 9.4.

current-grouping-key

Returns the value that is the grouping key of the current group selected by `xsl:for-each-group`

current-grouping-key() # xs:anyAtomicType

Table 17.35.

	xs:anyAtomicType

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 2.0 and later versions

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-current-grouping-key]

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-current-grouping-key]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

current-time

Returns the current time. Note that this does not change during the execution of the query or transformation.

current-time() # xs:time

Table 17.36.

	xs:time

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-current-time]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-current-time]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

data

Returns the result of atomizing the supplied sequence

data() # xs:anyAtomicType*

Returns the result of atomizing the context item. New in XPath 3.0

Table 17.37.

	xs:anyAtomicType*

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Specification [http://www.w3.org/TR/xpath-functions-11/#func-data]

data($arg as item()*) # xs:anyAtomicType*

Table 17.38.

	$arg	item()*	The value to be atomized
	xs:anyAtomicType*		

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Specification [http://www.w3.org/TR/xpath-functions/#func-data]

XPath 3.0 Specification [http://www.w3.org/TR/xpath-functions-11/#func-data]

Notes on the Saxon implementation

The zero-argument version of this function is newly implemented in Saxon 9.3, and available only if XPath 3.0 support is enabled.

dateTime

Combines the given date and time. The result has a timezone if either of the inputs has a timezone; if they both have a timezone, then the two timezones must be the same.

dateTime($arg1 as xs:date?, $arg2 as xs:time?) # xs:dateTime?

Table 17.39.

	$arg1	xs:date?	The supplied date
	$arg2	xs:time?	The supplied time
	xs:dateTime?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-dateTime]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-dateTime]

Notes on the Saxon implementation

If support for XSD 1.1 is enabled, the result will be an instance of `xs:dateTimeStamp`.

day-from-date

Extracts the day component of a date value

day-from-date($arg as xs:date?) # xs:integer?

Table 17.40.

	$arg	xs:date?	The supplied date
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-day-from-date]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-day-from-date]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

day-from-dateTime

Extracts the day component of a dateTime value

day-from-dateTime($arg as xs:dateTime?) # xs:integer?

Table 17.41.

	$arg	xs:dateTime?	The supplied dateTime
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-day-from-dateTime]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-day-from-dateTime]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

days-from-duration

Extracts the days component of a dayTimeDuration value

days-from-duration($arg as xs:duration?) # xs:integer?

Table 17.42.

	$arg	xs:duration?	The supplied dateTime
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-days-from-duration]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-days-from-duration]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

deep-equal

Compares two sequences for deep equality; string values are compared using the specified collation; nodes are compared for deep equality of names and content.

deep-equal($parameter1 as item()*, $parameter2 as item()*) # xs:boolean

Table 17.43.

	$parameter1	item()*	The first value to be compared
	$parameter2	item()*	The second value to be compared
	xs:boolean		

deep-equal($parameter1 as item()*, $parameter2 as item()*, $collation as xs:string) # xs:boolean

Table 17.44.

	$parameter1	item()*	The first value to be compared
	$parameter2	item()*	The second value to be compared
	$collation	xs:string	The collation to be used whenever strings are compared
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-deep-equal]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-deep-equal]

Notes on the Saxon implementation

A Saxon-specific variant of this function is available, with additional comparison options: see saxon:deep-equal

default-collation

Returns the name of the default collation. If no collation has been set explicitly this will be the URI of the Unicode codepoint collation.

default-collation() # xs:string

Table 17.45.

	xs:string

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-default-collation]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-default-collation]

Notes on the Saxon implementation

Various Saxon APIs allow the default collation to be set for a transformation or query. It is also possible to set this from the command line, by using of the option `--defaultCollation`.

distinct-values

Returns the set of distinct values present in a given sequence.

distinct-values($arg as xs:anyAtomicType*) # xs:anyAtomicType*

Table 17.46.

	$arg	xs:anyAtomicType*	The sequence to be de-duplicated
	xs:anyAtomicType*		

distinct-values($arg as xs:anyAtomicType*, $collation as xs:string) # xs:anyAtomicType*

Table 17.47.

	$arg	xs:anyAtomicType*	The sequence to be de-duplicated
	$collation	xs:string	The collation used for comparing strings
	xs:anyAtomicType*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-distinct-values]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-distinct-values]

Notes on the Saxon implementation

The current Saxon implementation returns the values in "order of first appearance", but it cannot be assumed that this will always remain the case.

doc

Retrieves an XML document located at the specified URI, parses it, and returns its document node

doc($uri as xs:string?) # document-node()?

Table 17.48.

	$uri	xs:string?	The URI of the required document (relative to the static base URI)
	document-node()?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-doc]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-doc]

Notes on the Saxon implementation

By default the URI is dereferenced using the conventional URL behaviour, as implement by the Java runtime library; this behaviour can be modified by means of a user-supplied `URIResolver`.

If the same URI is requested repeatedly, Saxon retains the document node in memory and returns the same instance each time.

The way the URI is handled depends on the `URIResolver` in use. The standard URI resolver has an option (set using -p on the command line, or via options on the `Configuration` or `TransformerFactory` classes) to recognize query parameters in the URI. These are keyword=value pairs, separated by semicolons or ampersand characters, giving options for parsing the file located via the URI. The options that are then recognized are:

- `validation=strict|lax|preserve|strip`: determines how the input document will be validated. The options "strict" and "lax" require Saxon-EE.
- `strip-space=yes|no|ignorable`: determines whether whitespace-only text nodes will be stripped from the source document. (Such nodes are stripped if this is requested either using this option, or using xsl:strip-space declarations in the stylesheet.) The value "ignorable" causes whitespace text nodes to be stripped if they belong to an element defined in a DTD or schema as having element-only content.
- `parser=full.class.name`: determines the name of the parser (XMLReader) to be used to parse this input file. For example, `parser=org.ccil.cowan.tagsoup.Parser` causes John Cowan's TagSoup parser for HTML to be used.

doc-available

Returns true if a document with the given URI can be successfully loaded, false otherwise.

doc-available($uri as xs:string?) # xs:boolean

Table 17.49.

	$uri	xs:string?	The URI of the required document (relative to the static base URI)
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-doc-available]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-doc-available]

Notes on the Saxon implementation

Saxon effectively executes the `doc()` function and returns true if it succeeded; as a side-effect, the document will be available in memory for use when the `doc()` function is subsequently called with this URI.

A user-supplied URIResolver is invoked in the same way as for the `doc()` function.

document

Single argument function: Loads one or more documents identified by their URIs. URIs are handled by the `URIResolver` in the same way as the `doc()` function.

Two argument functionLoads one or more documents identified by their URIs, using the base URI of the node given in the second argument to resolve any relative URIs. URIs are handled by the `URIResolver` in the same way as the `doc()` function.

document($uri as item()*) # node()*

Table 17.50.

	$uri	item()*	
	node()*		

document($uri as item()*, $base as node()*) # node()*

Table 17.51.

	$uri	item()*	
	$base	node()*	
	node()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 2.0 and later versions

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-document]

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-document]

Notes on the Saxon implementation

The Saxon implementation calls the same underlying code as the `doc` function; the differences are in the way relative URIs are handled, and the fact that a single call can process multiple URIs.

document-uri

Returns the URI of a document

document-uri() # xs:anyURI?

Table 17.52.

	xs:anyURI?

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Specification [http://www.w3.org/TR/xpath-functions-11/#func-document-uri]

document-uri($arg as node()?) # xs:anyURI?

Table 17.53.

	$arg	node()?	The node whose document URI is required
	xs:anyURI?		

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Specification [http://www.w3.org/TR/xpath-functions/#func-document-uri]

XPath 3.0 Specification [http://www.w3.org/TR/xpath-functions-11/#func-document-uri]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

element-available

Determines whether a particular instruction (typically, an extension element) is available for use in a stylesheet

element-available($arg as xs:string) # xs:boolean

Table 17.54.

	$arg	xs:string	The name of the extension element (a lexical QName)
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 2.0 and later versions

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-element-available]

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-element-available]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

element-with-id

Returns the sequence of element nodes that have an `ID` value matching the value of one or more of the `IDREF` values supplied in `$arg`.

element-with-id($arg as xs:string*) # element()*

Table 17.55.

	$arg	xs:string*	The ID value being sought, within the document containing the context node
	element()*		

element-with-id($arg as xs:string*, $node as node()) # element()*

Table 17.56.

	$arg	xs:string*	The ID value being sought
	$node	node()	A node (typically the root) of the document being searched
	element()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-element-with-id]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

empty

Returns true if the given sequence is empty

empty($arg as item()*) # xs:boolean

Table 17.57.

	$arg	item()*	The supplied sequence
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-empty]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-empty]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

encode-for-uri

Applies the %HH escaping convention to a URI, escaping both disallowed characters and reserved characters such as "/" and ":"

encode-for-uri($uri-part as xs:string?) # xs:string

Table 17.58.

	$uri-part	xs:string?	The string to be encoded for inclusion in a URI
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-encode-for-uri]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-encode-for-uri]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

ends-with

Returns true if the first string ends with the second string

ends-with($arg1 as xs:string?, $arg2 as xs:string?) # xs:boolean

Table 17.59.

	$arg1	xs:string?	The containing string
	$arg2	xs:string?	The supposed ending of the string
	xs:boolean		

ends-with($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) # xs:boolean

Table 17.60.

	$arg1	xs:string?	The containing string
	$arg2	xs:string?	The supposed ending of the string
	$collation	xs:string	The collation to be used for comparing characters
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-ends-with]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-ends-with]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

environment-variable

Returns the value of a system environment variable, if it exists.

Implemented in Saxon using the Java method `System.getenv()`. The rule in the XPath specification requiring the result to be deterministic is not enforced.

environment-variable($name as xs:string) # xs:string?

Table 17.61.

	$name	xs:string	The name of the required environment variable
	xs:string?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-environment-variable]

Notes on the Saxon implementation

Implemented in Saxon 9.3 under the name `get-environment-variable`; name changed to `environment-variable` in Saxon 9.4. Requires XPath 3.0 to be enabled.

error

Raises an error.

error() # none

Table 17.62.

	none

error($code as xs:QName) # none

Table 17.63.

	$code	xs:QName	The error code to be returned
	none		

error($code as xs:QName?, $description as xs:string) # none

Table 17.64.

	$code	xs:QName?	The error code to be returned
	$description	xs:string	An error message
	none		

error($code as xs:QName?, $description as xs:string, $error-object as item()*) # none

Table 17.65.

	$code	xs:QName?	The error code to be returned
	$description	xs:string	An error message
	$error-object	item()*	An object or value associated with the error
	none		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-error]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-error]

Notes on the Saxon implementation

Saxon allows the first argument to be an empty sequence, treating this as equivalent to calling fn:error() with no arguments. The error code (local part and namespace URI part) is recorded as part of the exception that's supplied to a user-defined JAXP `ErrorListener`, or that is returned to the calling application. The value of the `$description` is available by calling `getMessage()` on the `Exception` object. The error details are also available to the `catch` part of a call when using the XQuery 3.0 or XSLT 3.0 try/catch construct.

escape-html-uri

Applies the %HH escaping convention to a URI, according to the rules of the HTML specification: that is, non-ASCII characters are escaped, but all ASCII characters, including spaces, are retained intact.

escape-html-uri($uri as xs:string?) # xs:string

Table 17.66.

	$uri	xs:string?	The uri to be escaped
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-escape-html-uri]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-escape-html-uri]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

exactly-one

Checks whether `$arg` contains exactly one item; fails if it is empty or contains multiple items.

exactly-one($arg as item()*) # item()

Table 17.67.

	$arg	item()*	The sequence to be tested
	item()		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-exactly-one]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-exactly-one]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

exists

Returns true if the given sequence is not empty

exists($arg as item()*) # xs:boolean

Table 17.68.

	$arg	item()*	The input sequence
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-exists]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-exists]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

exp

Returns `e` to the power of `$arg`.

exp($arg as xs:double) # xs:double

Table 17.69.

	$arg	xs:double	The input number
	xs:double		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions/math

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-exp]

Notes on the Saxon implementation

Newly implemented in Saxon 9.4; available whether or not XPath 3.0 is enabled.

exp10

Returns 10 to the power of `$arg`.

exp10($arg as xs:double) # xs:double

Table 17.70.

	$arg	xs:double	The input number
	xs:double		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions/math

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-exp10]

Notes on the Saxon implementation

Newly implemented in Saxon 9.4; available whether or not XPath 3.0 is enabled.

false

Returns the boolean value false

false() # xs:boolean

Table 17.71.

	xs:boolean

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-false]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-false]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

filter

Returns those items from the sequence `$seq` for which the supplied function `$f` returns true.

filter($f as function(item()) as xs:boolean, $seq as item()*) # item()*

Table 17.72.

	$f	function(item()) as xs:boolean	The filtering function (used to test each item in the sequence
	$seq	item()*	The sequence to be filtered
	item()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-filter]

Notes on the Saxon implementation

Newly implemented in Saxon 9.3. Requires XPath 3.0 to be enabled.

floor

Rounds a number towards minus infinity

floor($arg as numeric?) # numeric?

Table 17.73.

	$arg	numeric?	The input number
	numeric?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-floor]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-floor]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

fold-left

Processes the supplied sequence from left to right, applying the supplied function repeatedly to each item in turn, together with an accumulated result value.

fold-left($f as function(item()*, item()) as item()*, $zero as item()*, $seq as item()*) # item()*

Table 17.74.

	$f	function(item()*, item()) as item()*	The function to be applied to each item in the sequence
	$zero	item()*	The initial value (the value to be returned if the sequence is empty)
	$seq	item()*	The input sequence

	item()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-fold-left]

Notes on the Saxon implementation

Newly implemented in Saxon 9.3. Requires XPath 3.0 to be enabled.

fold-right

Processes the supplied sequence from right to left, applying the supplied function repeatedly to each item in turn, together with an accumulated result value.

fold-right($f as function(item(), item()*) as item()*, $zero as item()*, $seq as item()*) # item()*

Table 17.75.

	$f	function(item(), item()*) as item()*	The function to be applied to each item in the sequence
	$zero	item()*	The initial value (the value to be returned if the sequence is empty)
	$seq	item()*	The input sequence
	item()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-fold-right]

Notes on the Saxon implementation

Newly implemented in Saxon 9.3. Requires XPath 3.0 to be enabled.

format-date

format-date($value as xs:date?, $picture as xs:string) # xs:string?

Formats a date, using a format controlled by the picture string. The result is equivalent to the 5-argument form of the function, with the third, fourth, and fifth arguments set to an empty sequence.

Table 17.76.

	$value	xs:date?	The date to be formatted
	$picture	xs:string	Picture showing how the date is to be formatted
	xs:string?		

format-date($value as xs:date?, $picture as xs:string, $language as xs:string?, $calendar as xs:string?, $place as xs:string?) # xs:string?

Formats a date, using a format controlled by the picture string.

Table 17.77.

	$value	xs:date?	The date to be formatted
	$picture	xs:string	Picture showing how the date is to be formatted
	$language	xs:string?	The language for the output
	$calendar	xs:string?	The calendar for the output
	$place	xs:string?	The country or Olson timezone associated with the event
	xs:string?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-format-date]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-format-date]

Notes on the Saxon implementation

See format-dateTime

format-dateTime

format-dateTime($value as xs:dateTime?, $picture as xs:string) # xs:string?

Formats a dateTime, using a format controlled by the picture string. The result is equivalent to the 5-argument form of the function, with the third, fourth, and fifth arguments set to an empty sequence.

Table 17.78.

	$value	xs:dateTime?	The dateTime to be formatted

	$picture	xs:string	Picture showing how the dateTime is to be formatted
	xs:string?		

format-dateTime($value as xs:dateTime?, $picture as xs:string, $language as xs:string?, $calendar as xs:string?, $place as xs:string?) # xs:string?

Formats a dateTime, using a format controlled by the picture string.

Table 17.79.

	$value	xs:dateTime?	The dateTime to be formatted
	$picture	xs:string	Picture showing how the dateTime is to be formatted
	$language	xs:string?	The language for the output
	$calendar	xs:string?	The calendar for the output
	$place	xs:string?	The country or Olson timezone where the event took place
	xs:string?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-format-dateTime]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-format-dateTime]

Notes on the Saxon implementation

Formats a date, using a format controlled by the picture string. If no language is specified, the value is taken from the current Java locale. If the language (explicitly supplied or defaulted) is other than "en", the system tries to locate a localization class for the language (called, for historic reasons, a ; this class must provide methods to perform the localization.

Support for English (language="en") is built-in in all versions of the product.

In Saxon-PE and Saxon-EE, support for a variety of other European languages is also built in: specifically da, de, fr, fr_BE, he, it, nl, nl_BE, and sv. The localization modules for these languages have names such as `net.sf.saxon.option.local.Numberer_da`. These modules are not built in with Saxon-HE, but they are available as open source code, and it is possible to configure them by writing a subclass of `LocalizerFactory [Javadoc: net.sf.saxon.lib.LocalizerFactory]` and registering it with the `Configuration [Javadoc: net.sf.saxon.Configuration]`.

Localization modules for other languages can be configured. They are written to implement the interface `Numberer` [Javadoc: `net.sf.saxon.lib.Numberer`], typically as subclasses of `AbstractNumberer` [Javadoc: `net.sf.saxon.expr.number.AbstractNumberer`]. In Saxon-HE they are configured in the same way as the system-supplied languages. In Saxon-PE and Saxon-EE they can be registered in one of two ways: programmatically using the call `Configuration.getLocalizerFactory().setLanguageProperties("ja", props))` where `props` is a `Properties` object in which the `class` property is set to the name of the relevant `Numberer` class; or by setting an entry in the configuration file. For more information on writing localization modules, see Localizing numbers and dates.

If a calendar other than AD or ISO is specified, the result is prefixed "[Calendar: AD]" and is otherwise output as if the default calendar were used.

The country argument is currently used only when the format requests output of timezones by name (using `[ZN]`): for example with language="en", country="gb" and a date that falls in British Summer Time, +01:00 is output as "BST". This is problematic, because the information is not really available: the data type maintains only a time zone offset, and different countries (time zones) use different names for the same offset, at different times of year. If the value is a date or dateTime, and if the country argument is supplied, Saxon uses the Java database of time zones and daylight savings time (summer time) changeover dates to work out the most likely timezone applicable to the date in question.

If the timezone is formatted as [ZN,6] (specifically, with a minumum length of 6 or more) then the Olsen timezone name is output (again, this requires the country to be supplied). The Olsen timezone name generally takes the form `Continent/City`, for example "Europe/London" or "America/Los_Angeles". If the date/time is in daylight savings time for that timezone, an asterisk is appended to the Olsen timezone name.

When formatting times in the 12-hour clock, with language="en", the abbreviations "a.m." and "p.m." are used. These can be shortened to "am" and "pm" by giving a maximum width of 2 (`[PN,*-2]`). The US convention of denoting noon as "pm" and midnight as "am" is followed, unless the maximum width is 8 or more (`[PN,*-8]`) in which case these values are represented as "noon" and "midnight" respectively.

format-integer

Formats an integer according to a given picture string, using the conventions of a given natural language if specified.

format-integer($value as xs:integer?, $picture as xs:string) # xs:string

Table 17.80.

	$value	xs:integer?	The integer to be formatted
	$picture	xs:string	Picture showing how the integer is to be formatted
	xs:string		

format-integer($value as xs:integer?, $picture as xs:string, $language as xs:language) # xs:string

Table 17.81.

	$value	xs:integer?	The integer to be formatted
	$picture	xs:string	Picture showing how the integer is to be formatted
	$language	xs:language	Language used for the output
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-format-integer]

Notes on the Saxon implementation

Newly implemented in Saxon 9.3. Requires XPath 3.0 to be enabled.

format-number

format-number($value as numeric?, $picture as xs:string) # xs:string

Formats a number as specified by a picture string, using the default decimal format

Table 17.82.

	$value	numeric?	The number to be formatted
	$picture	xs:string	Picture showing how the number is to be formatted
	xs:string		

format-number($value as numeric?, $picture as xs:string, $decimal-format-name as xs:string) # xs:string

Formats a number as specified by a picture string, using the named decimal format

Table 17.83.

	$value	numeric?	The number to be formatted
	$picture	xs:string	Picture showing how the number is to be formatted
	$decimal-format-name	xs:string	Name of a decimal format definition defined in the context
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-format-number]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-format-number]

Notes on the Saxon implementation

From Saxon 9.3 this becomes available for XPath/XQuery as well as XSLT. The query prolog declarations for defining a decimal format in XQuery become available in Saxon 9.4.

format-time

Formats a time value

format-time($value as xs:time?, $picture as xs:string) # xs:string?

Table 17.84.

	$value	xs:time?	The xs:time value to be formatted
	$picture	xs:string	Picture showing how the time should be displayed
	xs:string?		

format-time($value as xs:time?, $picture as xs:string, $language as xs:string?, $calendar as xs:string?, $place as xs:string?) # xs:string?

Table 17.85.

	$value	xs:time?	The xs:time value to be formatted
	$picture	xs:string	Picture showing how the time should be displayed
	$language	xs:string?	The language for the output
	$calendar	xs:string?	The calendar for the output
	$place	xs:string?	The country or Olsen timezone where the event took place
	xs:string?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-format-time]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-format-time]

Notes on the Saxon implementation

See format-dateTime

function-arity

Returns the arity of the function identified by a function item.

function-arity($func as function(*)) # xs:integer

Table 17.86.

	$func	function(*)	The function item whose arity is required
	xs:integer		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-function-arity]

Notes on the Saxon implementation

Newly implemented in Saxon 9.3. Requires XPath 3.0 to be enabled.

function-available

function-available($function as xs:string) # xs:boolean

Determines whether a function with a given name (regardless of arity) is available in the context.

Table 17.87.

	$function	xs:string	The name of the requested function, as a lexical QName
	xs:boolean		

function-available($function as xs:string, $arity as xs:integer) # xs:boolean

Determines whether a function with the given name and arity is available in the context.

Table 17.88.

	$function	xs:string	The name of the requested function, as a lexical QName
	$arity	xs:integer	The arity of the requested function (number of arguments)
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 2.0 and later versions

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-function-available]

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-function-available]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

function-lookup

function-lookup($function as xs:string, $arity as xs:integer) # xs:boolean

Determines whether a function with a given name and arity is available in the context, and if so, returns a function item that can be used to call the function.

The function is useful to allow fallback action when a function is not available: for example, when calling an XPath 3.0 function, the code can be conditional on whether the 3.0 function is or is not available.

Table 17.89.

	$function	xs:string	The name of the requested function, as an xs:QName value
	$arity	xs:integer	The arity of the requested function (number of arguments)
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-function-lookup]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

function-name

Returns the name of the function identified by a function item.

function-name($func as function(*)) # xs:QName?

Table 17.90.

	$func	function(*)	The function whose name is required
	xs:QName?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-function-name]

Notes on the Saxon implementation

Newly implemented in Saxon 9.3. Requires XPath 3.0 to be enabled.

generate-id

generate-id() # xs:string

Returns a generated unique ASCII identifier for the context node

Table 17.91.

	xs:string

generate-id($arg as node()?) # xs:string

Returns a generated unique ASCII identifier for the specified node

Table 17.92.

	$arg	node()?	The node for which a generated identifier is required
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-generate-id]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-generate-id]

Notes on the Saxon implementation

Iin Saxon 9.3 this becomes available for use in XPath and XQuery as well as XSLT, provided that XPath 3.0 to be enabled.

has-children

Asks whether the supplied node has one or more children.

has-children() # xs:boolean

Table 17.93.

	xs:boolean

has-children($seq as node()) # xs:boolean

Table 17.94.

	$seq	node()	The input node
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-has-children]

Notes on the Saxon implementation

Not yet implemented in Saxon 9.4

head

Returns the first item in a sequence.

head($arg as item()*) # item()?

Table 17.95.

	$arg	item()*	The sequence whose first item is required
	item()?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-head]

Notes on the Saxon implementation

Newly implemented in Saxon 9.3. Requires XPath 3.0 to be enabled.

hours-from-dateTime

Extracts the hour component of a dateTime value

hours-from-dateTime($arg as xs:dateTime?) # xs:integer?

Table 17.96.

	$arg	xs:dateTime?	The dateTime value whose hour component is required
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-hours-from-dateTime]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-hours-from-dateTime]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

hours-from-duration

Extracts the hours component of a `dayTimeDuration` value

hours-from-duration($arg as xs:duration?) # xs:integer?

Table 17.97.

	$arg	xs:duration?	The duration value whose hours component is required
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-hours-from-duration]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-hours-from-duration]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

hours-from-time

Extracts the hours component of a time value. Note that this is from the localized value, not the normalized value: for example if the supplied time value is 01:23:00+05:00 then the result is 1.

hours-from-time($arg as xs:time?) # xs:integer?

Table 17.98.

	$arg	xs:time?	The time value whose hours component is required
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-hours-from-time]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-hours-from-time]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

id

Finds the elements with given ID attribute values

id($arg as xs:string*) # element()*

Table 17.99.

	$arg	xs:string*	The ID value being sought
	element()*		

id($arg as xs:string*, $node as node()) # element()*

Table 17.100.

	$arg	xs:string*	The ID value being sought
	$node	node()	The document in which the search takes place
	element()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-id]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-id]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

idref

Finds the nodes that link to the element with a given ID value. These will be element or attribute nodes marked by virtue of schema or DTD validation as IDREF or IDREFS values.

idref($arg as xs:string*) # node()*

Table 17.101.

	$arg	xs:string*	The ID value being sought
	node()*		

idref($arg as xs:string*, $node as node()) # node()*

Table 17.102.

	$arg	xs:string*	The ID value being sought
	$node	node()	The document in which the search takes place
	node()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-idref]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-idref]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

implicit-timezone

Returns the implicit timezone.

implicit-timezone() # xs:dayTimeDuration

Table 17.103.

	xs:dayTimeDuration

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-implicit-timezone]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-implicit-timezone]

Notes on the Saxon implementation

The implicit timezone in Saxon is always the same as the timezone component of the value returned by `current-dateTime()`.

index-of

Finds the positions of items in a sequence that match the second argument

index-of($seq as xs:anyAtomicType*, $search as xs:anyAtomicType) # xs:integer*

Table 17.104.

	$seq	xs:anyAtomicType*	The sequence being searched
	$search	xs:anyAtomicType	The value being sought
	xs:integer*		

index-of($seq as xs:anyAtomicType*, $search as xs:anyAtomicType, $collation as xs:string) # xs:integer*

Table 17.105.

	$seq	xs:anyAtomicType*	The sequence being searched
	$search	xs:anyAtomicType	The value being sought
	$collation	xs:string	The collation to be used for comparing strings
	xs:integer*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-index-of]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-index-of]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

innermost

Given a sequence of nodes, returns those nodes in the sequence that have no descendant that is also in the sequence.

innermost($seq as node()*) # node()*

Table 17.106.

	$seq	node()*	The input sequence
	node()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 3.0 only (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-innermost]

Notes on the Saxon implementation

Not yet implemented in Saxon 9.4

in-scope-prefixes

Returns the names of the namespaces that are in scope for an element. Except for the unnamed namespace, which is represented by the string "", the names will be of type `xs:NCName`.

in-scope-prefixes($element as element()) # xs:string*

Table 17.107.

	$element	element()	The element node whose in-scope namespace prefixes are required
	xs:string*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-in-scope-prefixes]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-in-scope-prefixes]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

insert-before

Insert an item into a sequence

insert-before($target as item()*, $position as xs:integer, $inserts as item()*) # item()*

Table 17.108.

	$target	item()*	Input sequence, into which new values are inserted
	$position	xs:integer	The position of the item before which the new values are inserted
	$inserts	item()*	The values to be inserted into the sequence
	item()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-insert-before]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-insert-before]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

iri-to-uri

Applies the %HH escaping convention to a URI, escaping only disallowed characters (but not reserved characters such as "/" and ":")

iri-to-uri($iri as xs:string?) # xs:string

Table 17.109.

	$iri	xs:string?	The supplied IRI which will be escaped to form a valid URI
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-iri-to-uri]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-iri-to-uri]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

lang

lang($testlang as xs:string?) # xs:boolean

Returns true if the xml:lang value for the context node matches the given language

Table 17.110.

	$testlang	xs:string?	The language being tested for
	xs:boolean		

lang($testlang as xs:string?, $node as node()) # xs:boolean

Returns true if the xml:lang value for the supplied node matches the given language

Table 17.111.

	$testlang	xs:string?	The language being tested for
	$node	node()	The node being tested
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-lang]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-lang]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

last

Returns the context size (the size of the sequence of items currently being processed)

last() # xs:integer

Table 17.112.

	xs:integer

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-last]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-last]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

local-name

local-name() # xs:string

Returns the local part of the name of the context node

Table 17.113.

	xs:string

local-name($arg as node()?) # xs:string

Returns the local part of the name of the supplied node

Table 17.114.

	$arg	node()?	The node whose local name is required
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-local-name]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-local-name]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

local-name-from-QName

Extracts the local name component of a QName value, as an xs:NCName

local-name-from-QName($arg as xs:QName?) # xs:NCName?

Table 17.115.

	$arg	xs:QName?	The QName whose local part is required
	xs:NCName?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-local-name-from-QName]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-local-name-from-QName]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

log

Returns the natural logarithm of `$arg`

log($arg as xs:double?) # xs:double?

Table 17.116.

	$arg	xs:double?	The input number
	xs:double?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-log]

Notes on the Saxon implementation

Newly implemented in Saxon 9.4. Available whether or not XPath 3.0 is enabled.

log10

Returns the base-10 logarithm of `$arg`

log10($arg as xs:double?) # xs:double?

Table 17.117.

	$arg	xs:double?	The input number
	xs:double?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-log10]

Notes on the Saxon implementation

Newly implemented in Saxon 9.4. Available whether or not XPath 3.0 is enabled.

lower-case

Translates characters in a string to lower case

lower-case($arg as xs:string?) # xs:string

Table 17.118.

	$arg	xs:string?	The string to be converted to lower-case
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-lower-case]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-lower-case]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

map

Applies the function item `$f` to every item from the sequence `$seq` in turn, returning the concatenation of the resulting sequences in order.

map($f as function(item()) as item()*, $seq as item()*) # item()*

Table 17.119.

	$f	function(item()) as item()*	The function item to be invoked on each member of the supplied sequence
	$seq	item()*	The input sequence: the supplied function will

			be applied to each item in this sequence
	item()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-map]

Notes on the Saxon implementation

Newly implemented in Saxon 9.3. Requires XPath 3.0 to be enabled.

map-pairs

Applies the function item `$f` to successive pairs of items taken one from `$seq1` and one from `$seq2`, returning the concatenation of the resulting sequences in order.

map-pairs($f as function(item(), item()) as item()*, $seq1 as item()*, $seq2 as item()*) # item()*

Table 17.120.

	$f	function(item(), item()) as item()*	The function which will be applied to each pair of items from the two sequences
	$seq1	item()*	The first sequence
	$seq2	item()*	The second sequence
	item()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-map-pairs]

Notes on the Saxon implementation

Newly implemented in Saxon 9.3. Requires XPath 3.0 to be enabled.

matches

Returns true if the given string matches the given regular expression

matches($input as xs:string?, $pattern as xs:string) # xs:boolean

Table 17.121.

	$input	xs:string?	The string to be matched against a regular expression
	$pattern	xs:string	The regular expression
	xs:boolean		

matches($input as xs:string?, $pattern as xs:string, $flags as xs:string) # xs:boolean

Table 17.122.

	$input	xs:string?	The string to be matched against a regular expression
	$pattern	xs:string	The regular expression
	$flags	xs:string	Flags that control the interpretation of the regular expression
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-matches]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-matches]

Notes on the Saxon implementation

Saxon 9.3 introduced support for the `q` flag, and for XPath 3.0 regular expression enhancements, provided XPath 3.0 is enabled.

max

Returns the highest value in a sequence of comparable items

max($arg as xs:anyAtomicType*) # xs:anyAtomicType?

Table 17.123.

	$arg	xs:anyAtomicType*	The input sequence
	xs:anyAtomicType?		

max($arg as xs:anyAtomicType*, $collation as xs:string) # xs:anyAtomicType?

Table 17.124.

	$arg	xs:anyAtomicType*	The input sequence
	$collation	xs:string	The collation to be used when comparing strings
	xs:anyAtomicType?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-max]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-max]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

min

Returns the lowest value in a sequence of comparable items

min($arg as xs:anyAtomicType*) # xs:anyAtomicType?

Table 17.125.

	$arg	xs:anyAtomicType*	The input sequence
	xs:anyAtomicType?		

min($arg as xs:anyAtomicType*, $collation as xs:string) # xs:anyAtomicType?

Table 17.126.

	$arg	xs:anyAtomicType*	The input sequence
	$collation	xs:string	The collation to be used when comparing strings
	xs:anyAtomicType?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-min]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-min]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

minutes-from-dateTime

Extracts the minutes component of a dateTime value

minutes-from-dateTime($arg as xs:dateTime?) # xs:integer?

Table 17.127.

	$arg	xs:dateTime?	The supplied dateTime value
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-minutes-from-dateTime]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-minutes-from-dateTime]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

minutes-from-duration

Extracts the minutes component of a duration value

minutes-from-duration($arg as xs:duration?) # xs:integer?

Table 17.128.

	$arg	xs:duration?	The supplied duration value
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-minutes-from-duration]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-minutes-from-duration]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

minutes-from-time

Extracts the minutes component of a time value

minutes-from-time($arg as xs:time?) # xs:integer?

Table 17.129.

	$arg	xs:time?	The supplied time value
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-minutes-from-time]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-minutes-from-time]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

month-from-date

Extracts the month component of a date value

month-from-date($arg as xs:date?) # xs:integer?

Table 17.130.

	$arg	xs:date?	The supplied date value
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-month-from-date]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-month-from-date]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

month-from-dateTime

Extracts the month component of a dateTime value

month-from-dateTime($arg as xs:dateTime?) # xs:integer?

Table 17.131.

	$arg	xs:dateTime?	The supplied dateTime value
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-month-from-dateTime]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-month-from-dateTime]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

months-from-duration

Extracts the months component of a duration value

months-from-duration($arg as xs:duration?) # xs:integer?

Table 17.132.

	$arg	xs:duration?	The supplied duration value
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-months-from-duration]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-months-from-duration]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

name

name() # xs:string

Returns the name of the context node, as a string in the lexical form of a QName

Table 17.133.

	xs:string

name($arg as node()?) # xs:string

Returns the name of the supplied node, as a string in the lexical form of a QName

Table 17.134.

	$arg	node()?	The node whose name is required
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-name]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-name]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

namespace-uri

namespace-uri() # xs:anyURI

Returns the namespace URI of the name of the context node

Table 17.135.

	xs:anyURI

namespace-uri($arg as node()?) # xs:anyURI

Returns the namespace URI of the name of the supplied node

Table 17.136.

	$arg	node()?	The node whose namespace URI is required
	xs:anyURI		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-namespace-uri]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-namespace-uri]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

namespace-uri-for-prefix

Returns the namespace URI corresponding to a given prefix, using the namespaces that are in scope for a given element

namespace-uri-for-prefix($prefix as xs:string?, $element as element()) # xs:anyURI?

Table 17.137.

	$prefix	xs:string?	The supplied prefix
	$element	element()	The element node whose in-scope namespaces are to be examined
	xs:anyURI?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-namespace-uri-for-prefix]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-namespace-uri-for-prefix]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

namespace-uri-from-QName

Extracts the namespace URI component of a QName value

namespace-uri-from-QName($arg as xs:QName?) # xs:anyURI?

Table 17.138.

	$arg	xs:QName?	The QName value whose URI component is required
	xs:anyURI?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-namespace-uri-from-QName]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-namespace-uri-from-QName]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

nilled

Returns true if the argument is an element that has the "nilled" property.

nilled($arg as node()?) # xs:boolean?

Table 17.139.

	$arg	node()?	The node (typically an element) to be examined to see if it is nilled
	xs:boolean?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-nilled]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-nilled]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

node-name

node-name() # xs:QName?

Returns the name of the context node, as a QName value (that is, a namespace URI plus local name)

Table 17.140.

	xs:QName?

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Specification [http://www.w3.org/TR/xpath-functions-11/#func-node-name]

node-name($arg as node()?) # xs:QName?

Returns the name of the given node, as a QName value (that is, a namespace URI plus local name)

Table 17.141.

	$arg	node()?	The node whose name is required
	xs:QName?		

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Specification [http://www.w3.org/TR/xpath-functions/#func-node-name]

XPath 3.0 Specification [http://www.w3.org/TR/xpath-functions-11/#func-node-name]

Notes on the Saxon implementation

The single-argument version of this function is new in XPath 3.0, and is first implemented in Saxon 9.3, provided that support for XPath 3.0 is enabled.

normalize-space

normalize-space() # xs:string

Eliminates redundant spaces from the string value of the context item

Table 17.142.

	xs:string

normalize-space($arg as xs:string?) # xs:string

Eliminates redundant spaces from the supplied string

Table 17.143.

	$arg	xs:string?	The string to be normalized
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-normalize-space]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-normalize-space]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

normalize-unicode

normalize-unicode($arg as xs:string?) # xs:string

Converts a string to Unicode normalized form NFC by modifying the way in which combining characters are represented

Table 17.144.

	$arg	xs:string?	The string to be normalized
	xs:string		

normalize-unicode($arg as xs:string?, $normalizationForm as xs:string) # xs:string

Converts a string to the specified Unicode normalization form by modifying the way in which combining characters are represented

Table 17.145.

	$arg	xs:string?	The string to be normalized
	$normalizationForm	xs:string	The Unicode normalization form to apply
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-normalize-unicode]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-normalize-unicode]

Notes on the Saxon implementation

Saxon supports normalization forms NFC, NFD, NFKC, and NFKD.

not

Returns true if the effective boolean value of the argument is false, and vice versa

not($arg as item()*) # xs:boolean

Table 17.146.

	$arg	item()*	
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-not]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-not]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

number

number() # xs:double

Equivalent to number(.); converts the value of the context item to a double

Table 17.147.

	xs:double

number($arg as xs:anyAtomicType?) # xs:double

Converts the supplied value to a double, or returns NaN if conversion is not possible

Table 17.148.

	$arg	xs:anyAtomicType?	
	xs:double		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-number]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-number]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

one-or-more

Tests whether `$srcval` contains one or more items; fails if it is an empty sequence.

one-or-more($arg as item()*) # item()+

Table 17.149.

	$arg	item()*	The sequence to be tested
	item()+		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-one-or-more]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-one-or-more]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

outermost

Given a sequence of nodes, returns those nodes in the sequence that have no ancestor that is also in the sequence.

outermost($seq as node()*) # node()*

Table 17.150.

	$seq	node()*	The input sequence
	node()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-outermost]

Notes on the Saxon implementation

Not yet implemented in Saxon 9.4

parse-json

This function takes as input a string in JSON format and parses it typically returning a map.

parse-json($arg as xs:string) # xs:string

Table 17.151.

	$arg	xs:string	The JSON input to be parsed
	xs:string		

parse-json($arg as xs:string, $options as map(*)) # document-node(element(*, xs:untyped))

Table 17.152.

	$arg	xs:string	The JSON input to be parsed
	$options	map(*)	Parsing options
	document-node(element(*, xs:untyped))		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 3.0 only (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-parse-json]

Notes on the Saxon implementation

Newly implemented in Saxon 9.4, provided XPath 3.0 is enabled. The parsing options provided are those listed in the XSLT 3.0 specifications.

parse-xml

This function takes as input an XML document represented as a string, and returns the document node at the root of an XDM tree representing the parsed document.

parse-xml($arg as xs:string) # document-node(element(*, xs:untyped))

Table 17.153.

	$arg	xs:string	The lexical XML string to be parsed as a document
	document-node(element(*, xs:untyped))		

parse-xml($arg as xs:string, $baseURI as xs:string) # document-node(element(*, xs:untyped))

Table 17.154.

	$arg	xs:string	The lexical XML string to be parsed as a document
	$baseURI	xs:string	The base URI property of the constructed document

	document-node(element(*, xs:untyped))		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-parse-xml]

Notes on the Saxon implementation

Newly implemented in Saxon 9.3, provided XPath 3.0 is enabled; replaces the extension function `saxon:parse` which is retained for the time being.

The second argument has been dropped from the latest draft of the XPath 3.0 specification, but remains available in the Saxon implementation for the time being.

path

This function takes as input a node (defaulting to the context node), and returns an XPath expression defining a path to that node from the root of its containing tree (which must be a document node). The path will use expanded QNames so that it is not sensitive to the namespace context.

path() # xs:string

Table 17.155.

	xs:string

path($arg as node()?) # xs:string?

Table 17.156.

	$arg	node()?	The node whose path is to be determined
	xs:string?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-path]

Notes on the Saxon implementation

Newly implemented in Saxon 9.4, provided XPath 3.0 is enabled; replaces the extension function `saxon:path` which is retained for the time being.

pi

Returns an approximation to the mathematical constant #.

pi() # xs:double

Table 17.157.

	xs:double

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions/math

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-pi]

Notes on the Saxon implementation

Implemented in Saxon 9.3; available whether or not support for XPath 3.0 is enabled

position

Returns the context position (that is, the position of the context item in the sequence currenly being processed)

position() # xs:integer

Table 17.158.

	xs:integer

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-position]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-position]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

pow

Returns `$arg1` raised to the power of `$arg2`.

exp($arg1 as xs:double?, $arg2 as numeric) # xs:double

Table 17.159.

	$arg1	xs:double?	The input number
	$arg2	numeric	The power to which the number is to be raised
	xs:double		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions/math

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-pow]

Notes on the Saxon implementation

Newly implemented in Saxon 9.4; available whether or not XPath 3.0 is enabled.

prefix-from-QName

Extracts the prefix component of a QName value. Returns an empty sequence if the QName has no prefix.

prefix-from-QName($arg as xs:QName?) # xs:NCName?

Table 17.160.

	$arg	xs:QName?	The QName whose prefix component is required
	xs:NCName?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-prefix-from-QName]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-prefix-from-QName]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

put

Writes an updated document to disk.

put($doc as node(), $uri as xs:string) # xs:NCName?

Table 17.161.

	$doc	node()	The document to be written to disk
	$uri	xs:string	The location where the document is to be written, as a file:/// URI
	xs:NCName?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to:

Notes on the Saxon implementation

Implemented in Saxon; available only when XQuery Updates is enabled.

QName

Constructs a QName value from a URI and local name. The second argument may be a lexical QName, and the prefix of the lexical QName is returned in the constructed value, for use if it is converted back to a string.

QName($paramURI as xs:string?, $paramQName as xs:string) # xs:QName

Table 17.162.

	$paramURI	xs:string?	The namespace URI component of the constructed QName
	$paramQName	xs:string	A lexical QName that supplies the local name component of the constructed QName, and optionally the prefix
	xs:QName		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-QName]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-QName]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

regex-group

Returns the contents of the substring that matched the n'th subexpression in a regular expression processed using `xsl:analyze-string`

regex-group() # xs:string

Table 17.163.

	xs:string

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 2.0 and later versions

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-regex-group]

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-regex-group]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

remove

Removes the item at a given position in a sequence

remove($target as item()*, $position as xs:integer) # item()*

Table 17.164.

	$target	item()*	The input sequence, from which an item is to be removed
	$position	xs:integer	The position of the item to be removed
	item()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-remove]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-remove]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

replace

Replaces sequences of characters within a string that match a given regular expression

replace($input as xs:string?, $pattern as xs:string, $replacement as xs:string) # xs:string

Table 17.165.

	$input	xs:string?	The input string, parts of which are to be replaced

	$pattern	xs:string	The regular expression matching parts of the string that are to be replaced
	$replacement	xs:string	The replacement string
	xs:string		

replace($input as xs:string?, $pattern as xs:string, $replacement as xs:string, $flags as xs:string) # xs:string

Table 17.166.

	$input	xs:string?	The input string, parts of which are to be replaced
	$pattern	xs:string	The regular expression matching parts of the string that are to be replaced
	$replacement	xs:string	The replacement string
	$flags	xs:string	Flags controlling how the regular expression is interpreted
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-replace]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-replace]

Notes on the Saxon implementation

Saxon 9.3 introduces support for the `q` flag, and for XPath 3.0 regular expression enhancements, provided XPath 3.0 is enabled.

resolve-QName

Expands a lexical QName using the in-scope namespaces from the given element

resolve-QName($qname as xs:string?, $element as element()) # xs:QName?

Table 17.167.

	$qname	xs:string?	The lexical QName to be expanded
	$element	element()	The element whose in-scope namespaces

			determine how the QName is expanded
	xs:QName?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-resolve-QName]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-resolve-QName]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

resolve-uri

resolve-uri($relative as xs:string?) # xs:anyURI?

Resolves a relative URI against the base URI from the static context

Table 17.168.

	$relative	xs:string?	A relative URI reference to be resolved against the static base URI from the context
	xs:anyURI?		

resolve-uri($relative as xs:string?, $base as xs:string) # xs:anyURI?

Resolves a relative URI against a specified base URI

Table 17.169.

	$relative	xs:string?	A relative URI reference to be resolved against the specified base URI
	$base	xs:string	The base URI used for resolving the relative reference
	xs:anyURI?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-resolve-uri]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-resolve-uri]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

reverse

Reverses the order of the items in the input sequence.

reverse($arg as item()*) # item()*

Table 17.170.

	$arg	item()*	The sequences whose items are to be reversed in order
	item()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-reverse]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-reverse]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

root

root() # node()

Returns the root node (typically but not necessarily a document node) of the tree containing the context node

Table 17.171.

	node()

root($arg as node()?) # node()?

Returns the root node (typically but not necessarily a document node) of the tree containing the supplied node

Table 17.172.

	$arg	node()?	A node in the tree whose root is required

	node()?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-root]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-root]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

round

round($arg as numeric?) # numeric?

Rounds a numeric value to the nearest whole number, rounding x.5 towards positive infinity.

Table 17.173.

	$arg	numeric?	The value to be rounded to the nearest whole number
	numeric?		

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Specification [http://www.w3.org/TR/xpath-functions/#func-round]

XPath 3.0 Specification [http://www.w3.org/TR/xpath-functions-11/#func-round]

round($arg as numeric?, $precision as xs:integer) # numeric?

Rounds a numeric value to the nearest multiple of ten to the power of minus `$precision`, rounding x.5 towards positive infinity.

Table 17.174.

	$arg	numeric?	The value to be rounded to a given number of decimal places
	$precision	xs:integer	The number of decimal places required
	numeric?		

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Specification [http://www.w3.org/TR/xpath-functions-11/#func-round]

Notes on the Saxon implementation

The two-argument form of this function is specified in XPath 3.0, and is newly supported in Saxon 9.3, provided XPath 3.0 is enabled.

round-half-to-even

round-half-to-even($arg as numeric?) # numeric?

Rounds a numeric value to the nearest whole number, rounding x.5 towards the nearest even number.

Table 17.175.

	$arg	numeric?	The value to be rounded to the nearest whole number
	numeric?		

round-half-to-even($arg as numeric?, $precision as xs:integer) # numeric?

Rounds a numeric value to the nearest multiple of ten to the power of minus `$precision`, rounding x.5 towards positive infinity.

Table 17.176.

	$arg	numeric?	The value to be rounded to a given number of decimal places
	$precision	xs:integer	The number of decimal places required
	numeric?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-round-half-to-even]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-round-half-to-even]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

seconds-from-dateTime

Extracts the seconds component of a dateTime value

seconds-from-dateTime($arg as xs:dateTime?) # xs:decimal?

Table 17.177.

	$arg	xs:dateTime?	The input dateTime value

	xs:decimal?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-seconds-from-dateTime]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-seconds-from-dateTime]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

seconds-from-duration

Extracts the seconds component of a dayTimeDuration value

seconds-from-duration($arg as xs:duration?) # xs:decimal?

Table 17.178.

	$arg	xs:duration?	The input duration value
	xs:decimal?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-seconds-from-duration]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-seconds-from-duration]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

seconds-from-time

Extracts the seconds component of a time value

seconds-from-time($arg as xs:time?) # xs:decimal?

Table 17.179.

	$arg	xs:time?	The input time value

	xs:decimal?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-seconds-from-time]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-seconds-from-time]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

serialize

This function serializes the supplied node `$arg`, returning the serialized node as a string.

serialize($arg as node()) # xs:string

Table 17.180.

	$arg	node()	The node (typically a document or element node) to be serialized
	xs:string		

serialize($arg as node(), $params as node()*) # xs:string

Table 17.181.

	$arg	node()	The node (typically a document or element node) to be serialized
	$params	node()*	Serialization parameters
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-serialize]

Notes on the Saxon implementation

This function is specified in XPath 3.0, and is newly supported in Saxon 9.3, provided XPath 3.0 is enabled. It replaces the Saxon extension function `saxon:serialize`

Serialization parameters are supplied by a sequence of nodes, in which the node name acts as the name of the serialization parameter, and its string value as the value of the parameter. This mechanism is based on that in an earlier draft of the XPath 3.0 specification: see here [http://www.w3.org/TR/2009/WD-xpath-functions-11-20091215/#func-serialize]

serialize-json

This function serializes the supplied value in JSON format.

serialize($arg as item()*) # xs:string

Table 17.182.

	$arg	item()*	The value to be serialized as a JSON string
	xs:string		

serialize($arg as item()*, $options as map(*)) # xs:string

Table 17.183.

	$arg	item()*	The value to be serialized as a JSON string
	$options	map(*)	JSON serialization optionss
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 3.0 only (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-serialize-json]

Notes on the Saxon implementation

This function is specified in XSLT 3.0, and is newly supported in Saxon 9.4, provided XSLT 3.0 is enabled.

The options recognized are `escape=true|false`, `indent=true|false`, `spec=RFC4234|ECMA-262`, `fallback=(function)`.

sin

Returns the sine of the argument, expressed in radians.

sin($# as xs:double?) # xs:double?

Table 17.184.

	$#	xs:double?	
	xs:double?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions/math

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-sin]

Notes on the Saxon implementation

Implemented in Saxon 9.3; available whether or not support for XPath 3.0 is enabled

sqrt

Returns the non-negative square root of the argument.

sqrt($arg as xs:double?) # xs:double?

Table 17.185.

	$arg	xs:double?	
	xs:double?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions/math

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-sqrt]

Notes on the Saxon implementation

Implemented in Saxon 9.3; available whether or not support for XPath 3.0 is enabled

starts-with

Tests whether one string starts with another string

starts-with($arg1 as xs:string?, $arg2 as xs:string?) # xs:boolean

Table 17.186.

	$arg1	xs:string?	The containing string
	$arg2	xs:string?	The supposed initial part of the string
	xs:boolean		

starts-with($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) # xs:boolean

Table 17.187.

	$arg1	xs:string?	The containing string
	$arg2	xs:string?	The supposed initial part of the string
	$collation	xs:string	The collation to be used for comparing characters
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-starts-with]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-starts-with]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

static-base-uri

Returns the base URI of the static context

static-base-uri() # xs:anyURI?

Table 17.188.

	xs:anyURI?

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-static-base-uri]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-static-base-uri]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

string

string() # xs:string

Returns the string value of the context node

Table 17.189.

	xs:string

string($arg as item()?) # xs:string

Returns the string value of the supplied node, or converts the supplied atomic value to a string.

Table 17.190.

	$arg	item()?	A node whose string value is required, or an atomic value to be converted to a string
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-string]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-string]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

string-join

string-join($arg1 as xs:string*) # xs:string

Concatenates all the strings in the given sequence, with no separator

Table 17.191.

	$arg1	xs:string*	A sequence of strings to be joined into one
	xs:string		

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Specification [http://www.w3.org/TR/xpath-functions-11/#func-string-join]

string-join($arg1 as xs:string*, $arg2 as xs:string) # xs:string

Concatenates all the strings in the given sequence, separated by the given separator

Table 17.192.

	$arg1	xs:string*	A sequence of strings to be joined into one
	$arg2	xs:string	The separator to be used between adjacent strings
	xs:string		

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Specification [http://www.w3.org/TR/xpath-functions/#func-string-join]

XPath 3.0 Specification [http://www.w3.org/TR/xpath-functions-11/#func-string-join]

Notes on the Saxon implementation

The single-argument form of this function is specified in XPath 3.0, and is newly supported in Saxon 9.3, provided XPath 3.0 is enabled. Note that the default separator is a zero-length string, not a single space.

string-length

string-length() # xs:integer

Returns the number of characters in the string value of the context item

Table 17.193.

	xs:integer

string-length($arg as xs:string?) # xs:integer

Returns the number of characters in the specified string

Table 17.194.

	$arg	xs:string?	The input string
	xs:integer		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-string-length]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-string-length]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

string-to-codepoints

Returns a sequence of integers representing the Unicode codepoints of the characters in the supplied string

string-to-codepoints($arg as xs:string?) # xs:integer*

Table 17.195.

	$arg	xs:string?	The input string

	xs:integer*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-string-to-codepoints]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-string-to-codepoints]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

subsequence

subsequence($sourceSeq as item()*, $startingLoc as xs:double) # item()*

Returns those items in the given sequence from the given starting position to the end of the sequence

Table 17.196.

	$sourceSeq	item()*	The input sequence
	$startingLoc	xs:double	The position of the first item from the input sequence to be included in the result
	item()*		

subsequence($sourceSeq as item()*, $startingLoc as xs:double, $length as xs:double) # item()*

Returns those items in the given sequence from the given starting position to the end of the sequence, or `$length` items if fewer.

Table 17.197.

	$sourceSeq	item()*	The input sequence
	$startingLoc	xs:double	The position of the first item from the input sequence to be included in the result
	$length	xs:double	The number of items to be included in the result
	item()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-subsequence]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-subsequence]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

substring

substring($sourceString as xs:string?, $start as xs:double) # xs:string

Returns a substring of a given string starting at the given starting position and continuing to the end of the string

Table 17.198.

	$sourceString	xs:string?	The input string
	$start	xs:double	The position of the first character of the input string to be included in the result
	xs:string		

substring($sourceString as xs:string?, $start as xs:double, $length as xs:double) # xs:string

Returns a substring of a given string starting at the given starting position and continuing to the end of the string, or `$length` characters if shorter.

Table 17.199.

	$sourceString	xs:string?	The input string
	$start	xs:double	The position of the first character of the input string to be included in the result
	$length	xs:double	The number of characters to be included in the result
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-substring]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-substring]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

substring-after

Returns that part of the given input string that occurs after the first occurrence of the string given in $operand2

substring-after($arg1 as xs:string?, $arg2 as xs:string?) # xs:string

Table 17.200.

	$arg1	xs:string?	The input string
	$arg2	xs:string?	A substring of the input string; the function returns the rest of the input string after this substring
	xs:string		

substring-after($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) # xs:string

Table 17.201.

	$arg1	xs:string?	The input string
	$arg2	xs:string?	A substring of the input string; the function returns the rest of the input string after this substring
	$collation	xs:string	The collation to be used for comparing characters
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-substring-after]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-substring-after]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

substring-before

Returns that part of the given input string that occurs before the first occurrence of the string given in $operand2

substring-before($arg1 as xs:string?, $arg2 as xs:string?) # xs:string

Table 17.202.

	$arg1	xs:string?	The input string
	$arg2	xs:string?	A substring of the input string; the function returns the part of the input string before this substring
	xs:string		

substring-before($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) # xs:string

Table 17.203.

	$arg1	xs:string?	The input string
	$arg2	xs:string?	A substring of the input string; the function returns the part of the input string before this substring
	$collation	xs:string	The collation to be used for comparing characters
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-substring-before]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-substring-before]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

sum

Returns the total of a sequence of numbers or durations

sum($arg as xs:anyAtomicType*) # xs:anyAtomicType

Table 17.204.

	$arg	xs:anyAtomicType*	The sequence of values to be totalled
	xs:anyAtomicType		

sum($arg as xs:anyAtomicType*, $zero as xs:anyAtomicType?) # xs:anyAtomicType?

Table 17.205.

	$arg	xs:anyAtomicType*	The sequence of values to be totalled
	$zero	xs:anyAtomicType?	The value to return if the input sequence is empty (default integer zero)
	xs:anyAtomicType?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-sum]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-sum]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

system-property

Returns the value of a system property

system-property($arg as xs:string) # xs:string

Table 17.206.

	$arg	xs:string	The name of the system property required
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 2.0 and later versions

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-system-property]

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-system-property]

Notes on the Saxon implementation

As well as the standard system properties defined in the XSLT namespace, the Saxon implementation will return the value of a Java system property (e.g. as set using -X on the Java VM invocation) if the name is unprefixed. It does NOT return the values of operating system environment variables.

More specifically, if the argument is a name in no namespace, that is, if the name is unprefixed, then the name is taken to refer to a Java system property, and the value of that property is returned if it exists. For example, on a Windows platform, `system-property('file.separator')` returns "\". This can be used to obtain information from the environment, and is especially useful in conjunction with `use-when` conditional compilation.

tail

Returns all but the first item in a sequence.

tail($arg as item()*) # item()*

Table 17.207.

	$arg	item()*	The sequence whose tail is required
	item()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-tail]

Notes on the Saxon implementation

Newly implemented in Saxon 9.3. Requires XPath 3.0 to be enabled.

tan

Returns the tangent of the argument, expressed in radians.

tan($# as xs:double?) # xs:double?

Table 17.208.

	$#	xs:double?	The input angle in radians
	xs:double?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions/math

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-tan]

Notes on the Saxon implementation

Implemented in Saxon 9.3; available whether or not support for XPath 3.0 is enabled

timezone-from-date

Extracts the timezone component of a date value

timezone-from-date($arg as xs:date?) # xs:dayTimeDuration?

Table 17.209.

	$arg	xs:date?	The supplied xs:date value
	xs:dayTimeDuration?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-timezone-from-date]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-timezone-from-date]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

timezone-from-dateTime

Extracts the timezone component of a dateTime value

timezone-from-dateTime($arg as xs:dateTime?) # xs:dayTimeDuration?

Table 17.210.

	$arg	xs:dateTime?	The supplied xs:dateTime value
	xs:dayTimeDuration?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-timezone-from-dateTime]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-timezone-from-dateTime]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

timezone-from-time

Extracts the timezone component of a time value

timezone-from-time($arg as xs:time?) # xs:dayTimeDuration?

Table 17.211.

	$arg	xs:time?	The supplied xs:time value
	xs:dayTimeDuration?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-timezone-from-time]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-timezone-from-time]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

tokenize

Returns a sequence of strings formed by breaking the input string at any substring that matches the given regular expression

tokenize($input as xs:string?, $pattern as xs:string) # xs:string*

Table 17.212.

	$input	xs:string?	The input string to be tokenized
	$pattern	xs:string	Regular expression matching the separators between tokens
	xs:string*		

tokenize($input as xs:string?, $pattern as xs:string, $flags as xs:string) # xs:string*

Table 17.213.

	$input	xs:string?	The input string to be tokenized

	$pattern	xs:string	Regular expression matching the separators between tokens
	$flags	xs:string	Flags controlling how the regular expression is interpreted
	xs:string*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-tokenize]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-tokenize]

Notes on the Saxon implementation

Saxon 9.3 introduces support for the `q` flag, and for XPath 3.0 regular expression enhancements, provided XPath 3.0 is enabled.

trace

Returns the value of the first argument after outputting a diagnostic message

trace($value as item()*, $label as xs:string) # item()*

Table 17.214.

	$value	item()*	The value to be traced, and to be returned as the function result
	$label	xs:string	Label to be included in the trace output
	item()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-trace]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-trace]

Notes on the Saxon implementation

Saxon by default outputs trace messages to `System.err`. However, the output may be redirected to a `TraceListener` or to an alternative output destination.

The Saxon implementation outputs the value of each item in a sequence as it is evaluated (except when the sequence is empty, in which case it outputs "empty sequence" at the start). Atomic values are output by converting them to a string, nodes by calling getPath() to generate a path expression to the node. With complex expressions the order of evaluation may be rather different from the expected order.

The trace output is directed to `System.err`, this may be redirected by using "2>log.txt" on the command line. If a TraceListener has been nominated, then instead of writing the output to `System.err`, the information instead results in events being notified to the TraceListener.

translate

Returns a string formed by replacing individual characters that appear in the second argument with the characters that appear at the corresponding position in the third argument

translate($arg as xs:string?, $mapString as xs:string, $transString as xs:string) # xs:string

Table 17.215.

	$arg	xs:string?	The string to be translated
	$mapString	xs:string	Characters to be replaced if they appear in the input string
	$transString	xs:string	Characters to be used as the replacement for corresponding characters in the second argument
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-translate]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-translate]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

true

Return the boolean value true

true() # xs:boolean

Table 17.216.

	xs:boolean

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-true]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-true]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

type-available

Returns true if a type with the given name is available, false otherwise

type-available($type as xs:string) # xs:boolean

Table 17.217.

	$type	xs:string	The name of the required type, as a lexical QName
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 2.0 and later versions

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-type-available]

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-type-available]

Notes on the Saxon implementation

The Saxon implementation has a minor restriction: if the argument is known only at run-time, then the function tests whether the type exists in the run-time configuration, which does not necessarily prove that it was present in the static context.

In Saxon the `type-available()` function can be used to check for the availability of Java classes. For example `type-available('jt:java.util.HashMap')` returns true, where the prefix `jt` is bound to the URI `http://saxon.sf.net/java-type`.

unordered

Returns a random permutation of its argument

unordered($sourceSeq as item()*) # item()*

Table 17.218.

	$sourceSeq	item()*	The input sequence
	item()*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-unordered]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-unordered]

Notes on the Saxon implementation

The Saxon implementation currently returns the input sequence unchanged, except where the expression used as the argument navigates a reverse axis (for example the ancestor or preceding-sibling axis), in which case the presence of the function causes the nodes to be returned in document order rather than reverse document order.

unparsed-entity-public-id

Return the public ID of an unparsed entity, given its name

unparsed-entity-public-id() # xs:string

Table 17.219.

	xs:string

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 2.0 and later versions

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-unparsed-entity-public-id]

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-unparsed-entity-public-id]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

unparsed-entity-uri

Return the system ID of an unparsed entity, given its name

unparsed-entity-uri() # xs:string

Table 17.220.

	xs:string

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XSLT 2.0 and later versions

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-unparsed-entity-uri]

XSLT 2.1 Specification [http://www.w3.org/TR/xslt-21/#function-unparsed-entity-uri]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

unparsed-text

Returns the contents of an external text file, given its URI. The function attempts to infer the encoding. First it looks in the HTTP headers if available. Then it examines the start of the file looking first for a byte-order-mark, and failing that for an XML declaration. If none of this works, it assumes the encoding is UTF-8.

unparsed-text($href as xs:string?) # xs:string?

Table 17.221.

	$href	xs:string?	
	xs:string?		

unparsed-text($href as xs:string?, $encoding as xs:string) # xs:string?

Table 17.222.

	$href	xs:string?	
	$encoding	xs:string	
	xs:string?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-unparsed-text]

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-unparsed-text]

Notes on the Saxon implementation

The current Saxon implementation is not deterministic: if the function is called twice with the same argument, it will read the external file twice, and may return different results if it has changed.

unparsed-text-available

Determines whether the corresponding call on `unparsed-text()` with the same arguments would succeed.

unparsed-text-available($href as xs:string?) # xs:boolean

Table 17.223.

	$href	xs:string?	The uri of the text file to be read
	xs:boolean		

unparsed-text-available($href as xs:string?, $encoding as xs:string) # xs:boolean

Table 17.224.

	$href	xs:string?	The uri of the text file to be read
	$encoding	xs:string	The encoding to be assumed for the text file
	xs:boolean		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-unparsed-text-available]

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-unparsed-text-available]

Notes on the Saxon implementation

The current Saxon implementation is not deterministic: if the function is called twice with the same argument, it will read the external file twice, and may return different results if it has changed; the fact that a file is available or unavailable does not guarantee that its status will remain unchanged for the rest of the query or transformation.

>The current implementation is inefficient: if a call on `unparsed-text-available()` is followed by a call on `unparsed-text()` to read the same file, the file will be read twice.

unparsed-text-lines

Equivalent to calling `unparsed-text()` and splitting the result at newline boundaries.

unparsed-text-lines($href as xs:string?) # xs:boolean

Table 17.225.

	$href	xs:string?	The uri of the text file to be read
	xs:boolean		

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Specification [http://www.w3.org/TR/xpath-functions-11/#func-unparsed-text-lines]

XSLT 2.0 Specification [http://www.w3.org/TR/xslt20/#function-unparsed-text-lines]

unparsed-text-lines($href as xs:string?, $encoding as xs:string) # xs:string*

Table 17.226.

	$href	xs:string?	The uri of the text file to be read
	$encoding	xs:string	The encoding to be assumed for the text file
	xs:string*		

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Specification [http://www.w3.org/TR/xpath-functions-11/#func-unparsed-text-lines]

Notes on the Saxon implementation

Newly implemented in Saxon 9.4; requires XPath 3.0 to be enabled.

upper-case

Converts a string to upper case

upper-case($arg as xs:string?) # xs:string

Table 17.227.

	$arg	xs:string?	The string to be converted to upper-case
	xs:string		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-upper-case]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-upper-case]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

uri-collection

Returns a sequence of `xs:anyURI` values representing the document URIs of the documents in a collection (either the default collection or a named collection).

uri-collection() # xs:anyURI*

Table 17.228.

	xs:anyURI*

uri-collection($arg as xs:string?) # xs:anyURI*

Table 17.229.

	$arg	xs:string?	The collection URI
	xs:anyURI*		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 3.0, XSLT 3.0, XQuery 3.0 (if enabled in Saxon: requires Saxon-PE or Saxon-EE)

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-uri-collection]

Notes on the Saxon implementation

Newly implemented in Saxon 9.3. Requires XPath 3.0 to be enabled.

The mechanisms for interpreting collection URIs are essentially the same as for the `collection` function, except that the collection URI resolver (if used) must return URIs rather than returning parsed document nodes directly.

year-from-date

Extracts the year component of a date value

year-from-date($arg as xs:date?) # xs:integer?

Table 17.230.

	$arg	xs:date?	The input date value
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-year-from-date]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-year-from-date]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

year-from-dateTime

Extracts the year component of a dateTime value

year-from-dateTime($arg as xs:dateTime?) # xs:integer?

Table 17.231.

	$arg	xs:dateTime?	The input dateTime value

	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-year-from-dateTime]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-year-from-dateTime]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

years-from-duration

Extracts the years component of a duration value

years-from-duration($arg as xs:duration?) # xs:integer?

Table 17.232.

	$arg	xs:duration?	The input duration value
	xs:integer?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-years-from-duration]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-years-from-duration]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

zero-or-one

Tests whether `$srcval` contains zero or one items; fails if it contains multiple items.

zero-or-one($arg as item()*) # item()?

Table 17.233.

	$arg	item()*	The input sequence to be tested

	item()?		

Links to W3C specifications

Namespace: http://www.w3.org/2005/xpath-functions

Applies to: XPath 2.0, XSLT 2.0, XQuery 1.0 and later versions

XPath 2.0 Functions and Operators [http://www.w3.org/TR/xpath-functions/#func-zero-or-one]

XPath 3.0 Functions and Operators [http://www.w3.org/TR/xpath-functions-30/#func-zero-or-one]

Notes on the Saxon implementation

The function is fully implemented according to the W3C specifications.

Chapter 18. Standards Conformance

Introduction

This section of the documentation describes the extent to which Saxon conforms to external specifications, notably W3C language specifications and Java API specifications.

Statements of conformance are made here in good faith, and are based on the results of running W3C test suites. To the extent that these test suites are incomplete, however, there may be edge case non-conformances that are not captured here. Any information about discrepancies or non-conformances not noted here will be welcomed, and the differences will either be corrected in the product, or documented in a future revision of this section.

XSLT 2.0 conformance

This release of Saxon is a complete implementation of the XSLT 2.0 Recommendation [http://www.w3.org/TR/xslt20/] of 23 January 2007, together with all errata published up to 10 April 2009, which are consolidated in the Proposed Edited Recommendation [http://www.w3.org/TR/2009/PER-xslt20-20090421/] of 21 April 2009.

Saxon-HE 9.x and Saxon-PE 9.x act as a , while Saxon-EE 9.x acts as a . The distinction is that a Basic XSLT Processor does not allow schemas to be imported and does not support validation of source or result documents or reference to user-defined types. These correspond to the two conformance levels defined in the XSLT 2.0 specification.

The XSLT 2.0 specification defines two optional conformance features, the serialization feature and the backwards compatibility feature. These optional features are implemented all three Saxon editions.

The following non-conformances apply only on the .NET platform, and only when using the Microsoft `System.Xml` parser:

- Under .NET, when the `System.Xml` parser is used, attributes declared in the DTD as being of type `ID` are not accessible using the `id()` function. This is because the `System.Xml` parser (`XMLValidatingReader`) does not make the DTD-defined attribute type available to the application.
- Similarly, when the `System.Xml` parser is used, unparsed entities are not reported to Saxon by the .NET parser, so the calls `unparsed-entity-uri()` and `unparsed-entity-public-id()` will always return a zero-length string.

These restrictions do not apply when a JAXP parser is used in place of the Microsoft parser. Since 9.3, the JAXP parser has therefore been the default. Setting the option `processor.SetProperty("http://saxon.sf.net/feature/preferJaxpParser", "true")` causes Saxon to use the Apache Xerces parser in preference to the (Microsoft) `System.Xml` parser. Xerces is bundled in the Saxon DLL; this parser is used in preference to the JAXP parser included in the OpenJDK library because it is more reliable and because it has no unnecessary references to other libraries.

Test results

Saxonica has submitted test results for the W3C XSLT Test Suite. At present this test suite, and the submitted results, are available to W3C members only. Saxon's submitted results in the suite (for Saxon 8.8, which was the first version to claim conformance), are available here [http://www.saxonica.com/conformance/xslts1.0.4/published-results8.8.html].

A number of bugs have been raised against the XSLT 2.0 specification which are not yet the subject of published errata. The most significant is Bug 5857 [http://www.w3.org/Bugs/Public/show_bug.cgi?

id=5857], which affects whether or not `xmlns=""` namespace undeclarations should appear in the result of copy operations. Saxon 9.4 partially implements the proposed fix to this bug.

Checklist of Implementation-Defined Items

The following list describes the way in which Saxon implements the features that the specification leaves implementation-defined. The numbering of items in the list corresponds to the numbering in the checklist provided as Appendix F of the XSLT 2.0 specification [http://www.w3.org/TR/xslt20/#implementation-defined-features].

1. Saxon offers a command line interface, and a Java API. The Java API conforms to the JAXP 1.3 Transformation interface defined in the JDK specifications, extended as necessary to support XSLT 2.0 facilities.

2. The mechanisms offered by Saxon are described in Extension instructions extension functions respectively.

3. In most cases Saxon allows the user to make this choice, using the options -w0, -w1, and -w2 on the command line (meaning respectively: ignore the error silently, continue after a warning, treat the error as fatal). Equivalent options are also available in the Java API.

   There are some cases where this does not apply:

   - XTRE1160 (unrecognized media type): this error is not detected, Saxon always takes the recovery action.
   - XTRE1630 (disable-output-escaping when the result is not serialized): this error is not detected, Saxon always takes the recovery action.

4. Saxon does extensive type checking at compile time, though it does not follow the precise inference rules as defined in the W3C . Errors are signaled statically only where a construct cannot possibly succeed (that is, where it will always fail with a type error at run time if evaluated). Warnings are signaled (a) where a path expression will always return an empty sequence (for example, @x/@y), and (b) in the case of a construct that can only succeed if the supplied value is an empty sequence, for example when comparing values whose statically-inferred types are `integer?` and `string?` respectively.

5. Saxon reports all serialization errors defined in the serialization specification, and treats them as fatal unless the serialization specification itself defines them as recoverable, in which case they are handled as warnings.

6. The only namespace that is specially recognized is `http://saxon.sf.net/`.

7. The user-defined data elements recognized by Saxon (for example, `saxon:collation`, `saxon:import-query`, and `saxon:script`) are described in extension instructions. Any other element in the Saxon namespace is signaled as an error.

8. Saxon supports backwards-compatible behavior.

9. Saxon allows a user-specified `URIResolver` to handle these URIs, in which case the forms of URI that are accepted depend on this `URIResolver`. By default, Saxon on the Java platform uses the mechanisms in the `java.net.URI` class of the underlying Java VM, while on the .NET platform the capabilities of the System.Uri class are used. The capabilities of these underlying

classes depend on the version and variant of the platform in use, and may also be customized by users.

Saxon places no restriction on the media type of a stylesheet module. Regardless of the media type, it accepts a "bare name" fragment identifier as a reference to an element within the retrieved document, identified by an attribute of type ID.

10. In addition to importing types using `xsl:import-schema`, Saxon implicitly imports a type corresponding to each class that is present in the Java classpath. These types have names in the namespace `http://saxon.sf.net/java-type`; the local name of the type is the same as the full name of the Java class (for example, `java.net.URI`, with any "$" signs replaced by hyphens. These types are intended for use with extension functions written in Java.

11. Saxon gives the user the choice. See Saxon and XML 1.1.

12. Saxon uses the timezone obtained from the system clock, unless the user specifies a different timezone as a run-time option.

13. Saxon allows localized numbering sequences to be defined by user-written plug-in code: see implementing localized numbers. In the absence of such a plug-in, the sequences that are supported are those defined in the specification, plus Greek upper-case (x0391), Greek lower-case (x03b1), Cyrillic upper case (x0410), Cyrillic lower-case (x0430), Hebrew (x05d0), Hiragana A (x3042), Katakana A (x30a2), Hiragana I (x3044), Katakana I (x30a4), and Kanji digits (x4e00). If an unrecognized letter is used as a formatting token, Saxon constructs a sequence starting with that letter and making use of the contiguous Unicode code-points starting with that letter that are classified as letters or digits. For example, the format token "x" produces the sequence x, y, z, xx, xy, xz, ...

14. Saxon imposes no limits on numbering sequences using letters or digits (other than those imposed by resource limitations). Roman numerals are handled in the range 1 to 9999, though values above 4000 are best avoided because there are no recognized conventions.

15. The default language is English. Localizations for number and date formatting are available for Belgian French, Danish, Dutch, English, French, Flemish, German, Italian, and Swedish. Other languages are supported only if user-written (or third-party) localization plug-ins are provided.

16. Any value other than "text" or "number" is treated as an error.

17. Saxon allows a user-written `CollationURIResolver` to interpret the collation URI, in which case there are no restrictions on the URI that is used. If the standard `CollationURIResolver` is used, two forms of URI are recognized: a URI declared using the `saxon:collation` element in the stylesheet, and a URI of the form `http://saxon.sf.net/collation?keyword=value;keyword=value;...` as described in Collations. It is also possible to register collations (with user-defined names) via the Java API.

18. Given the `lang` attribute, Saxon on the Java platform uses the Java Locale mechanisms to find a locale for that language, and hence a collation. On the .NET platform, Saxon similarly finds a collation appropriate to the .NET culture for that language. Given the `case-order` attribute, Saxon takes the collation that would be used in the absence of this attribute, changes its strength to `secondary` (making it case-blind), and then re-evaluates the result of any comparison performed by the base collator so that if the base collator decides two strings are equal, they are examined again to establish the effect of any case differences.

19.
Saxon ignores the media type entirely. Fragment identifiers are interpreted as bare names (matching ID attribute values) regardless of the media type.

20.
Saxon allows the localizations for particular languages to be defined as user-written plug-ins. The localizations supported for date/time formatting are the same languages that are supported for numbering (see above). The country argument is ignored except when determining a timezone name: in this case Saxon outputs a time zone name if the timezone is used in the specified country; if the timezone is attached to a date or dateTime then it also takes account of whether that date is known to be in daylight savings time (summer time) in the country in question.

21.
For English, the days of the week are Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday. If abbreviations are requested the values used are Mon, Tues, Weds, Thurs, Fri, Sat, Sun, right-truncated if necessary to the requested maximum length. If the minimum length is 1 and the maximum is 2, then the values used are M Tu We Th F Sa Su.

For English, the names of the months are January, February, March, April, May, June, July, August, September, October, November, and December. If abbreviation is requested, the leading part of the name to the required length is used (always returning at least three characters).

22.
The following table shows the values returned for system-defined properties, for both Saxon-B and Saxon-EE: here "9.4.0.1" is replaced by the current version number. The suffix "J" indicates the Java platform; this is replaced by "N" on the .NET platform.

Table 18.1.

Name	**Saxon-HE**	**Saxon-PE**	**Saxon-EE**	**Notes**
xsl:version	2.0	2.0	2.0	
xsl:vendor	Saxonica	Saxonica	Saxonica	Changed in Saxon 9.4
xsl:vendor-url	http://www.saxonica.com/	http://www.saxonica.com/	http://www.saxonica.com/	
xsl:product-name	SAXON	SAXON	SAXON	
xsl:product-version	HE 9.4.0.1J	PE 9.4.0.1J	EE 9.4.0.1J	For PE, and EE, if no license file is available then the string "(unlicensed)" is added after the edition code.
xsl:is-schema-aware	no	no	yes or no	depends on whether the particular transformation is schema-aware
xsl:supports-serialization	yes	yes	yes	
xsl:supports-backwards-compatibility	yes	yes	yes	
xsl:supports-namespace-axis	yes	yes	yes	See Erratum E14 to the specification

If the name of the system property is in the null namespace, Saxon returns the value of the Java system property whose name matches the local name.

23.
By default, messages are formatted as XML and written to the standard error output. Both the formatting and the destination can be customized through the Java API.

24.
Saxon does not validate the values that are returned. Invalid values may cause an error during subsequent processing, or may be written to the final output destination, resulting in ill-formed XML.

25.
Saxon represents external objects as a subtype of `xdt:anyAtomicType`. A QName is allocated to such types based on the Java class name of the external object, within the namespace "http://saxon.sf.net/java-type". The local name is the same as the expanded Java class name, with "$" replaced by "-".

26.
In the case of the principal result tree, the destination is specified using the JAXP API (as the second argument of the transform() method). If secondary result trees are not to be serialized to filestore, a user-written `OutputURIResolver` must be nominated. Saxon will pass all generated result trees to this class, which can then do what it likes with them.

27.
See previous item.

28.
The `href` attribute of `xsl:result-document` is interpreted as a relative URI, relative to the URI that defines the destination to which the principal result tree is serialized. This is defined by the `-o` option on the command line, or by the SystemID of the Result object supplied using the JAXP API.

29.
The default encoding is UTF-8.

30.
For HTML and XHTML, Saxon treats the version attribute as documentary only. For XML, versions 1.0 and 1.1 are recognized.

31.
A byte order mark is written only if explicitly requested (that is, the default is "no").

32.
Disable-output-escaping is supported provided that the final result tree is being written to a StreamResult. It can also be notified to a SAXResult, as described in the JAXP documentation. Disable-output-escaping is not supported when writing to a temporary tree.

XSLT 3.0 conformance

The XSLT 3.0 specification is far from complete, so at the time of writing it is not meaningful to discuss conformance in fine detail.

Broadly speaking:

- Saxon-HE does not implement any XSLT 3.0 features
- Saxon-PE implements a selection of XSLT 3.0 (and XPath 3.0) features, with the exception of schema-awareness and streaming

- Saxon-EE implements additional features relating to streaming (processing of a source document without constructing a tree in memory.

For information on streaming, see Streaming of large source documents.

At the time of writing, the latest published working draft is XSLT 2.1 [http://www.w3.org/TR/xslt-21/] published on 11 May 2010. Since then the WG has produced many internal drafts, and some of the new features appear in Saxon 9.4 ahead of publication by W3C. The WG has announced that XSLT 2.1 will be renumbered XSLT 3.0, and Saxon uses the new numbering.

XSLT 3.0 features are not available unless explicitly requested. The request can be by setting `-xsltversion:3.0` on the command line, by calling `setXsltLanguageVersion()` on the `XsltCompiler [Javadoc: net.sf.saxon.s9api.XsltCompiler]` object, or by use of the configuration setting `FeatureKeys.XSLT_VERSION [Javadoc: net.sf.saxon.lib.FeatureKeys]`. Setting `version="3.0"` on the `xsl:stylesheet` element is recommended, but is not sufficient on its own.

XSLT 3.0 features implemented in Saxon 9.4 include the following:

- The `xsl:iterate` instruction
- The `xsl:mode` declaration
- The `xsl:merge` instruction
- The `unparsed-text-lines()` function
- The `copy-of` and `snapshot()` functions
- The `xsl:try` instruction
- The `xsl:evaluate` instruction
- The syntax of patterns has been generalized; though not all the new XSLT 3.0 features are implemented.
- The `select` attribute of the `xsl:copy` instruction

Maps, as defined in the draft XSLT 3.0 specification, are implemented as an extension to XPath 3.0. For details see Maps in XPath 3.0.

XPath 2.0 conformance

This release of Saxon implements the full XPath 2.0 language as defined in the Second Edition Recommendation [http://www.w3.org/TR/xpath20/] of 14 December 2010.

There is a minor non-conformance in XPath 2.0 regular expression support, caused by bugs or restrictions in the underlying platform:

- When the "i" flag is used in conjunction with back-references, Saxon relies on the semantics of the underlying regular expression engine for case-independent matching. On most platforms the behaviour is not identical to the rules defined in XPath, though typically this only affects corner cases.

All the functions defined in the Functions and Operators [http://www.w3.org/TR/xpath-functions] specification are implemented. Information about the way they are implemented is provided in functions reference.

There is a non-conformance in the implementation of the `collection()` function: with the default collection URI resolver, the results are not guaranteed to be stable. That is, when the same collection

URI is used more than once, both the contents of the collection and the contents of the documents within the collection may be different on each occasion. It is possible to achieve stable results with a user-written collection URI resolver.

Implementation-defined aspects of Functions and Operators

The Functions and Operators specification includes a number of implementation-defined behaviors. This section describes how these are implemented in Saxon.

1. By default trace output is sent to the standard error output. It can be redirected in various ways: by redirecting System.err, by supplying a user-written MessageEmitter, or by changing the output destination of the standard MessageEmitter to a defined output stream or writer.

2. Saxon supports unlimited-precision integer arithmetic, so this provision does not apply.

3. Saxon supports unlimited-precision decimal arithmetic. For addition, subtraction, and multiplication, no rounding or truncation occurs; the full precision of the result is maintained. For division, the number of digits in the result is: the number of digits in the numerator, the number of digits in the divisor, or 18, whichever is greatest. Rounding of values in the range .0 to .5 inclusive is towards zero, otherwise away from zero.

4. For the Unicode character categories used in regular expressions, Saxon (since release 9.4) uses the tables from Unicode 6.0.0. For Unicode blocks in regular expressions, Saxon uses the definitions in Unicode 6.0.0, but also supports block names (such as "Greek") from earlier Unicode versions where the blocks have since been renamed.

   Some functions such as the implementations of `upper-case()` and `lower-case()` are delegated to the Java VM, and the result therefore depends on the Unicode version supported by the Java VM. In JDK 5.0 this is Unicode version 4.0.

   Other functions such as `normalize-unicode()` are implemented natively within Saxon. The character tables used for this are derived from the Unicode 4.0 database.

5. Saxon currently supports normalization forms NFC, NFD, NFKC, and NFKD.

6. Saxon supports this capability for a collation implemented using a Java `RuleBasedCollator`. Any Java `Comparator` can be used to implement a collation, but the substring matching functions will fail (error FOCH0004) if the `Comparator` is not a `RuleBasedCollator`.

   Saxon also supports this capability on the .NET platform. However, the results delivered by the collation support on the .NET platform do not appear to be 100% aligned with the XPath specification.

7. Saxon allows the year to be any 32-bit signed integer. The interpretation of negative years depends on whether the system is configured to use XSD 1.0 or XSD 1.1 semantics (in XSD 1.1, year zero exists, in XSD 1.0 it does not). Seconds are supported to a fractional precision of six digits (that is, microsecond precision).

8. Saxon allows the URI dereferencing to be performed using a user-supplied `URIResolver`, as defined in the JAXP specification, or using an `XmlResolver` on the .NET platform. Saxon also provides various options to control whitespace stripping, validation using a DTD or schema,

handling of errors, and support for XML 1.1. If appropriate configuration options are set, then query parameters are recognized in the URI to control some of these decisions.

XPath 3.0 Conformance

Saxon 9.4 implements nearly all the new features of the XPath 3.0 specification. At the time of writing the Working Groups are preparing a new Working Draft which is intended to have "Last Call" status; it is this working draft that has been used as the basis for this release.

New features that are fully implemented include the following:

- String contatenation operator `||`
- Simple mapping operator `!`
- Casting is allowed from a string to a union or list type
- Union types, provided they meet certain rules, can be used as a `SequenceType`
- Dynamic function call (Functions are first class values (items) in the data model)
- Function literals, for example `substring#2`
- Inline functions, for example `function($i) {$i*$i}`
- Partial application of functions, for example `concat('$', ?)`
- Let expressions (`let $v := expr return f($v)`)
- EQNames (`'uri':local`) wherever QNames are allowed

Maps are implemented in XPath 3.0 as described in the XSLT 3.0 specification. This is true whether or not the XPath processor is being invoked in an XSLT context. For details see Maps in XPath 3.0.

For details of the function library supported, see XSLT 2.0 and XPath 2.0 Functions.

XQuery 1.0 Conformance

This release of Saxon implements the full XQuery 1.0 language as defined in the Second Edition Recommendation [http://www.w3.org/TR/xquery/] of 14 December 2010. The restrictions noted with respect to XPath 2.0 apply equally to Saxon's support for XQuery 1.0.

XQuery conformance levels are described in Section 5 of the XQuery specification [http://www.w3.org/TR/xquery/#id-xquery-conformance].

Saxon-HE and Saxon-PE provide plus the following :

- Full Axis Feature
- Module Feature
- Serialization Feature

Saxon-EE provides plus the following :

- Schema Import Feature
- Schema Validation Feature

- Full Axis Feature
- Module Feature
- Serialization Feature

Neither Saxon product supports the Static Typing Feature, and there are no plans to do so.

There are some known non-conformances if the product is used in particular ways:

- In pull mode (-pull on the command line) there is a restriction that namespace undeclaration is not supported; this means that the query prolog option "copy-namespaces no-inherit" has no effect.
- Under .NET, if the `System.Xml` parser is selected, attributes declared in the DTD as being of type `ID` are not accessible using the `id()` function. This is because the `System.Xml` parser does not make the DTD-defined attribute type available to the application.

Conformance Tests

Saxonica has submitted results for the published W3C XQuery Test suite [http://www.w3.org/XML/Query/test-suite/]. W3C has published have generated both summary [http://www.w3.org/XML/Query/test-suite/XQTSReportSimple.html] and detail [http://www.w3.org/XML/Query/test-suite/XQTSReport.html] reports from the results; at the time of submission Saxon-EE was the only product to achieve a 100% pass rate.

The test driver used to measure these results is included in the Saxon distribution as part of the `saxon-resources-9.n` download.

Checklist of Implementation-Defined Items

Appendix D of the XQuery specification lists a number of features whose behavior is implementation-defined. For Saxon, the behavior is as follows:

1. For details of Saxon Unicode support, see XSLT 2.0 Conformance

2. Collation URIs of the form `http://saxon.sf.net/collation?` are always available: these map to the collations offered by the locales available within the Java VM in use. Additional collation URIs may be implemented by users and registered with the Saxon configuration via the Java API.

3. The implicit timezone is normally taken from the system clock. This may, however, be overridden via the Java API.

4. Warnings are raised for a variety of conditions. Compile time warnings include:

   - Use of a path expression that can never select anything (for example `@a/@b`)
   - Use of an expression that can only succeed if one or more operands are empty sequences
   - Use of a string literal in a PITest that is not a valid NCName

   Warnings may also be raised at run-time, for example if the `collection()` function fails to load a document and recovery has been requested.

   Compile-time warnings are sent to the JAXP `ErrorListener` registered with the `Configuration`; the default `ErrorListener` displays them as messages on `System.err`.

Run-time warnings are sent to the JAXP `ErrorListener` registered with the `Controller`; the default is the same.

5. Compile-time errors are sent to the JAXP `ErrorListener` registered with the `Configuration`; the default `ErrorListener` displays them as messages on `System.err`. Run-time errors are sent to the JAXP `ErrorListener` registered with the `Controller`; the default is the same. When queries are invoked from a Java application, any error will also be notified by throwing an exception.

6. Saxon gives the user the choice: see Saxon and XML 1.1. Note that since Saxon allows the user to select which XML parser to use on a per-document basis, achievement of consistency across "all aspects of the implementation" is in part a user responsibility. Saxon will not reject any attempt to use an XML 1.1 parser in a 1.0 configuration, or vice versa.

   Since Saxon 9.4, all operations in Saxon itself (as distinct from the underlying parser) that depend on the rules for XML Names use the rules that are defined in XML 1.0 fifth edition and in XML 1.1, rather than the rules defined in earlier XML 1.0 editions.

7. The following table shows the components of the static context. For each component, the second column shows whether the component is overwritten or augmented by Saxon itself. The third columns shows whether it may be overwritten or augmented by an application, through facilities in the Java API.

Table 18.2.

Component	Overritten or augmented by Saxon?	Can be set from Java API?
XPath 1.0 Compatibility Mode	no	no
Statically known namespaces	no	yes
Default element/type namespace	no	yes
Default function namespace	no	no
In-scope schema types	yes: types corresponding to Java classes for use in extension functions	no
In-scope element declarations	no	no
In-scope attribute declarations	no	no
In-scope variables	no	no
Context item static type	no	no
Function signatures	yes: any Java method on the classpath can be referenced	yes: there is a plug-in mechanism
Statically known collations	yes: collations of the form http://saxon.sf.net/collation?	yes
Default collation	no	yes
Construction mode	no	no
Ordering mode	no	no
Default order for empty sequences	no	no
Boundary-space policy	no	no
Copy-namespaces mode	no	no

Base URI	yes: inferred from the location of the query	yes
Statically known documents	no	no
Statically known collections	no	no
Statically known default collection type	no	no

For the dynamic context, the corresponding settings are:

Table 18.3.

Context item	no	yes
Context position	no	no
Context size	no	no
Variable values	no	only for declared external variables
Function implementations	no	yes (extension functions)
Current dateTime	yes (from system clock)	yes
Implicit timezone	yes (from system clock)	yes
Available documents	yes (all documents reachable using Java URL connections)	yes (URIResolver)
Available collections	no	yes (CollectionResolver/ catalog)
Default collection	no	yes (CollectionResolver/ catalog)

8. The Full-Axis features is supported.

9. Empty least

10. Saxon-EE recognizes one pragma: `saxon:validate-type`. For details, see XQuery extensions.

11. See XQuery Extensibility.

12. Java methods can be called as Extension Functions

13. The location hint is interpreted as a relative URI, relative to the base URI of the referencing module. It is then dereferenced using the Java URL mechanisms, unless the user has nominated a `ModuleURIResolver`, in which case the interpretation is under user control.

14. Not applicable.

15. Serialization may be invoked from the command line or from the Java or .NET API. Serialization parameters may be supplied on the command line, from the Java or .NET API, or through option declarations in the query prolog.

16.

As specified in Appendix C.3 of the XQuery specification.

XQuery 3.0 Conformance

Saxon 9.4 implements nearly all the new features of the XQuery 3.0 specification. At the time of writing the Working Groups are preparing a new Working Draft which is intended to have "Last Call" status; it is this working draft that has been used as the basis for this release.

See also XPath 3.0 Conformance; all the new features implemented for XPath 3.0 are also available in XQuery 3.0.

Other new features that are fully implemented include the following:

- Group by clause in FLWOR expressions
- Tumbling window and sliding window in FLWOR expressions
- Count clause in FLWOR expressions
- Outer joins, represented by the syntax `allowing empty` in a FLWOR expression
- Try/Catch expressions
- Private functions and variables
- Switch expressions
- Computed namespace constructors
- Output declarations to control serialization
- Decimal format declarations to control use of the `format-number` function.
- Validation against a named type
- Default values for external variables

Maps are implemented in XPath 3.0 as described in the XSLT 3.0 specification. This is true whether or not the XPath processor is being invoked in an XSLT context.

The main omissions are: function annotations other than private and public; annotations on inline functions; annotation assertions; some options for casting to union types (casting from string to a union type is fully supported).

In the try/catch clause, the error variables (such as `err:code` are not available).

For details of the function library supported, see XSLT 2.0 and XPath 2.0 Functions.

XML Schema 1.0 Conformance

Saxon Enterprise Edition (Saxon-EE) includes a complete implementation of XML Schema 1.0.

Test results have been submitted against the W3C XML Schema Test Suite. All deviations from the expected test results have been accounted for, and are believed to represent areas where the tests are either incorrect, or where implementations can legitimately differ.

Known limitations include the following:

- Saxon imposes limits on the values of `minOccurs` and `maxOccurs` appearing on any particle other than an element particle in a content model. These limits are designed to prevent out-of-memory errors. By default, the upper limit for `minOccurs` is 100 and the upper limit for `maxOccurs` (other than "unbounded") is 250. If these limits are exceeded, a warning is output, and the values specified are replaced with the highest allowed value (which is "unbounded" in the case of `maxOccurs`). The limits are configurable using the method `setOccurrenceLimits()` in the class `EnterpriseConfiguration`. If they are set too high, however, out-of-memory or stack overflow errors will occur either during schema compilation or during instance validation. Since Saxon 9.1, these limits no longer apply to the common case of element particles, which are now handled using counters.
- Saxon implements schema processing only to the extent required for XPath, XSLT, and XQuery processing and for assessment of document validity. This means that PSVI properties beyond those required for the XPath data model are not provided. It also means that validation errors are fatal, they do not result in a PSVI that indicates the validation outcome for individual nodes.
- Saxon uses the rules in XSD 1.1 rather than those in XSD 1.0 to evaluate type subsumption. The Saxon algorithm (an implementation of the algorithm published by Henry Thompson and Richard Tobin of the University of Edinburgh) is more accurate than the one in the XSD 1.0 specification in deciding whether one complex type subsumes another. It therefore permits some complex type restrictions that the cruder XSD 1.0 algorithm disallows.
- Saxon also uses the Thomson and Tobin algorithm to evaluate the Unique Particle Attribution constraint. The effect of this is that no UPA violation is reported when a complex type contains two distinct element particles that are references to the same element declaration, since the element declaration can be identified unambiguously.
- Saxon does not allow a user-defined type to be derived directly from `xs:anySimpleType`. It seems that the XSD 1.0 specification does not disallow this, though the effect of doing it is very unclear, and it has been banned in XSD 1.1.

XML Schema 1.1 Conformance

Saxon includes a complete implementation of the proposed specification of XSD 1.1. At the time of release of Saxon 9.4, the specification is very close to becoming a Recommendation, and there is a test suite that is believed to cover all new features of the specification. Saxon achieves 100% pass rate againts these tests. Saxonica has worked closely with other implementors of XSD 1.1 to ensure that implementations are fully interoperable. There is still, however, a possibility that the treatment of some edge cases in the specification will change, or that new tests will be added which Saxon does not pass.

XSD 1.1 features are available only if explicitly requested by using the -xsdversion:1.1 option on the command line, or equivalent options in the API.

The rules for valid type derivation follow the XML Schema 1.1 specification, regardless of this option setting. This is because the rules in the 1.0 specification do not meet the stated intent, namely that type derivation is valid in all cases where the restricted type allows a subset of the instances permitted by the base type.

An outline of the changes between XSD 1.0 and XSD 1.1 can be found in Appendix G of the specification [http://www.w3.org/TR/xmlschema11-1/#changes].

Implementation-defined features

Appendix E1 of XSD 1.1 (Part 1) provides a checklist of implementation-defined and implementation-dependent features. The following list describes how these are implemented in Saxon.

- The definition of whitespace is the same in all XML editions. The definition of the syntax of names is the same in XML 1.1 and in XML 1.0 fifth edition, and Saxon uses this definition

- Saxon-EE can read XML schema documents.

- Saxon-EE is able to read XML schema documents from the Web.

- Saxon-EE allows validation to be invoked from the command line, via a Java or .NET API, or from XSLT and XQuery. The documentation fors interfaces provides the required information.

- Saxon provides only that part of the PSVI used by the XQuery and XSLT specifications. This amounts to adding a type annotation to each element and attribute node where the outcome of validation is 'valid', and providing error messages where it is not.

- See previous answer.

- Saxon allocates unique names to anonymous type definitions.

- Saxon generally assembles schema components from XML schema documents as described in the XSD specification. However, it has built-in knowledge of the schema for the XML namespace, and these components can be included in a schema simply by importing the relevant namespace. It also has built-in knowledge of the schema components defined in the FN namespace, used in the result document of the `analyze-string` function in XPath 3.0

- The information is not directly available.

- See the checklist that follows.

- Saxon-EE attempts to detect all violations of Derivation Valid by examination of the schema in isolation; if it is not able to determine at this stage that the derivation is valid, the schema is considered to be in error.

- Saxon uses optimistic static type checking; it reports a static error in the XPath expression if its type analysis concludes that the type of a supplied value and the type required by the context in which it is used are disjoint, so that execution can never succeed.

- In considering whether two type tables are equivalent, Saxon tests the equivalence of XPath expressions by comparing their normalized expression trees. Inessential factors such as whitespace and parentheses are therefore ignored, and in some cases different expressions are recognized as equivalent, for example `(a=b)` is equivalent to `(b=a)`. In general the two expressions must have the same static context, but differences in the static context that do not affect the outcome of evaluating the expressions may be discounted.

- Saxon ignores the content of annotations entirely, other than ensuring that they are namespace-well-formed.

A similar checklist appears in appendix H.1 of XSD 1.1 Part 2. The corresponding answers for Saxon-EE are given below.

- See above. The definitions are taken from XML 1.0 Fifth Edition and XML 1.1, which are identical.

-

For integers, decimals, and strings, the only limits imposed by Saxon are those inherent in the underlying Java types `BigInteger`, `BigDecimal`, and `String`.

For calendar data types, the year must be in the range of a signed 32-bit integer.

For durations, the number of months must fit in a signed 32-bit integer; the integer number of seconds must fit in a signed 64-bit integer, and the precision is to microseconds.

- Saxon does not support any additional primitive datatypes.

- Saxon supports an additional facet, `saxon:preprocess`. This is a pre-lexical facet that invokes user-written Java code to modify a value before validation and before serialization. For example, this can be used to allow `xs:decimal` values to be written using comma as a decimal point. For details see The saxon:preprocess facet.

Serialization

The Serialization [] specification does not define its own conformance rules, saying instead that these are up to the host language to define.

Saxon implements all the mandatory provisions of the serialization specification.

The known non-conformances are:

- The serialization specification states that characters that can be natively encoded in the chosen encoding must be natively encoded and must not be represented using character or entity references. Saxon behaves this way by default, but provides an extension, `saxon:character-representation`, which changes the behavior. Such extensions have recently been declared non-conformant, but this one is retained in Saxon for backwards compatibility reasons.

The following page defines how Saxon interprets those aspects of the serialization specification that are implementation-defined.

Implementation-defined aspects of Serialization

This section defines how Saxon interprets those aspects of the serialization specification that are implementation-defined. The list follows the numbering of Appendix D [http://www.w3.org/TR/xslt-xquery-serialization/#implementation-defined-features] of the Serialization specification.

1. Sequence normalization takes place for all output methods, including user-defined output methods.

2. In such cases the local name of the method must be the name of a Java class that implements one of the interfaces org.xml.sax.ContentHandler, net.sf.saxon.event.Emitter, or net.sf.saxon.event.Receiver. The class is loaded from the classpath and then takes responsibility for producing the serialized output (if any). The actual namespace URI is ignored.

3. Any value other than those listed is an error.

4. Saxon allows the serialized output to be written to a Java Writer, which is a character stream. In this case no encoding takes place.

5. Saxon defines a factory class that enables Java applications to insert user-defined classes into the serialization pipeline. This mechanism could be used to override the standard CDATA processing.

XQuery Update 1.0

Saxon 9.4 implements XQuery Update Facility 1.0, which finally reached Recommendation status on 17 March 2011. All features, including optional features, are implemented.

Update is currently supported only on documents built using the `linked` tree model.

There are several limitations worth noting:

- The specification requires that updates should be atomic: that is, if any failure occurs, the original document is restored to its original state. Saxon meets this requirement unless revalidation is requested. If failures occur during the revalidation phase, these cannot currently be rolled back. Of course, this only affects the in-memory copy of the document; the application should be organized so that in this situation it is possible to revert to the copy on disk.
- The specification requires that node identity should be preserved across updates. The extent to which this requirement is met by Saxon is debatable. Saxon allows several Java objects to represent the same node. With the linked tree this applies in particular to attribute nodes: two requests to get all the attributes of a node will return different Java objects, though comparisons using `equals()` or `isSameNodeInfo()` will reveal that they refer to the same nodes. After an update, it is in general not safe to continue using existing `NodeInfo` objects, and the effect of doing so is undefined. This means that it is impossible to determine in any reliable way whether the old nodes have the same identity as the new nodes. In practice, with the linked tree implementation, it is safe to continue using variables that refer to element nodes (where there is always one Java object per node), but it is not safe in the case of attribute nodes.
- The `put()` function is implemented using the same run-time support code as `xsl:result-document`, and it currently imposes similar restrictions: for example, it must not be called from within a variable initialization or function body, and it must to write to a URI that has been read during the execution of the query; also, two calls on `put()` must not write to the same URI. However, it does use the XQuery semantics for resolving the relative URI.
- The specification change introduced by bug 9432 [http://www.w3.org/Bugs/Public/show_bug.cgi?id=9432], which is included in the final Recommendation, has not been implemented in Saxon 9.4. The effect of this is that an updated document may sometimes be missing the namespace undeclaration `xmlns=""` where the specification requires it.

Conformance with other specifications

Saxon is dependent on the user-selected XML parser to ensure conformance with the XML 1.0 or 1.1 Recommendation and the XML Namespaces Recommendation. The syntax for names appearing in XPath expressions follows the XML 1.0 or 1.1 rules depending on Saxon configuration settings.

Saxon implements the `<?xml-stylesheet?>` processing instruction as described in the W3C Recommendation . The pseudo-attribute must be a URI identifying an XML document containing a stylesheet, or a URI with a fragment identifier identifying an embedded stylesheet. The fragment must be the value of an ID attribute declared as such in the DTD.

Saxon's two native tree models, the standard tree and the tiny tree, both support the `xml:id` Recommendation. An attribute named `xml:id` is recognized by the `id()` function, provided that its value after space-trimming is a valid `NCName`. Saxon's schema processor imposes the constraint that an `xml:id` attribute, if allowed at all, must be declared as being of type `xs:ID`.

Saxon on the Java platform works with any SAX2-conformant XML parser that is configured to enable namespace processing. There is one limitation: on the startElement() call from the XMLReader to the ContentHandler, the QName (that is, the third argument) must be present. According to the SAX2 specification, namespace-aware parsers are not obliged to supply this argument. However, all commonly-used parsers appear to do so.

Saxon on the Java platform should work with any DOM-conformant XML parser, however, Saxon's DOM interface is tested only with Crimson and Xerces, and DOM implementations are known to vary widely. Saxon has been used successfully with the Oracle DOM implementation, though this is not included in the standard test suite and problems have occasionally been reported with this combination.

When a XOM tree is supplied as the transformation input, Saxon does not combine adjacent text nodes into a single node. Adjacent text nodes can occur as the result of user modifications to the tree, or as a result of the presence of CDATA sections or entity references, depending on the options in force when the tree was constructed.

Character Encodings Supported

The encodings supported on input depend entirely on your choice of XML parser.

On output, any encoding supported by the Java VM or the .NET platform (as appropriate) may be used.

A list of the character encodings supported by the Java VM can be obtained by using the command `java net.sf.saxon.charcode.CharacterSetFactory`, with no parameters.

JAXP Conformance

Saxon on the Java platform implements the JAXP 1.3 API. This is available as a standard part of JDK 1.5 (also known as J2SE SDK 5.0), and is available as an optional package for use with JDK 1.4.

Saxon implements the interfaces in the `javax.xml.transform` package in full, including support for SAX, DOM, and Stream input, and SAX, DOM, and Stream output.

Saxon implements the XPath API (the interfaces in the `javax.xml.xpath` package) in full. Note however that the `XPathException` exception used throughout Saxon is unrelated to the class of the same name defined in the XPath API. The Saxon XPath API works with five object models: DOM, JDOM, XOM, DOM4J, and the native Saxon object model. The URIs used to identify these object models are:

- http://java.sun.com/jaxp/xpath/dom
- http://jdom.org/jaxp/xpath/jdom
- http://www.xom.nu/jaxp/xpath/xom
- http://www.dom4j.org/jaxp/xpath/dom4j
- http://saxon.sf.net/jaxp/xpath/om

The JAXP 1.3 XPath API is designed primarily for use with XPath 1.0. Saxon implements it with XPath 2.0. This means that decisions were necessary on how to handle the richer set of return types available with 2.0. If the return type requested is String, Number, or Boolean, then Saxon converts the result to one of these types as if by using the XPath functions `string()`, `number()`, or `boolean()`. Items after the first in the atomized sequence are discarded. If the return type requested is `NODE`, Saxon returns the first item in the result sequence if it is a node, and reports an error if it is an atomic value. If the result sequence is empty, it returns null. If the return type requested is NODELIST, Saxon returns a Java List containing all the items in the sequence, whether they are nodes or not. Nodes are returned using the native node object in the input data model, atomic values are returned using the most appropriate Java class. Note that in the case of numeric results, it is not always easy to predict whether the result will be a Long, a Double, or a BigDecimal, and it is advisable to cast the data to one of the numeric types within the XPath expression to make the result predictable.

Saxon-EE also implements the JAXP 1.3 Validation API. This allows a schema to be parsed and validated, and provides two mechanisms for validating a document against a schema: the `Validator` and the `ValidatorHandler`.

There are some minor non-conformances in the Saxon implementation of this interface:

- The interface specification restricts the types of `Source` and `Result` object that can be supplied to a `Validator`. Saxon does not enforce these restrictions, it allows any kind of `Source` and `Result` that a `Transformer` would accept.
- Saxon's implementation of `ValidatorHandler` performs more buffering of events than is permitted by the specification.
- The method `isSpecified(int)` in the `TypeInfoProvider` always returns true.

In addition, Saxon implements part of the `javax.xml.parsers` API. Saxon no longer provides its own SAX parser, however it does provide a `DocumentBuilder`. The DOM interfaces are limited by the capabilities of the Saxon DOM, specifically the fact that it is read-only. Nevertheless, the DocumentBuilder may be used to construct a Saxon tree, or to obtain an empty Document node which can be supplied in a DOMResult to hold the result of a transformation.

XQJ Conformance

Saxon implements XQJ, the XQuery API for Java defined in JSR 225 [http://www.jcp.org/en/jsr/detail?id=225]. The current version that is implemented is the Final Release dated March 2009, which was publicly announced on 24 June 2009.

The compliance definition for XQJ (section 3 of the specification) requires a statement of how all aspects of the specification that are implementation-defined have been implemented. The following table provides this statement for the Saxon implementation.

Table 18.4.

The class name of the XQDataSource implementation	`net.sf.saxon.xqj.SaxonXQDataSource`
All properties defined on the XQDataSource implementation	The following properties are defined: allowExternalFunctions, dtdValidation, expandAttributeDefaults, expandXInclude, retainLineNumbers, schemaValidationMode, stripWhitespace, useXsiSchemaLocation, xmlVersion, xsdVersion.
The syntax and semantics for commands, assuming executing commands through XQExpression is supported.	No commands are supported (only XQuery expressions).
Is cancelling of query execution supported?	No, Query execution cannot be cancelled.
The default and supported values for each parameter described in XQuery Serialization	Although this is implementation-defined, the test suite makes some assumptions and these have been followed. The defaults are: byte-order-mark="no" cdata-section-elements="" doctype-public=null doctype-system=null encoding="utf-8" indent="yes" media-type="application/xml" method="xml" normalization-form="none" omit-xml-declaration="yes" standalone="omit" undeclare-prefixes="no" use-character-maps="" version="1.0"
Additional StAX or SAX event types being reported, beside the event types documented in [the] specification	None.

Support for XDM instances and types based on user-defined schema types	When used with Saxon-EE, user-defined schema types are supported, to the extent that the XQJ interface allows them to be used.
The semantics with respect to node identity, document order, and full node context, when a node is bound to an external variable.	When a node is bound to an XQItem and hence to a variable in a query, node identity, document order, and "context" (relationships to other nodes) are maintained.
Is login timeout supported?	No. (There is no concept of login.)
Are transactions supported?	No. Saxon only supports read-only query via the XQJ interface.
Behaviour of the `getNodeUri()` method, defined on `XQItemAccessor`, for other than document nodes.	The method is defined on any node, and returns the URI of the external entity in which the containing element originally appeared, if known, or the empty URI otherwise.
Behaviour of the `getTypeName()` method, defined on `XQItemType`, for anonymous types.	Anonymous types have a system-generated name.
Behaviour of the `getSchemaURI()` method, defined on `XQItemType`	The system identifier (document URI) of the original schema document is reported if the information is available.
Behaviour of the `createItemFromDocument()` methods, defined on `XQDataFactory`, if the specified value is not a well-formed XML document.	An exception is thrown.
Behaviour of the `bindDocument` methods, defined on `XQDynamicContext`, if the specified value is not a well-formed XML document.	An exception is thrown.
The error codes, reported through `XQQueryException`, in addition to the standard error codes listed in [XQuery] and its associated specifications.	None. The Saxon implementation does not currently use the class `XQQueryException`.

Section 19 of the XQJ specification ("Interoperability") suggests that an XQJ specification should accept items and built-in types that were created using a different vendor's XQJ implementation. In general, Saxon will only handle the Saxon implementation of XQJ interfaces. At any rate, it has not been validated with third-party implementations.

Chapter 19. Alphabetical Index

Introduction

Click on the initial letter of the term you are looking for:

- | . | 1 | 2 | 3 | 9 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | Θ

head($arg as item()*) → item()?

hours-from-dateTime($arg as xs:dateTime?) → xs:integer?

hours-from-duration($arg as xs:duration?) → xs:integer?

hours-from-time($arg as xs:time?) → xs:integer?

id($arg as xs:string*, $node as node()) → element()*

id($arg as xs:string*) → element()*

idref($arg as xs:string*, $node as node()) → node()*

idref($arg as xs:string*) → node()*

local-name($arg as node()?) → xs:string

local-name-from-QName($arg as xs:QName?) → xs:NCName?

log($arg as xs:double?) → xs:double?

log10($arg as xs:double?) → xs:double?

lower-case($arg as xs:string?) → xs:string

max($arg as xs:anyAtomicType*, $collation as xs:string) → xs:anyAtomicType?

max($arg as xs:anyAtomicType*) → xs:anyAtomicType?

min($arg as xs:anyAtomicType*, $collation as xs:string) → xs:anyAtomicType?

min($arg as xs:anyAtomicType*) → xs:anyAtomicType?

minutes-from-dateTime($arg as xs:dateTime?) → xs:integer?

minutes-from-duration($arg as xs:duration?) → xs:integer?

minutes-from-time($arg as xs:time?) → xs:integer?

month-from-date($arg as xs:date?) → xs:integer?

month-from-dateTime($arg as xs:dateTime?) → xs:integer?

months-from-duration($arg as xs:duration?) → xs:integer?

name($arg as node()?) → xs:string

namespace-uri($arg as node()?) → xs:anyURI

namespace-uri-from-QName($arg as xs:QName?) → xs:anyURI?

nilled($arg as node()?) → xs:boolean?

node-name($arg as node()?) → xs:QName?

normalize-space($arg as xs:string?) → xs:string

normalize-unicode($arg as xs:string?, $normalizationForm as xs:string) → xs:string

normalize-unicode($arg as xs:string?) → xs:string

not($arg as item()*) → xs:boolean

XSL