Document Object Model (DOM) Level 3 Load and Save Specification

Version 1.0

W3C Proposed Recommendation 05 February 2004

This version:
http://www.w3.org/TR/2004/PR-DOM-Level-3-LS-20040205
Latest version:
http://www.w3.org/TR/DOM-Level-3-LS
Previous version:
http://www.w3.org/TR/2003/CR-DOM-Level-3-LS-20031107

Editors:
Johnny Stenback, *Netscape*
Andy Heninger, *IBM (until March 2001)*

This document is also available in these non-normative formats: XML file, plain text, PostScript file, PDF file, single HTML file, and ZIP file.

Copyright ©2004 W3C® (MIT, ERCIM, Keio), All Rights Reserved. W3C liability, trademark, document use and software licensing rules apply.

Abstract

This specification defines the Document Object Model Load and Save Level 3, a platform- and language-neutral interface that allows programs and scripts to dynamically load the content of an XML document into a DOM document and serialize a DOM document into an XML document; DOM documents being defined in [DOM Level 2 Core] or newer, and XML documents being defined in [XML 1.0] or newer. It also allows filtering of content at load time and at serialization time.

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This document contains the Document Object Model Level 3 Load and Save specification and is a Proposed Recommendation. It has been produced as part of the W3C DOM Activity. The authors of this document are the DOM Working Group members.

It is based on the feedback received during the Candidate Recommendation period. An implementation report is available. Changes were mostly made in the handling of exceptions, and default value of parameters.

W3C Advisory Committee Representatives are now invited to submit their formal review via Web form, as described in the Call for Review. Additional comments may be sent to a Team-only list, dom-review@w3.org. The public is invited to send comments to the public mailing list www-dom@w3.org (public archive). The review period ends on 5 March 2004.

Publication as a Proposed Recommendation does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

Patent disclosures relevant to this specification may be found on the Working Group's patent disclosure page.

Table of contents

Expanded Table of Contents 3
W3C Copyright Notices and Licenses 5

1. Document Object Model Load and Save 9

Appendix A: IDL Definitions 39
Appendix B: Java Language Binding 43
Appendix C: ECMAScript Language Binding 49
Appendix D: Acknowledgements 53
Glossary 55
References 57
Index 61

Expanded Table of Contents

Expanded Table of Contents 3
W3C Copyright Notices and Licenses 5
W3C® Document Copyright Notice and License 5
W3C® Software Copyright Notice and License 6
W3C® Short Software Notice 7

1 Document Object Model Load and Save 9
1.1 Overview of the Interfaces 9
1.2 Basic types 9
1.2.1 The LSInputStream type 9
1.2.2 The LSOutputStream type 10
1.2.3 The LSReader type 10
1.2.4 The LSWriter type 10
1.3 Fundamental interfaces 11

Appendix A: IDL Definitions 39
Appendix B: Java Language Binding 43
Appendix C: ECMAScript Language Binding 49
Appendix D: Acknowledgements 53
D.1 Production Systems 53
Glossary 55
References 57
1 Normative references 57
2 Informative references 58
Index 61

Expanded Table of Contents

W3C Copyright Notices and Licenses

Copyright © 2004 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved.

This document is published under the W3C® Document Copyright Notice and License [p.5] . The bindings within this document are published under the W3C® Software Copyright Notice and License [p.6] . The software license requires "Notice of any changes or modifications to the W3C files, including the date changes were made." Consequently, modified versions of the DOM bindings must document that they do not conform to the W3C standard; in the case of the IDL definitions, the pragma prefix can no longer be 'w3c.org'; in the case of the Java language binding, the package names can no longer be in the 'org.w3c' package.

W3C® Document Copyright Notice and License

Note: This section is a copy of the W3C® Document Notice and License and could be found at http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231.

Copyright © 2004 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved.

http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231

Public documents on the W3C site are provided by the copyright holders under the following license. By using and/or copying this document, or the W3C document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the W3C document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on *ALL* copies of the document, or portions thereof, that you use:

1. A link or URL to the original W3C document.
2. The pre-existing copyright notice of the original author, or if it doesn't exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright © [$date-of-document] World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231"
3. *If it exists*, the STATUS of the W3C document.

When space permits, inclusion of the full text of this **NOTICE** should be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of W3C documents is granted pursuant to this license. However, if additional requirements (documented in the Copyright FAQ) are satisfied, the right to create modifications or derivatives is sometimes granted by the W3C to individuals complying with those

requirements.

THIS DOCUMENT IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

W3C® Software Copyright Notice and License

Note: This section is a copy of the W3C® Software Copyright Notice and License and could be found at http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231

Copyright © 2004 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved.

http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231

This work (and included software, documentation such as READMEs, or other related items) is being provided by the copyright holders under the following license. By obtaining, using and/or copying this work, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions.

Permission to copy, modify, and distribute this software and its documentation, with or without modification, for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the software and documentation or portions thereof, including modifications:

1. The full text of this NOTICE in a location viewable to users of the redistributed or derivative work.
2. Any pre-existing intellectual property disclaimers, notices, or terms and conditions. If none exist, the W3C® Short Software Notice [p.7] should be included (hypertext is preferred, text is permitted) within the body of any redistributed or derivative code.
3. Notice of any changes or modifications to the files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SOFTWARE OR DOCUMENTATION.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the software without specific, written prior permission. Title to copyright in this software and any associated documentation will at all times remain with copyright holders.

W3C® Short Software Notice

Note: This section is a copy of the W3C® Short Software Notice and could be found at http://www.w3.org/Consortium/Legal/2002/copyright-software-short-notice-20021231

Copyright © 2004 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved.

Copyright © [$date-of-software] World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. This work is distributed under the W3C® Software License [1] in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

[1] http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231

1. Document Object Model Load and Save

Editors:
Johnny Stenback, Netscape
Andy Heninger, IBM (until March 2001)

This section defines a set of interfaces for loading and saving document objects as defined in [*DOM Level 2 Core*] or newer. The functionality specified in this section (the *Load and Save* functionality) is sufficient to allow software developers and web script authors to load and save XML content inside conforming products. The DOM Load and Save API [p.55] also allows filtering of XML content using only DOM API calls; access and manipulation of the `Document` is defined in [*DOM Level 2 Core*] or newer.

The proposal for loading is influenced by the Java APIs for XML Processing [*JAXP*] and by SAX2 [*SAX*].

1.1 Overview of the Interfaces

The list of interfaces involved with the Loading and Saving of XML documents is:

- `DOMImplementationLS` [p.12] -- An extended `DOMImplementation` interface that provides the factory methods for creating the objects required for loading and saving.
- `LSParser` [p.14] -- An interface for parsing data into DOM documents.
- `LSInput` [p.22] -- Encapsulates information about the data to be loaded.
- `LSResourceResolver` [p.24] -- Provides a way for applications to redirect references to external resources when parsing.
- `LSParserFilter` [p.25] -- Provides the ability to examine and optionally remove nodes as they are being processed while parsing.
- `LSSerializer` [p.29] -- An interface for serializing DOM documents or nodes.
- `LSOutput` [p.36] -- Encapsulates information about the destination for the data to be output.
- `LSSerializerFilter` [p.37] -- Provides the ability to examine and filter DOM nodes as they are being processed for the serialization.

1.2 Basic types

To ensure interoperability, this specification specifies the following basic types used in various DOM modules. Even though the DOM uses the basic types in the interfaces, bindings may use different types and normative bindings are only given for Java and ECMAScript in this specification.

1.2.1 The `LSInputStream` type

This type is used to represent a sequence of input bytes.

Type Definition *LSInputStream*

A `LSInputStream` [p.9] represents a reference to a byte stream source of an XML input.
IDL Definition

```
typedef Object LSInputStream;
```

Note: For Java, `LSInputStream` [p.9] is bound to the `java.io.InputStream` type. For ECMAScript, `LSInputStream` is bound to `Object`.

1.2.2 The `LSOutputStream` type

This type is used to represent a sequence of output bytes.

Type Definition *LSOutputStream*

A `LSOutputStream` [p.10] represents a byte stream destination for the XML output.
IDL Definition

```
typedef Object LSOutputStream;
```

Note: For Java, `LSOutputStream` [p.10] is bound to the `java.io.OutputStream` type. For ECMAScript, `LSOutputStream` is bound to `Object`.

1.2.3 The `LSReader` type

This type is used to represent a sequence of input characters in 16-bit units [p.55] . The encoding used for the characters is UTF-16, as defined in [*Unicode*] and in [*ISO/IEC 10646*]).

Type Definition *LSReader*

A `LSReader` [p.10] represents a character stream for the XML input.
IDL Definition

```
typedef Object LSReader;
```

Note: For Java, `LSReader` [p.10] is bound to the `java.io.Reader` type. For ECMAScript, `LSReader` is *NOT* bound, and therefore as no recommended meaning in ECMAScript.

1.2.4 The `LSWriter` type

This type is used to represent a sequence of output characters in 16-bit units [p.55] . The encoding used for the characters is UTF-16, as defined in [*Unicode*] and in [*ISO/IEC 10646*]).

Type Definition *LSWriter*

A `LSWriter` [p.10] represents a character stream for the XML output.
IDL Definition

```
typedef Object LSWriter;
```

Note: For Java, `LSWriter` [p.10] is bound to the `java.io.Writer` type. For ECMAScript, `LSWriter` is *NOT* bound, and therefore has no recommended meaning in ECMAScript.

1.3 Fundamental interfaces

The interface within this section is considered fundamental, and must be fully implemented by all conforming implementations of the DOM Load and Save module.

A DOM application may use the `hasFeature(feature, version)` method of the `DOMImplementation` interface with parameter values `"LS"` (or `"LS-Async"`) and `"3.0"` (respectively) to determine whether or not these interfaces are supported by the implementation. In order to fully support them, an implementation must also support the "Core" feature defined in [*DOM Level 2 Core*].

A DOM application may use the `hasFeature(feature, version)` method of the `DOMImplementation` interface with parameter values `"LS-Async"` and `"3.0"` (respectively) to determine whether or not the asynchronous mode is supported by the implementation. In order to fully support the asynchronous mode, an implementation must also support the `"LS"` feature defined in this section.

For additional information about *conformance*, please see the DOM Level 3 Core specification [*DOM Level 3 Core*].

Exception *LSException*

Parser or write operations may throw an `LSException` [p.11] if the processing is stopped. The processing can be stopped due to a `DOMError` with a severity of `DOMError.SEVERITY_FATAL_ERROR` or a non recovered `DOMError.SEVERITY_ERROR`, or if `DOMErrorHandler.handleError()` returned `false`.

Note: As suggested in the definition of the constants in the `DOMError` interface, a DOM implementation may choose to continue after a fatal error, but the resulting DOM tree is then implementation dependent.

IDL Definition

```
exception LSException {
  unsigned short   code;
};
// LSExceptionCode
const unsigned short      PARSE_ERR                      = 81;
const unsigned short      SERIALIZE_ERR                  = 82;
```

Definition group *LSExceptionCode*

An integer indicating the type of error generated.

Defined Constants

`PARSE_ERR`

If an attempt was made to load a document, or an XML Fragment, using `LSParser` [p.14] and the processing has been stopped.

`SERIALIZE_ERR`

If an attempt was made to serialize a `Node` using `LSSerializer` [p.29] and the processing has been stopped.

Interface *DOMImplementationLS*

`DOMImplementationLS` contains the factory methods for creating Load and Save objects.

The expectation is that an instance of the `DOMImplementationLS` interface can be obtained by using binding-specific casting methods on an instance of the `DOMImplementation` interface or, if the `Document` supports the feature `"Core"` version `"3.0"` defined in [*DOM Level 3 Core*], by using the method `DOMImplementation.getFeature` with parameter values `"LS"` (or `"LS-Async"`) and `"3.0"` (respectively).

IDL Definition

```
interface DOMImplementationLS {

  // DOMImplementationLSMode
  const unsigned short      MODE_SYNCHRONOUS               = 1;
  const unsigned short      MODE_ASYNCHRONOUS              = 2;

  LSParser           createLSParser(in unsigned short mode,
                                    in DOMString schemaType)
                                        raises(DOMException);
  LSSerializer       createLSSerializer();
  LSInput            createLSInput();
  LSOutput           createLSOutput();
};
```

Definition group *DOMImplementationLSMode*

Integer parser mode constants.

Defined Constants

`MODE_ASYNCHRONOUS`

Create an asynchronous `LSParser` [p.14] .

`MODE_SYNCHRONOUS`

Create a synchronous `LSParser` [p.14] .

Methods

`createLSInput`

Create a new empty input source object where `LSInput.characterStream` [p.23] , `LSInput.byteStream` [p.23] , `LSInput.stringData` [p.24] `LSInput.systemId` [p.24] , `LSInput.publicId` [p.24] , `LSInput.baseURI` [p.23] , and `LSInput.encoding` [p.23] are null, and `LSInput.certifiedText` [p.23] is false.

Return Value

`LSInput` [p.22] The newly created input object.

No Parameters

No Exceptions

`createLSOutput`

Create a new empty output destination object where `LSOutput.characterStream` [p.36] , `LSOutput.byteStream` [p.36] , `LSOutput.systemId` [p.37] , `LSOutput.encoding` [p.37] are null.

Return Value

`LSOutput` [p.36] The newly created output object.

No Parameters

No Exceptions

`createLSParser`

Create a new `LSParser` [p.14] . The newly constructed parser may then be configured by means of its `DOMConfiguration` object, and used to parse documents by means of its `parse` method.

Parameters

`mode` of type `unsigned short`

The `mode` argument is either `MODE_SYNCHRONOUS` or `MODE_ASYNCHRONOUS`, if `mode` is `MODE_SYNCHRONOUS` then the `LSParser` [p.14] that is created will operate in synchronous mode, if it's `MODE_ASYNCHRONOUS` then the `LSParser` that is created will operate in asynchronous mode.

`schemaType` of type `DOMString`

An absolute URI representing the type of the schema [p.55] language used during the load of a `Document` using the newly created `LSParser` [p.14] . Note that no lexical checking is done on the absolute URI. In order to create a `LSParser` for any kind of schema types (i.e. the LSParser will be free to use any schema found), use the value `null`.

Note: For W3C XML Schema [*XML Schema Part 1*], applications must use the value `"http://www.w3.org/2001/XMLSchema"`. For XML DTD [*XML 1.0*], applications must use the value `"http://www.w3.org/TR/REC-xml"`. Other Schema languages are outside the scope of the W3C and therefore should recommend an absolute URI in order to use this method.

Return Value

`LSParser` [p.14]	The newly created `LSParser` object. This `LSParser` is either synchronous or asynchronous depending on the value of the `mode` argument. **Note:** By default, the newly created `LSParser` does not contain a `DOMErrorHandler`, i.e. the value of the "*error-handler*" configuration parameter is `null`. However, implementations may provide a default error handler at creation time. In that case, the initial value of the `"error-handler"` configuration parameter on the new `LSParser` object contains a reference to the default error handler.

Exceptions

`DOMException`	NOT_SUPPORTED_ERR: Raised if the requested mode or schema type is not supported.

`createLSSerializer`

Create a new `LSSerializer` [p.29] object.

Return Value

`LSSerializer` [p.29]	The newly created `LSSerializer` object. **Note:** By default, the newly created `LSSerializer` has no `DOMErrorHandler`, i.e. the value of the `"error-handler"` configuration parameter is `null`. However, implementations may provide a default error handler at creation time. In that case, the initial value of the `"error-handler"` configuration parameter on the new `LSSerializer` object contains a reference to the default error handler.

No Parameters

No Exceptions

Interface *LSParser*

An interface to an object that is able to build, or augment, a DOM tree from various input sources.

`LSParser` provides an API for parsing XML and building the corresponding DOM document structure. A `LSParser` instance can be obtained by invoking the `DOMImplementationLS.createLSParser()` [p.13] method.

As specified in [*DOM Level 3 Core*], when a document is first made available via the LSParser:

- there will never be two adjacent nodes of type NODE_TEXT, and there will never be empty text nodes.
- it is expected that the `value` and `nodeValue` attributes of an `Attr` node initially return the *XML 1.0 normalized value*. However, if the parameters "*validate-if-schema*" and

"*datatype-normalization*" are set to `true`, depending on the attribute normalization used, the attribute values may differ from the ones obtained by the XML 1.0 attribute normalization. If the parameters "*datatype-normalization*" is set to `false`, the XML 1.0 attribute normalization is guaranteed to occur, and if the attributes list does not contain namespace declarations, the `attributes` attribute on `Element` node represents the property **[attributes]** defined in [*XML Information Set*].

Asynchronous `LSParser` objects are expected to also implement the `events::EventTarget` interface so that event listeners can be registered on asynchronous `LSParser` objects.

Events supported by asynchronous `LSParser` objects are:

load
: The `LSParser` finishes to load the document. See also the definition of the `LSLoadEvent` [p.29] interface.

progress
: The `LSParser` signals progress as data is parsed.
This specification does not attempt to define exactly when progress events should be dispatched, that is intentionally left as implementation dependent, but here is one example of how an application might dispatch progress events. Once the parser starts receiving data, a progress event is dispatched to indicate that the parsing starts, then from there on, a progress event is dispatched for every 4096 bytes of data that is received and processed. This is only one example, though, and implementations can choose to dispatch progress events at any time while parsing, or not dispatch them at all.
See also the definition of the `LSProgressEvent` [p.28] interface.

Note: All events defined in this specification use the namespace URI `"http://www.w3.org/2002/DOMLS"`.

While parsing an input source, errors are reported to the application through the error handler (`LSParser.domConfig` [p.17] 's "*error-handler*" parameter). This specification does in no way try to define all possible errors that can occur while parsing XML, or any other markup, but some common error cases are defined. The types (`DOMError.type`) of errors and warnings defined by this specification are:

`"check-character-normalization-failure" [error]`
: Raised if the paramter "*check-character-normalization*" is set to true and a string is encountered that fails normalization checking.

`"doctype-not-allowed" [fatal]`
: Raised if the configuration parameter "disallow-doctype [p.18] " is set to `true` and a doctype is encountered.

`"no-input-specified" [fatal]`
: Raised when loading a document and no input is specified in the `LSInput` [p.22] object.

`"pi-base-uri-not-preserved" [warning]`
: Raised if a processing instruction is encountered in a location where the base URI of the processing instruction can not be preserved.
One example of a case where this warning will be raised is if the configuration parameter "*entities*" is set to `false` and the following XML file is parsed:

```
<!DOCTYPE root [
<!ENTITY e SYSTEM 'subdir/myentity.ent'
]>

<root>
&e;
</root>
```

And `subdir/myentity.ent` contains:

```
<one>
  <two/>
</one>
<?pi 3.14159?>
<more/>
```

`"unbound-prefix-in-entity" [warning]`
: An implementation dependent warning that may be raised if the configuration parameter "*namespaces*" is set to `true` and an unbound namespace prefix is encountered in an entity's replacement text. Raising this warning is not enforced since some existing parsers may not recognize unbound namespace prefixes in the replacement text of entities.

`"unknown-character-denormalization" [fatal]`
: Raised if the configuration parameter "ignore-unknown-character-denormalizations [p.18] " is set to `false` and a character is encountered for which the processor cannot determine the normalization properties.

`"unsupported-encoding" [fatal]`
: Raised if an unsupported encoding is encountered.

`"unsupported-media-type" [fatal]`
: Raised if the configuration parameter "supported-media-types-only [p.19] " is set to `true` and an unsupported media type is encountered.

In addition to raising the defined errors and warnings, implementations are expected to raise implementation specific errors and warnings for any other error and warning cases such as IO errors (file not found, permission denied,...), XML well-formedness errors, and so on.

IDL Definition

```
interface LSParser {
  readonly attribute DOMConfiguration domConfig;
           attribute LSParserFilter  filter;
  readonly attribute boolean         async;
  readonly attribute boolean         busy;
  Document           parse(in LSInput input)
                                     raises(DOMException,
                                            LSException);
  Document           parseURI(in DOMString uri)
                                     raises(DOMException,
                                            LSException);

  // ACTION_TYPES
  const unsigned short      ACTION_APPEND_AS_CHILDREN      = 1;
  const unsigned short      ACTION_REPLACE_CHILDREN        = 2;
  const unsigned short      ACTION_INSERT_BEFORE           = 3;
  const unsigned short      ACTION_INSERT_AFTER            = 4;
```

```
  const unsigned short      ACTION_REPLACE                = 5;

  Node                parseWithContext(in LSInput input,
                                       in Node contextArg,
                                       in unsigned short action)
                                         raises(DOMException,
                                                LSException);
  void                abort();
};
```

Definition group *ACTION_TYPES*

A set of possible actions for the `parseWithContext` method.

Defined Constants

`ACTION_APPEND_AS_CHILDREN`
: Append the result of the parse operation as children of the context node. For this action to work, the context node must be an `Element` or a `DocumentFragment`.

`ACTION_INSERT_AFTER`
: Insert the result of the parse operation as the immediately following sibling of the context node. For this action to work the context node's parent must be an `Element` or a `DocumentFragment`.

`ACTION_INSERT_BEFORE`
: Insert the result of the parse operation as the immediately preceding sibling of the context node. For this action to work the context node's parent must be an `Element` or a `DocumentFragment`.

`ACTION_REPLACE`
: Replace the context node with the result of the parse operation. For this action to work, the context node must have a parent, and the parent must be an `Element` or a `DocumentFragment`.

`ACTION_REPLACE_CHILDREN`
: Replace all the children of the context node with the result of the parse operation. For this action to work, the context node must be an `Element`, a `Document`, or a `DocumentFragment`.

Attributes

`async` of type `boolean`, readonly
: `true` if the `LSParser` is asynchronous, `false` if it is synchronous.

`busy` of type `boolean`, readonly
: `true` if the `LSParser` is currently busy loading a document, otherwise `false`.

`domConfig` of type `DOMConfiguration`, readonly
: The `DOMConfiguration` object used when parsing an input source. This `DOMConfiguration` is specific to the parse operation and no parameter values from this `DOMConfiguration` object are passed automatically to the `DOMConfiguration` object on the `Document` that is created, or used, by the parse operation. The DOM application is responsible for passing any needed parameter values from this `DOMConfiguration` object to the `DOMConfiguration` object referenced by the `Document` object.
  In addition to the parameters recognized in on the *DOMConfiguration* interface defined in [*DOM Level 3 Core*], the `DOMConfiguration` objects for `LSParser` add or modify the following parameters:

`"charset-overrides-xml-encoding"`
: `true`
  : [*optional*] (*default*)
    If a higher level protocol such as HTTP [*IETF RFC 2616*] provides an indication of the character encoding of the input stream being processed, that will override any encoding specified in the XML declaration or the Text declaration (see also section 4.3.3, "Character Encoding in Entities", in [*XML 1.0*]). Explicitly setting an encoding in the `LSInput` [p.22] overrides any encoding from the protocol.
: `false`
  : [*required*]
    The parser ignores any character set encoding information from higher-level protocols.

`"disallow-doctype"`
: `true`
  : [*optional*]
    Throw a fatal **"doctype-not-allowed"** error if a doctype node is found while parsing the document. This is useful when dealing with things like SOAP envelopes where doctype nodes are not allowed.
: `false`
  : [*required*] (*default*)
    Allow doctype nodes in the document.

`"ignore-unknown-character-denormalizations"`
: `true`
  : [*required*] (*default*)
    If, while verifying full normalization when [*XML 1.1*] is supported, a processor encounters characters for which it cannot determine the normalization properties, then the processor will ignore any possible denormalizations caused by these characters.
    This parameter is ignored for [*XML 1.0*].
: `false`
  : [*optional*]
    Report an fatal **"unknown-character-denormalization"** error if a character is encountered for which the processor cannot determine the normalization properties.

`"infoset"`
: See the definition of `DOMConfiguration` for a description of this parameter.
  Unlike in [*DOM Level 3 Core*], this parameter will default to `true` for `LSParser`.

`"namespaces"`
: `true`
  : [*required*] (*default*)
    Perform the namespace processing as defined in [*XML Namespaces*] and [*XML Namespaces 1.1*].
: `false`
  : [*optional*]
    Do not perform the namespace processing.

`"resource-resolver"`
[*required*]
A reference to a `LSResourceResolver` [p.24] object, or null. If the value of this parameter is not null when an external resource (such as an external XML entity or an XML schema location) is encountered, the implementation will request that the `LSResourceResolver` referenced in this parameter resolves the resource.
`"supported-media-types-only"`
`true`
[*optional*]
Check that the media type of the parsed resource is a supported media type. If an unsupported media type is encountered, a fatal error of type **"unsupported-media-type"** will be raised. The media types defined in [*IETF RFC 3023*] must always be accepted.
`false`
[*required*] (*default*)
Accept any media type.
The parameter "*well-formed*" cannot be set to `false`.
`filter` of type `LSParserFilter` [p.25]
When a filter is provided, the implementation will call out to the filter as it is constructing the DOM tree structure. The filter can choose to remove elements from the document being constructed, or to terminate the parsing early.
The filter is invoked after the operations requested by the `DOMConfiguration` parameters have been applied. For example, if "*validate*" is set to `true`, the validation is done before invoking the filter.

Methods

`abort`
Abort the loading of the document that is currently being loaded by the `LSParser`. If the `LSParser` is currently not busy, a call to this method does nothing.
No Parameters
No Return Value
No Exceptions
`parse`
Parse an XML document from a resource identified by a `LSInput` [p.22] .
Parameters
`input` of type `LSInput` [p.22]
The `LSInput` from which the source of the document is to be read.
Return Value

`Document`	If the `LSParser` is a synchronous `LSParser`, the newly created and populated `Document` is returned. If the `LSParser` is asynchronous, `null` is returned since the document object may not yet be constructed when this method returns.

Exceptions

`DOMException`	INVALID_STATE_ERR: Raised if the `LSParser`'s `LSParser.busy` [p.17] attribute is `true`.
`LSException` [p.11]	PARSE_ERR: Raised if the `LSParser` was unable to load the XML document. DOM applications should attach a `DOMErrorHandler` using the parameter "*error-handler*" if they wish to get details on the error.

`parseURI`

Parse an XML document from a location identified by a URI reference [*IETF RFC 2396*]. If the URI contains a fragment identifier (see section 4.1 in [*IETF RFC 2396*]), the behavior is not defined by this specification, future versions of this specification may define the behavior.

Parameters

`uri` of type `DOMString`

The location of the XML document to be read.

Return Value

`Document`	If the `LSParser` is a synchronous `LSParser`, the newly created and populated `Document` is returned, or `null` if an error occured. If the `LSParser` is asynchronous, `null` is returned since the document object may not yet be constructed when this method returns.

Exceptions

`DOMException`	INVALID_STATE_ERR: Raised if the `LSParser.busy` [p.17] attribute is `true`.
`LSException` [p.11]	PARSE_ERR: Raised if the `LSParser` was unable to load the XML document. DOM applications should attach a `DOMErrorHandler` using the parameter "*error-handler*" if they wish to get details on the error.

`parseWithContext`

Parse an XML fragment from a resource identified by a `LSInput` [p.22] and insert the content into an existing document at the position specified with the `context` and `action` arguments. When parsing the input stream, the context node (or its parent, depending on where the result will be inserted) is used for resolving unbound namespace prefixes. The context node's `ownerDocument` node (or the node itself if the node of type `DOCUMENT_NODE`) is used to resolve default attributes and entity references.

As the new data is inserted into the document, at least one mutation event is fired per new immediate child or sibling of the context node.

If the context node is a `Document` node and the action is `ACTION_REPLACE_CHILDREN`, then the document that is passed as the context node will be changed such that its `xmlEncoding`, `documentURI`, `xmlVersion`, `inputEncoding`, `xmlStandalone`, and all other such attributes are set to what they

would be set to if the input source was parsed using `LSParser.parse()` [p.19] . This method is always synchronous, even if the `LSParser` is asynchronous (`LSParser.async` [p.17] is `true`).
If an error occurs while parsing, the caller is notified through the `ErrorHandler` instance associated with the "*error-handler*" parameter of the `DOMConfiguration`.
When calling `parseWithContext`, the values of the following configuration parameters will be ignored and their default values will always be used instead: "*validate*", "*validate-if-schema*", and "*element-content-whitespace*". Other parameters will be treated normally, and the parser is expected to call the `LSParserFilter` [p.25] just as if a whole document was parsed.

Parameters

`input` of type `LSInput` [p.22]
: The `LSInput` from which the source document is to be read. The source document must be an XML fragment, i.e. anything except a complete XML document (except in the case where the context node of type `DOCUMENT_NODE`, and the action is `ACTION_REPLACE_CHILDREN`), a DOCTYPE (internal subset), entity declaration(s), notation declaration(s), or XML or text declaration(s).

`contextArg` of type `Node`
: The node that is used as the context for the data that is being parsed. This node must be a `Document` node, a `DocumentFragment` node, or a node of a type that is allowed as a child of an `Element` node, e.g. it cannot be an `Attribute` node.

`action` of type `unsigned short`
: This parameter describes which action should be taken between the new set of nodes being inserted and the existing children of the context node. The set of possible actions is defined in `ACTION_TYPES` above.

Return Value

`Node` Return the node that is the result of the parse operation. If the result is more than one top-level node, the first one is returned.

Exceptions

`DOMException`	HIERARCHY_REQUEST_ERR: Raised if the content cannot replace, be inserted before, after, or as a child of the context node (see also `Node.insertBefore` or `Node.replaceChild` in [*DOM Level 3 Core*]). NOT_SUPPORTED_ERR: Raised if the `LSParser` doesn't support this method, or if the context node is of type `Document` and the DOM implementation doesn't support the replacement of the `DocumentType` child or `Element` child. NO_MODIFICATION_ALLOWED_ERR: Raised if the context node is a read only node [p.55] and the content is being appended to its child list, or if the parent node of the context node is read only node [p.55] and the content is being inserted in its child list. INVALID_STATE_ERR: Raised if the `LSParser.busy` [p.17] attribute is `true`.
`LSException` [p.11]	PARSE_ERR: Raised if the `LSParser` was unable to load the XML fragment. DOM applications should attach a `DOMErrorHandler` using the parameter "*error-handler*" if they wish to get details on the error.

Interface *LSInput*

This interface represents an input source for data.

This interface allows an application to encapsulate information about an input source in a single object, which may include a public identifier, a system identifier, a byte stream (possibly with a specified encoding), a base URI, and/or a character stream.

The exact definitions of a byte stream and a character stream are binding dependent.

The application is expected to provide objects that implement this interface whenever such objects are needed. The application can either provide its own objects that implement this interface, or it can use the generic factory method `DOMImplementationLS.createLSInput()` [p.12] to create objects that implement this interface.

The `LSParser` [p.14] will use the `LSInput` object to determine how to read data. The `LSParser` will look at the different inputs specified in the `LSInput` in the following order to know which one to read from, the first one that is not null and not an empty string will be used:

1. `LSInput.characterStream` [p.23]
2. `LSInput.byteStream` [p.23]
3. `LSInput.stringData` [p.24]
4. `LSInput.systemId` [p.24]
5. `LSInput.publicId` [p.24]

If all inputs are null, the `LSParser` [p.14] will report a `DOMError` with its `DOMError.type` set to `"no-input-specified"` and its `DOMError.severity` set to `DOMError.SEVERITY_FATAL_ERROR`.

`LSInput` objects belong to the application. The DOM implementation will never modify them (though it may make copies and modify the copies, if necessary).

IDL Definition

```
interface LSInput {
  // Depending on the language binding in use,
  // this attribute may not be available.
           attribute LSReader        characterStream;
           attribute LSInputStream   byteStream;
           attribute DOMString       stringData;
           attribute DOMString       systemId;
           attribute DOMString       publicId;
           attribute DOMString       baseURI;
           attribute DOMString       encoding;
           attribute boolean         certifiedText;
};
```

Attributes

`baseURI` of type `DOMString`

The base URI to be used (see section 5.1.4 in [*IETF RFC 2396*]) for resolving a relative `systemId` to an absolute URI.

If, when used, the base URI is itself a relative URI, an empty string, or null, the behavior is implementation dependent.

`byteStream` of type `LSInputStream` [p.9]

An attribute of a language and binding dependent type that represents a stream of bytes.

If the application knows the character encoding of the byte stream, it should set the encoding attribute. Setting the encoding in this way will override any encoding specified in an XML declaration in the data.

`certifiedText` of type `boolean`

If set to true, assume that the input is certified (see section 2.13 in [*XML 1.1*]) when parsing [*XML 1.1*].

`characterStream` of type `LSReader` [p.10]

Depending on the language binding in use, this attribute may not be available.

An attribute of a language and binding dependent type that represents a stream of 16-bit units [p.55] . The application must encode the stream using UTF-16 (defined in [*Unicode*] and in [*ISO/IEC 10646*]).

`encoding` of type `DOMString`

The character encoding, if known. The encoding must be a string acceptable for an XML encoding declaration ([*XML 1.0*] section 4.3.3 "Character Encoding in Entities").

This attribute has no effect when the application provides a character stream or string data. For other sources of input, an encoding specified by means of this attribute will override any encoding specified in the XML declaration or the Text declaration, or an encoding obtained from a higher level protocol, such as HTTP [*IETF RFC 2616*].

`publicId` of type `DOMString`
: The public identifier for this input source. This may be mapped to an input source using an implementation dependent mechanism (such as catalogues or other mappings). The public identifier, if specified, may also be reported as part of the location information when errors are reported.

`stringData` of type `DOMString`
: String data to parse. If provided, this will always be treated as a sequence of 16-bit units [p.55] (UTF-16 encoded characters).

`systemId` of type `DOMString`
: The system identifier, a URI reference [*IETF RFC 2396*], for this input source. The system identifier is optional if there is a byte stream, a character stream, or string data, but it is still useful to provide one, since the application will use it to resolve any relative URIs and can include it in error messages and warnings (the `LSParser` [p.14] will only attempt to fetch the resource identified by the URI reference if there is no other input available in the input source).
: If the application knows the character encoding of the object pointed to by the system identifier, it can set the encoding using the `encoding` attribute.
: If the specified system ID is a relative URI reference (see section 5 in [*IETF RFC 2396*]), the DOM implementation will attempt to resolve the relative URI with the `baseURI` as the base, if that fails, the behavior is implementation dependent.

Interface *LSResourceResolver*

`LSResourceResolver` provides a way for applications to redirect references to external resources.

Applications needing to implement custom handling for external resources can implement this interface and register their implementation by setting the "resource-resolver" parameter of `DOMConfiguration` objects attached to `LSParser` [p.14] and `LSSerializer` [p.29] . It can also be register on `DOMConfiguration` objects attached to `Document` if the "LS" feature is supported.

The `LSParser` [p.14] will then allow the application to intercept any external entities, including the external DTD subset and external parameter entities, before including them. The top-level document entity is never passed to the `resolveResource` method.

Many DOM applications will not need to implement this interface, but it will be especially useful for applications that build XML documents from databases or other specialized input sources, or for applications that use URN's.

Note: `LSResourceResolver` is based on the SAX2 [*SAX*] `EntityResolver` interface.

IDL Definition

```
interface LSResourceResolver {
  LSInput            resolveResource(in DOMString type,
                                     in DOMString namespaceURI,
                                     in DOMString publicId,
                                     in DOMString systemId,
                                     in DOMString baseURI);
};
```

Methods

`resolveResource`

Allow the application to resolve external resources.

The `LSParser` [p.14] will call this method before opening any external resource, including the external DTD subset, external entities referenced within the DTD, and external entities referenced within the document element (however, the top-level document entity is not passed to this method). The application may then request that the `LSParser` resolve the external resource itself, that it use an alternative URI, or that it use an entirely different input source.

Application writers can use this method to redirect external system identifiers to secure and/or local URI, to look up public identifiers in a catalogue, or to read an entity from a database or other input source (including, for example, a dialog box).

Parameters

`type` of type `DOMString`

The type of the resource being resolved. For XML [*XML 1.0*] resources (i.e. entities), applications must use the value `"http://www.w3.org/TR/REC-xml"`, for XML Schema [*XML Schema Part 1*], applications must use the value `"http://www.w3.org/2001/XMLSchema"`. Other types of resources are outside the scope of this specification and therefore should recommend an absolute URI in order to use this method.

`namespaceURI` of type `DOMString`

The namespace of the resource being resolved, e.g. the target namespace of the XML Schema [*XML Schema Part 1*] when resolving XML Schema resources.

`publicId` of type `DOMString`

The public identifier of the external entity being referenced, or `null` if no public identifier was supplied or if the resource is not an entity.

`systemId` of type `DOMString`

The system identifier, a URI reference [*IETF RFC 2396*], of the external resource being referenced, or `null` if no system identifier was supplied.

`baseURI` of type `DOMString`

The absolute base URI of the resource being parsed, or `null` if there is no base URI.

Return Value

`LSInput` [p.22]	A `LSInput` object describing the new input source, or `null` to request that the parser open a regular URI connection to the resource.

No Exceptions

Interface *LSParserFilter*

`LSParserFilters` provide applications the ability to examine nodes as they are being constructed while parsing. As each node is examined, it may be modified or removed, or the entire parse may be terminated early.

At the time any of the filter methods are called by the parser, the owner Document and DOMImplementation objects exist and are accessible. The document element is never passed to the `LSParserFilter` methods, i.e. it is not possible to filter out the document element. `Document`, `DocumentType`, `Notation`, `Entity`, and `Attr` nodes are never passed to the `acceptNode` method on the filter. The child nodes of an `EntityReference` node are passed to the filter if the parameter "*entities*" is set to `false`. Note that, as described by the parameter "*entities*", entity reference nodes to non-defined entities are never discarded and are always passed to the filter.

All validity checking while parsing a document occurs on the source document as it appears on the input stream, not on the DOM document as it is built in memory. With filters, the document in memory may be a subset of the document on the stream, and its validity may have been affected by the filtering.

All default attributes must be present on elements when the elements are passed to the filter methods. All other default content must be passed to the filter methods.

DOM applications must not raise exceptions in a filter. The effect of throwing exceptions from a filter is DOM implementation dependent.

IDL Definition

```
interface LSParserFilter {

  // Constants returned by startElement and acceptNode
  const short               FILTER_ACCEPT                 = 1;
  const short               FILTER_REJECT                 = 2;
  const short               FILTER_SKIP                   = 3;
  const short               FILTER_INTERRUPT              = 4;

  unsigned short     startElement(in Element elementArg);
  unsigned short     acceptNode(in Node nodeArg);
  readonly attribute unsigned long   whatToShow;
};
```

Definition group *Constants returned by startElement and acceptNode*

Constants returned by `startElement` and `acceptNode`.

Defined Constants

`FILTER_ACCEPT`
: Accept the node.

`FILTER_INTERRUPT`
: Interrupt the normal processing of the document.

`FILTER_REJECT`
: Reject the node and its children.

`FILTER_SKIP`
: Skip this single node. The children of this node will still be considered.

Attributes

`whatToShow` of type `unsigned long`, readonly

Tells the `LSParser` [p.14] what types of nodes to show to the method `LSParserFilter.acceptNode` [p.27] . If a node is not shown to the filter using this attribute, it is automatically included in the DOM document being built. See `NodeFilter` for definition of the constants. The constants `SHOW_ATTRIBUTE`, `SHOW_DOCUMENT`, `SHOW_DOCUMENT_TYPE`, `SHOW_NOTATION`, `SHOW_ENTITY`, and `SHOW_DOCUMENT_FRAGMENT` are meaningless here, those nodes will never be passed to `LSParserFilter.acceptNode`.

The constants used here are defined in [*DOM Level 2 Traversal and Range*].

Methods

`acceptNode`

This method will be called by the parser at the completion of the parsing of each node. The node and all of its descendants will exist and be complete. The parent node will also exist, although it may be incomplete, i.e. it may have additional children that have not yet been parsed. Attribute nodes are never passed to this function.

From within this method, the new node may be freely modified - children may be added or removed, text nodes modified, etc. The state of the rest of the document outside this node is not defined, and the affect of any attempt to navigate to, or to modify any other part of the document is undefined.

For validating parsers, the checks are made on the original document, before any modification by the filter. No validity checks are made on any document modifications made by the filter.

If this new node is rejected, the parser might reuse the new node and any of its descendants.

Parameters

`nodeArg` of type `Node`

The newly constructed element. At the time this method is called, the element is complete - it has all of its children (and their children, recursively) and attributes, and is attached as a child to its parent.

Return Value

`unsigned short`

- `FILTER_ACCEPT` if this `Node` should be included in the DOM document being built.
- `FILTER_REJECT` if the `Node` and all of its children should be rejected.
- `FILTER_SKIP` if the `Node` should be skipped and the `Node` should be replaced by all the children of the `Node`.
- `FILTER_INTERRUPT` if the filter wants to stop the processing of the document. Interrupting the processing of the document does no longer guarantee that the resulting DOM tree is XML well-formed [p.55] . The `Node` is accepted and will be the last completely parsed node.

No Exceptions

`startElement`

The parser will call this method after each `Element` start tag has been scanned, but before the remainder of the `Element` is processed. The intent is to allow the element, including any children, to be efficiently skipped. Note that only element nodes are passed to the `startElement` function.

The element node passed to `startElement` for filtering will include all of the Element's attributes, but none of the children nodes. The Element may not yet be in place in the document being constructed (it may not have a parent node.)

A `startElement` filter function may access or change the attributes for the Element. Changing Namespace declarations will have no effect on namespace resolution by the parser.

For efficiency, the Element node passed to the filter may not be the same one as is actually placed in the tree if the node is accepted. And the actual node (node object identity) may be reused during the process of reading in and filtering a document.

Parameters

`elementArg` of type `Element`

The newly encountered element. At the time this method is called, the element is incomplete - it will have its attributes, but no children.

Return Value

`unsigned short`

- `FILTER_ACCEPT` if the `Element` should be included in the DOM document being built.
- `FILTER_REJECT` if the `Element` and all of its children should be rejected.
- `FILTER_SKIP` if the `Element` should be skipped. All of its children are inserted in place of the skipped `Element` node.
- `FILTER_INTERRUPT` if the filter wants to stop the processing of the document. Interrupting the processing of the document does no longer guarantee that the resulting DOM tree is XML well-formed [p.55] . The `Element` is rejected.

Returning any other values will result in unspecified behavior.

No Exceptions

Interface *LSProgressEvent*

This interface represents a progress event object that notifies the application about progress as a document is parsed. It extends the `Event` interface defined in [*DOM Level 3 Events*].

The units used for the attributes `position` and `totalSize` are not specified and can be implementation and input dependent.

IDL Definition

```
interface LSProgressEvent : events::Event {
  readonly attribute LSInput         input;
  readonly attribute unsigned long   position;
  readonly attribute unsigned long   totalSize;
};
```

Attributes

`input` of type `LSInput` [p.22] , readonly
: The input source that is being parsed.

`position` of type `unsigned long`, readonly
: The current position in the input source, including all external entities and other resources that have been read.

`totalSize` of type `unsigned long`, readonly
: The total size of the document including all external resources, this number might change as a document is being parsed if references to more external resources are seen. A value of `0` is returned if the total size cannot be determined or estimated.

Interface *LSLoadEvent*

This interface represents a load event object that signals the completion of a document load.

IDL Definition

```
interface LSLoadEvent : events::Event {
  readonly attribute Document        newDocument;
  readonly attribute LSInput         input;
};
```

Attributes

`input` of type `LSInput` [p.22] , readonly
: The input source that was parsed.

`newDocument` of type `Document`, readonly
: The document that finished loading.

Interface *LSSerializer*

A `LSSerializer` provides an API for serializing (writing) a DOM document out into XML. The XML data is written to a string or an output stream. Any changes or fixups made during the serialization affect only the serialized data. The `Document` object and its children are never altered by the serialization operation.

During serialization of XML data, namespace fixup is done as defined in [*DOM Level 3 Core*], Appendix B. [*DOM Level 2 Core*] allows empty strings as a real namespace URI. If the `namespaceURI` of a `Node` is empty string, the serialization will treat them as `null`, ignoring the prefix if any.

`LSSerializer` accepts any node type for serialization. For nodes of type `Document` or `Entity`, well-formed XML will be created when possible (well-formedness is guaranteed if the document or entity comes from a parse operation and is unchanged since it was created). The serialized output for these node types is either as a XML document or an External XML Entity, respectively, and is acceptable input for an XML parser. For all other types of nodes the serialized form is implementation dependent.

Within a `Document`, `DocumentFragment`, or `Entity` being serialized, `Nodes` are processed as follows

- `Document` nodes are written, including the XML declaration (unless the parameter "xml-declaration [p.33] " is set to `false`) and a DTD subset, if one exists in the DOM. Writing a `Document` node serializes the entire document.
- `Entity` nodes, when written directly by `LSSerializer.write` [p.34] , outputs the entity expansion but no namespace fixup is done. The resulting output will be valid as an external entity.
- If the parameter "*entities*" is set to `true`, `EntityReference` nodes are serialized as an entity reference of the form "`&entityName;`" in the output. Child nodes (the expansion) of the entity reference are ignored. If the parameter "*entities*" is set to `false`, only the children of the entity reference are serialized. `EntityReference` nodes with no children (no corresponding `Entity` node or the corresponding `Entity` nodes have no children) are always serialized.
- `CDATAsections` containing content characters that cannot be represented in the specified output encoding are handled according to the "*split-cdata-sections*" parameter.
  If the parameter is set to `true`, `CDATAsections` are split, and the unrepresentable characters are serialized as numeric character references in ordinary content. The exact position and number of splits is not specified.
  If the parameter is set to `false`, unrepresentable characters in a `CDATAsection` are reported as "`wf-invalid-character`" errors if the parameter "*well-formed*" is set to `true`. The error is not recoverable - there is no mechanism for supplying alternative characters and continuing with the serialization.
- `DocumentFragment` nodes are serialized by serializing the children of the document fragment in the order they appear in the document fragment.
- All other node types (Element, Text, etc.) are serialized to their corresponding XML source form.

Note: The serialization of a `Node` does not always generate a well-formed [p.55] XML document, i.e. a `LSParser` [p.14] might throw fatal errors when parsing the resulting serialization.

Within the character data of a document (outside of markup), any characters that cannot be represented directly are replaced with character references. Occurrences of '<' and '&' are replaced by the predefined entities < and &. The other predefined entities (>, ', and ") might not be used, except where needed (e.g. using > in cases such as ']]>'). Any characters that cannot be represented directly in the output character encoding are serialized as numeric character references (and since character encoding standards commonly use hexadecimal representations of characters, using the hexadecimal representation when serializing character references is encouraged).

To allow attribute values to contain both single and double quotes, the apostrophe or single-quote character (') may be represented as "'", and the double-quote character (") as """. New line characters and other characters that cannot be represented directly in attribute values in the output character encoding are serialized as a numeric character reference.

Within markup, but outside of attributes, any occurrence of a character that cannot be represented in the output character encoding is reported as a `DOMError` fatal error. An example would be serializing the element <LaCañada/> with `encoding="us-ascii"`. This will result with a generation of a `DOMError` "wf-invalid-character-in-node-name" (as proposed in "*well-formed*").

When requested by setting the parameter "*normalize-characters*" on `LSSerializer` to true, character normalization is performed according to the definition of fully normalized characters included in appendix E of [*XML 1.1*] on all data to be serialized, both markup and character data. The character normalization process affects only the data as it is being written; it does not alter the DOM's view of the document after serialization has completed.

When outputting unicode data, whether or not a byte order mark is serialized, or if the output is big-endian or little-endian, is implementation dependent.

Namespaces are fixed up during serialization, the serialization process will verify that namespace declarations, namespace prefixes and the namespace URI associated with elements and attributes are consistent. If inconsistencies are found, the serialized form of the document will be altered to remove them. The method used for doing the namespace fixup while serializing a document is the algorithm defined in Appendix B.1, "Namespace normalization", of [*DOM Level 3 Core*].

While serializing a document, the parameter "discard-default-content [p.32] " controls whether or not non-specified data is serialized.

While serializing, errors and warnings are reported to the application through the error handler (`LSSerializer.domConfig` [p.32] 's "*error-handler*" parameter). This specification does in no way try to define all possible errors and warnings that can occur while serializing a DOM node, but some common error and warning cases are defined. The types (`DOMError.type`) of errors and warnings defined by this specification are:

`"no-output-specified" [fatal]`
: Raised when writing to a `LSOutput` [p.36] if no output is specified in the `LSOutput`.

`"unbound-prefix-in-entity-reference" [fatal]`
: Raised if the configuration parameter "*namespaces*" is set to `true` and an entity whose replacement text contains unbound namespace prefixes is referenced in a location where there are no bindings for the namespace prefixes.

`"unsupported-encoding" [fatal]`
: Raised if an unsupported encoding is encountered.

In addition to raising the defined errors and warnings, implementations are expected to raise implementation specific errors and warnings for any other error and warning cases such as IO errors (file not found, permission denied,...) and so on.

IDL Definition

```
interface LSSerializer {
  readonly attribute DOMConfiguration domConfig;
           attribute DOMString       newLine;
           attribute LSSerializerFilter filter;
  boolean            write(in Node nodeArg,
                           in LSOutput destination)
                                       raises(LSException);
```

```
  boolean            writeToURI(in Node nodeArg,
                                in DOMString uri)
                                        raises(LSException);
  DOMString          writeToString(in Node nodeArg)
                                        raises(DOMException,
                                               LSException);
};
```

Attributes

`domConfig` of type `DOMConfiguration`, readonly

The `DOMConfiguration` object used by the `LSSerializer` when serializing a DOM node.

In addition to the parameters recognized in on the *DOMConfiguration* interface defined in [*DOM Level 3 Core*], the `DOMConfiguration` objects for `LSSerializer` adds, or modifies, the following parameters:

`"canonical-form"`

`true`

[*optional*]

Writes the document according to the rules specified in [*Canonical XML*]. In addition to the behavior described in "*canonical-form*" [*DOM Level 3 Core*], setting this parameter to `true` will set the parameters "format-pretty-print [p.32] ", "discard-default-content [p.32] ", and "xml-declaration [p.33] ", to `false`. Setting one of those parameters to `true` will set this parameter to `false`. Serializing an XML 1.1 document when "canonical-form" is `true` will generate a fatal error.

`false`

[*required*] (*default*)

Do not canonicalize the output.

`"discard-default-content"`

`true`

[*required*] (*default*)

Use the `Attr.specified` attribute to decide what attributes should be discarded. Note that some implementations might use whatever information available to the implementation (i.e. XML schema, DTD, the `Attr.specified` attribute, and so on) to determine what attributes and content to discard if this parameter is set to `true`.

`false`

[*required*]

Keep all attributes and all content.

`"format-pretty-print"`

`true`

[*optional*]

Formatting the output by adding whitespace to produce a pretty-printed, indented, human-readable form. The exact form of the transformations is not specified by this specification. Pretty-printing changes the content of the document and may affect the validity of the document, validating implementations should preserve validity.

`false`
[*required*] (*default*)
Don't pretty-print the result.

`"ignore-unknown-character-denormalizations"`
`true`
[*required*] (*default*)
If, while verifying full normalization when [*XML 1.1*] is supported, a character is encountered for which the normalization properties cannot be determined, then raise a `"unknown-character-denormalization"` warning (instead of raising an error, if this parameter is not set) and ignore any possible denormalizations caused by these characters.

`false`
[*optional*]
Report a fatal error if a character is encountered for which the processor cannot determine the normalization properties.

`"normalize-characters"`
This parameter is equivalent to the one defined by `DOMConfiguration` in [*DOM Level 3 Core*]. Unlike in the Core, the default value for this parameter is `true`. While DOM implementations are not required to support fully normalizing the characters in the document according to appendix E of [*XML 1.1*], this parameter must be activated by default if supported.

`"xml-declaration"`
`true`
[*required*] (*default*)
If a `Document`, `Element`, or `Entity` node is serialized, the XML declaration, or text declaration, should be included. The version (`Document.xmlVersion` if the document is a Level 3 document and the version is non-null, otherwise use the value "1.0"), and the output encoding (see `LSSerializer.write` [p.34] for details on how to find the output encoding) are specified in the serialized XML declaration.

`false`
[*required*]
Do not serialize the XML and text declarations. Report a `"xml-declaration-needed"` warning if this will cause problems (i.e. the serialized data is of an XML version other than [*XML 1.0*], or an encoding would be needed to be able to re-parse the serialized data).

`filter` of type `LSSerializerFilter` [p.37]
When the application provides a filter, the serializer will call out to the filter before serializing each Node. The filter implementation can choose to remove the node from the stream or to terminate the serialization early.
The filter is invoked after the operations requested by the `DOMConfiguration` parameters have been applied. For example, CDATA sections won't be passed to the filter if "*cdata-sections*" is set to `false`.

`newLine` of type `DOMString`
The end-of-line sequence of characters to be used in the XML being written out. Any string is supported, but XML treats only a certain set of characters sequence as end-of-line (See

section 2.11, "End-of-Line Handling" in [*XML 1.0*], if the serialized content is XML 1.0 or section 2.11, "End-of-Line Handling" in [*XML 1.1*], if the serialized content is XML 1.1). Using other character sequences than the recommended ones can result in a document that is either not serializable or not well-formed).
On retrieval, the default value of this attribute is the implementation specific default end-of-line sequence. DOM implementations should choose the default to match the usual convention for text files in the environment being used. Implementations must choose a default sequence that matches one of those allowed by XML 1.0 or XML 1.1, depending on the serialized content. Setting this attribute to `null` will reset its value to the default value.

Methods

`write`

Serialize the specified node as described above in the general description of the `LSSerializer` interface. The output is written to the supplied `LSOutput` [p.36] . When writing to a `LSOutput` [p.36] , the encoding is found by looking at the encoding information that is reachable through the `LSOutput` and the item to be written (or its owner document) in this order:

1. `LSOutput.encoding` [p.37] ,
2. `Document.inputEncoding`,
3. `Document.xmlEncoding`.

If no encoding is reachable through the above properties, a default encoding of "UTF-8" will be used.
If the specified encoding is not supported an "unsupported-encoding" fatal error is raised.
When outputting XML data, implementations are required to support the encodings "UTF-8", "UTF-16BE", and "UTF-16LE" to guarantee that data is serializable in all encodings that are required to be supported by all XML parsers.
If no output is specified in the `LSOutput` [p.36] , a "no-output-specified" fatal error is raised.
The implementation is responsible of associating the appropriate media type with the serialized data.
When writing to a HTTP URI, a HTTP PUT is performed. When writing to other types of URIs, the mechanism for writing the data to the URI is implementation dependent.

Parameters

`nodeArg` of type `Node`
: The node to serialize.

`destination` of type `LSOutput` [p.36]
: The destination for the serialized DOM.

Return Value

`boolean`	Returns `true` if `node` was successfully serialized. Return `false` in case the normal processing stopped but the implementation kept serializing the document; the result of the serialization being implementation dependent then.

Exceptions

`LSException` [p.11]	SERIALIZE_ERR: Raised if the `LSSerializer` was unable to serialize the node. DOM applications should attach a `DOMErrorHandler` using the parameter "*error-handler*" if they wish to get details on the error.

`writeToString`

Serialize the specified node as described above in the general description of the `LSSerializer` interface. The output is written to a `DOMString` that is returned to the caller. The encoding used is the encoding of the `DOMString` type, i.e. UTF-16.

Parameters

`nodeArg` of type `Node`

The node to serialize.

Return Value

`DOMString`	Returns the serialized data.

Exceptions

`DOMException`	DOMSTRING_SIZE_ERR: Raised if the resulting string is too long to fit in a `DOMString`.
`LSException` [p.11]	SERIALIZE_ERR: Raised if the `LSSerializer` was unable to serialize the node. DOM applications should attach a `DOMErrorHandler` using the parameter "*error-handler*" if they wish to get details on the error.

`writeToURI`

A convenience method that acts as if `LSSerializer.write` [p.34] was called with a `LSOutput` [p.36] with no encoding specified and `LSOutput.systemId` [p.37] set to the `uri` argument.

Parameters

`nodeArg` of type `Node`

The node to serialize.

`uri` of type `DOMString`

The URI to write to.

Return Value

`boolean`	Returns `true` if `node` was successfully serialized. Return `false` in case the normal processing stopped but the implementation kept serializing the document; the result of the serialization being implementation dependent then.

Exceptions

`LSException` [p.11]	SERIALIZE_ERR: Raised if the `LSSerializer` was unable to serialize the node. DOM applications should attach a `DOMErrorHandler` using the parameter "*error-handler*" if they wish to get details on the error.

Interface *LSOutput*

This interface represents an output destination for data.

This interface allows an application to encapsulate information about an output destination in a single object, which may include a URI, a byte stream (possibly with a specified encoding), a base URI, and/or a character stream.

The exact definitions of a byte stream and a character stream are binding dependent.

The application is expected to provide objects that implement this interface whenever such objects are needed. The application can either provide its own objects that implement this interface, or it can use the generic factory method `DOMImplementationLS.createLSOutput()` [p.13] to create objects that implement this interface.

The `LSSerializer` [p.29] will use the `LSOutput` object to determine where to serialize the output to. The `LSSerializer` will look at the different outputs specified in the `LSOutput` in the following order to know which one to output to, the first one that is not null and not an empty string will be used:

1. `LSOutput.characterStream` [p.36]
2. `LSOutput.byteStream` [p.36]
3. `LSOutput.systemId` [p.37]

`LSOutput` objects belong to the application. The DOM implementation will never modify them (though it may make copies and modify the copies, if necessary).

IDL Definition

```
interface LSOutput {
  // Depending on the language binding in use,
  // this attribute may not be available.
           attribute LSWriter        characterStream;
           attribute LSOutputStream  byteStream;
           attribute DOMString       systemId;
           attribute DOMString       encoding;
};
```

Attributes

`byteStream` of type `LSOutputStream` [p.10]

An attribute of a language and binding dependent type that represents a writable stream of bytes.

`characterStream` of type `LSWriter` [p.10]

Depending on the language binding in use, this attribute may not be available.

An attribute of a language and binding dependent type that represents a writable stream to which 16-bit units [p.55] can be output.

`encoding` of type `DOMString`
: The character encoding to use for the output. The encoding must be a string acceptable for an XML encoding declaration ([*XML 1.0*] section 4.3.3 "Character Encoding in Entities"), it is recommended that character encodings registered (as charsets) with the Internet Assigned Numbers Authority [*IANA-CHARSETS*] should be referred to using their registered names.

`systemId` of type `DOMString`
: The system identifier, a URI reference [*IETF RFC 2396*], for this output destination. If the system ID is a relative URI reference (see section 5 in [*IETF RFC 2396*]), the behavior is implementation dependent.

Interface *LSSerializerFilter*

`LSSerializerFilters` provide applications the ability to examine nodes as they are being serialized and decide what nodes should be serialized or not. The `LSSerializerFilter` interface is based on the `NodeFilter` interface defined in [*DOM Level 2 Traversal and Range*].

`Document`, `DocumentType`, `DocumentFragment`, `Notation`, `Entity`, and children of `Attr` nodes are not passed to the filter. The child nodes of an `EntityReference` node are only passed to the filter if the `EntityReference` node is skipped by the method `LSParserFilter.acceptNode()` [p.27] .

When serializing an `Element`, the element is passed to the filter before any of its attributes are passed to the filter. Namespace declaration attributes, and default attributes (except in the case when "discard-default-content [p.32] " is set to `false`), are never passed to the filter.

The result of any attempt to modify a node passed to a `LSSerializerFilter` is implementation dependent.

DOM applications must not raise exceptions in a filter. The effect of throwing exceptions from a filter is DOM implementation dependent.

For efficiency, a node passed to the filter may not be the same as the one that is actually in the tree. And the actual node (node object identity) may be reused during the process of filtering and serializing a document.

IDL Definition

```
interface LSSerializerFilter : traversal::NodeFilter {
  readonly attribute unsigned long   whatToShow;
};
```

Attributes

`whatToShow` of type `unsigned long`, readonly
: Tells the `LSSerializer` [p.29] what types of nodes to show to the filter. If a node is not shown to the filter using this attribute, it is automatically serialized. See `NodeFilter` for definition of the constants. The constants `SHOW_DOCUMENT`, `SHOW_DOCUMENT_TYPE`, `SHOW_DOCUMENT_FRAGMENT`, `SHOW_NOTATION`, and `SHOW_ENTITY` are meaningless here, such nodes will never be passed to a `LSSerializerFilter`. Unlike [*DOM Level 2 Traversal and Range*], the `SHOW_ATTRIBUTE` constant indicates that the `Attr` nodes are shown and passed to the filter.

The constants used here are defined in [*DOM Level 2 Traversal and Range*].

Appendix A: IDL Definitions

This appendix contains the complete OMG IDL [*OMG IDL*] for the Level 3 Document Object Model Abstract Schemas and Load and Save definitions.

The IDL files are also available as: http://www.w3.org/TR/2004/PR-DOM-Level-3-LS-20040205/idl.zip

ls.idl:

```
// File: ls.idl

#ifndef _LS_IDL_
#define _LS_IDL_

#include "dom.idl"
#include "events.idl"
#include "traversal.idl"

#pragma prefix "dom.w3c.org"
module ls
{

  typedef   Object LSInputStream;

  typedef   Object LSOutputStream;

  typedef   Object LSReader;

  typedef   Object LSWriter;

  typedef dom::DOMString DOMString;
  typedef dom::DOMConfiguration DOMConfiguration;
  typedef dom::Node Node;
  typedef dom::Document Document;
  typedef dom::Element Element;

  interface LSParser;
  interface LSSerializer;
  interface LSInput;
  interface LSOutput;
  interface LSParserFilter;
  interface LSSerializerFilter;

  exception LSException {
    unsigned short   code;
  };
  // LSExceptionCode
  const unsigned short      PARSE_ERR                     = 81;
  const unsigned short      SERIALIZE_ERR                 = 82;


  interface DOMImplementationLS {

    // DOMImplementationLSMode
```

```
  const unsigned short      MODE_SYNCHRONOUS               = 1;
  const unsigned short      MODE_ASYNCHRONOUS              = 2;

  LSParser          createLSParser(in unsigned short mode,
                                   in DOMString schemaType)
                                     raises(dom::DOMException);
  LSSerializer      createLSSerializer();
  LSInput           createLSInput();
  LSOutput          createLSOutput();
};

interface LSParser {
  readonly attribute DOMConfiguration domConfig;
           attribute LSParserFilter  filter;
  readonly attribute boolean         async;
  readonly attribute boolean         busy;
  Document          parse(in LSInput input)
                                     raises(dom::DOMException,
                                            LSException);
  Document          parseURI(in DOMString uri)
                                     raises(dom::DOMException,
                                            LSException);

  // ACTION_TYPES
  const unsigned short      ACTION_APPEND_AS_CHILDREN      = 1;
  const unsigned short      ACTION_REPLACE_CHILDREN        = 2;
  const unsigned short      ACTION_INSERT_BEFORE           = 3;
  const unsigned short      ACTION_INSERT_AFTER            = 4;
  const unsigned short      ACTION_REPLACE                 = 5;

  Node              parseWithContext(in LSInput input,
                                     in Node contextArg,
                                     in unsigned short action)
                                     raises(dom::DOMException,
                                            LSException);
  void              abort();
};

interface LSInput {
  // Depending on the language binding in use,
  // this attribute may not be available.
           attribute LSReader        characterStream;
           attribute LSInputStream   byteStream;
           attribute DOMString       stringData;
           attribute DOMString       systemId;
           attribute DOMString       publicId;
           attribute DOMString       baseURI;
           attribute DOMString       encoding;
           attribute boolean         certifiedText;
};

interface LSResourceResolver {
  LSInput           resolveResource(in DOMString type,
                                    in DOMString namespaceURI,
                                    in DOMString publicId,
                                    in DOMString systemId,
                                    in DOMString baseURI);
```

```
  };

  interface LSParserFilter {

    // Constants returned by startElement and acceptNode
    const short               FILTER_ACCEPT                 = 1;
    const short               FILTER_REJECT                 = 2;
    const short               FILTER_SKIP                   = 3;
    const short               FILTER_INTERRUPT              = 4;

    unsigned short     startElement(in Element elementArg);
    unsigned short     acceptNode(in Node nodeArg);
    readonly attribute unsigned long   whatToShow;
  };

  interface LSSerializer {
    readonly attribute DOMConfiguration domConfig;
             attribute DOMString       newLine;
             attribute LSSerializerFilter filter;
    boolean            write(in Node nodeArg,
                             in LSOutput destination)
                                       raises(LSException);
    boolean            writeToURI(in Node nodeArg,
                                  in DOMString uri)
                                       raises(LSException);
    DOMString          writeToString(in Node nodeArg)
                                       raises(dom::DOMException,
                                              LSException);
  };

  interface LSOutput {
    // Depending on the language binding in use,
    // this attribute may not be available.
             attribute LSWriter        characterStream;
             attribute LSOutputStream  byteStream;
             attribute DOMString       systemId;
             attribute DOMString       encoding;
  };

  interface LSProgressEvent : events::Event {
    readonly attribute LSInput         input;
    readonly attribute unsigned long   position;
    readonly attribute unsigned long   totalSize;
  };

  interface LSLoadEvent : events::Event {
    readonly attribute Document        newDocument;
    readonly attribute LSInput         input;
  };

  interface LSSerializerFilter : traversal::NodeFilter {
    readonly attribute unsigned long   whatToShow;
  };
};

#endif // _LS_IDL_
```

ls.idl:

Appendix B: Java Language Binding

This appendix contains the complete Java [*Java*] bindings for the Level 3 Document Object Model Load and Save.

The Java files are also available as
http://www.w3.org/TR/2004/PR-DOM-Level-3-LS-20040205/java-binding.zip

org/w3c/dom/ls/LSException.java:

```
package org.w3c.dom.ls;

public class LSException extends RuntimeException {
    public LSException(short code, String message) {
       super(message);
       this.code = code;
    }
    public short   code;
    // LSExceptionCode
    public static final short PARSE_ERR                = 81;
    public static final short SERIALIZE_ERR            = 82;

}
```

org/w3c/dom/ls/DOMImplementationLS.java:

```
package org.w3c.dom.ls;

import org.w3c.dom.DOMException;

public interface DOMImplementationLS {
    // DOMImplementationLSMode
    public static final short MODE_SYNCHRONOUS          = 1;
    public static final short MODE_ASYNCHRONOUS         = 2;

    public LSParser createLSParser(short mode,
                                   String schemaType)
                                   throws DOMException;

    public LSSerializer createLSSerializer();

    public LSInput createLSInput();

    public LSOutput createLSOutput();

}
```

org/w3c/dom/ls/LSParser.java:

```
package org.w3c.dom.ls;

import org.w3c.dom.Document;
import org.w3c.dom.DOMConfiguration;
import org.w3c.dom.Node;
import org.w3c.dom.DOMException;

public interface LSParser {
    public DOMConfiguration getDomConfig();

    public LSParserFilter getFilter();
    public void setFilter(LSParserFilter filter);

    public boolean getAsync();

    public boolean getBusy();

    public Document parse(LSInput input)
                         throws DOMException, LSException;

    public Document parseURI(String uri)
                            throws DOMException, LSException;

    // ACTION_TYPES
    public static final short ACTION_APPEND_AS_CHILDREN = 1;
    public static final short ACTION_REPLACE_CHILDREN   = 2;
    public static final short ACTION_INSERT_BEFORE      = 3;
    public static final short ACTION_INSERT_AFTER       = 4;
    public static final short ACTION_REPLACE            = 5;

    public Node parseWithContext(LSInput input,
                                 Node contextArg,
                                 short action)
                                 throws DOMException, LSException;

    public void abort();

}
```

org/w3c/dom/ls/LSInput.java:

```
package org.w3c.dom.ls;

public interface LSInput {
    public java.io.Reader getCharacterStream();
    public void setCharacterStream(java.io.Reader characterStream);

    public java.io.InputStream getByteStream();
    public void setByteStream(java.io.InputStream byteStream);

    public String getStringData();
    public void setStringData(String stringData);

    public String getSystemId();
```

```
    public void setSystemId(String systemId);

    public String getPublicId();
    public void setPublicId(String publicId);

    public String getBaseURI();
    public void setBaseURI(String baseURI);

    public String getEncoding();
    public void setEncoding(String encoding);

    public boolean getCertifiedText();
    public void setCertifiedText(boolean certifiedText);

}
```

org/w3c/dom/ls/LSResourceResolver.java:

```
package org.w3c.dom.ls;

public interface LSResourceResolver {
    public LSInput resolveResource(String type,
                                   String namespaceURI,
                                   String publicId,
                                   String systemId,
                                   String baseURI);

}
```

org/w3c/dom/ls/LSParserFilter.java:

```
package org.w3c.dom.ls;

import org.w3c.dom.Element;
import org.w3c.dom.Node;

public interface LSParserFilter {
    // Constants returned by startElement and acceptNode
    public static final short FILTER_ACCEPT             = 1;
    public static final short FILTER_REJECT             = 2;
    public static final short FILTER_SKIP               = 3;
    public static final short FILTER_INTERRUPT          = 4;

    public short startElement(Element elementArg);

    public short acceptNode(Node nodeArg);

    public int getWhatToShow();

}
```

org/w3c/dom/ls/LSProgressEvent.java:

```
package org.w3c.dom.ls;

import org.w3c.dom.events.Event;

public interface LSProgressEvent extends Event {
    public LSInput getInput();

    public int getPosition();

    public int getTotalSize();

}
```

org/w3c/dom/ls/LSLoadEvent.java:

```
package org.w3c.dom.ls;

import org.w3c.dom.Document;
import org.w3c.dom.events.Event;

public interface LSLoadEvent extends Event {
    public Document getNewDocument();

    public LSInput getInput();

}
```

org/w3c/dom/ls/LSSerializer.java:

```
package org.w3c.dom.ls;

import org.w3c.dom.DOMConfiguration;
import org.w3c.dom.Node;
import org.w3c.dom.DOMException;

public interface LSSerializer {
    public DOMConfiguration getDomConfig();

    public String getNewLine();
    public void setNewLine(String newLine);

    public LSSerializerFilter getFilter();
    public void setFilter(LSSerializerFilter filter);

    public boolean write(Node nodeArg,
                         LSOutput destination)
                         throws LSException;

    public boolean writeToURI(Node nodeArg,
                              String uri)
                              throws LSException;
```

```
    public String writeToString(Node nodeArg)
                                throws DOMException, LSException;

}
```

org/w3c/dom/ls/LSOutput.java:

```
package org.w3c.dom.ls;

public interface LSOutput {
    public java.io.Writer getCharacterStream();
    public void setCharacterStream(java.io.Writer characterStream);

    public java.io.OutputStream getByteStream();
    public void setByteStream(java.io.OutputStream byteStream);

    public String getSystemId();
    public void setSystemId(String systemId);

    public String getEncoding();
    public void setEncoding(String encoding);

}
```

org/w3c/dom/ls/LSSerializerFilter.java:

```
package org.w3c.dom.ls;

import org.w3c.dom.traversal.NodeFilter;

public interface LSSerializerFilter extends NodeFilter {
    public int getWhatToShow();

}
```

org/w3c/dom/ls/LSSerializerFilter.java:

Appendix C: ECMAScript Language Binding

This appendix contains the complete ECMAScript [*ECMAScript*] binding for the Level 3 Document Object Model Load and Save definitions.

Properties of the **LSException** Constructor function:

LSException.PARSE_ERR

The value of the constant **LSException.PARSE_ERR** is **81**.

LSException.SERIALIZE_ERR

The value of the constant **LSException.SERIALIZE_ERR** is **82**.

Objects that implement the **LSException** interface:

Properties of objects that implement the **LSException** interface:

code

This property is a **Number**.

Properties of the **DOMImplementationLS** Constructor function:

DOMImplementationLS.MODE_SYNCHRONOUS

The value of the constant **DOMImplementationLS.MODE_SYNCHRONOUS** is **1**.

DOMImplementationLS.MODE_ASYNCHRONOUS

The value of the constant **DOMImplementationLS.MODE_ASYNCHRONOUS** is **2**.

Objects that implement the **DOMImplementationLS** interface:

Functions of objects that implement the **DOMImplementationLS** interface:

createLSParser(mode, schemaType)

This function returns an object that implements the **LSParser** interface.

The **mode** parameter is a **Number**.

The **schemaType** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

createLSSerializer()

This function returns an object that implements the **LSSerializer** interface.

createLSInput()

This function returns an object that implements the **LSInput** interface.

createLSOutput()

This function returns an object that implements the **LSOutput** interface.

Properties of the **LSParser** Constructor function:

LSParser.ACTION_APPEND_AS_CHILDREN

The value of the constant **LSParser.ACTION_APPEND_AS_CHILDREN** is **1**.

LSParser.ACTION_REPLACE_CHILDREN

The value of the constant **LSParser.ACTION_REPLACE_CHILDREN** is **2**.

LSParser.ACTION_INSERT_BEFORE

The value of the constant **LSParser.ACTION_INSERT_BEFORE** is **3**.

LSParser.ACTION_INSERT_AFTER

The value of the constant **LSParser.ACTION_INSERT_AFTER** is **4**.

LSParser.ACTION_REPLACE

The value of the constant **LSParser.ACTION_REPLACE** is **5**.

Objects that implement the **LSParser** interface:

Properties of objects that implement the **LSParser** interface:

domConfig

This read-only property is an object that implements the **DOMConfiguration** interface.

filter

This property is an object that implements the **LSParserFilter** interface.

async

This read-only property is a **Boolean**.

busy

This read-only property is a **Boolean**.

Functions of objects that implement the **LSParser** interface:

parse(input)

This function returns an object that implements the **Document** interface.
The **input** parameter is an object that implements the **LSInput** interface.
This function can raise an object that implements the **DOMException** interface or the **LSException** interface.

parseURI(uri)

This function returns an object that implements the **Document** interface.
The **uri** parameter is a **String**.
This function can raise an object that implements the **DOMException** interface or the **LSException** interface.

parseWithContext(input, contextArg, action)

This function returns an object that implements the **Node** interface.
The **input** parameter is an object that implements the **LSInput** interface.
The **contextArg** parameter is an object that implements the **Node** interface.
The **action** parameter is a **Number**.
This function can raise an object that implements the **DOMException** interface or the **LSException** interface.

abort()

This function has no return value.

Objects that implement the **LSInput** interface:

Properties of objects that implement the **LSInput** interface:

byteStream

This property is an object that implements the **Object** interface.

stringData

This property is a **String**.

systemId

This property is a **String**.

publicId

This property is a **String**.

baseURI

This property is a **String**.

encoding

This property is a **String**.

certifiedText

This property is a **Boolean**.

Objects that implement the **LSResourceResolver** interface:

Functions of objects that implement the **LSResourceResolver** interface:

resolveResource(type, namespaceURI, publicId, systemId, baseURI)

This function returns an object that implements the **LSInput** interface.

The **type** parameter is a **String**.

The **namespaceURI** parameter is a **String**.

The **publicId** parameter is a **String**.

The **systemId** parameter is a **String**.

The **baseURI** parameter is a **String**.

Properties of the **LSParserFilter** Constructor function:

LSParserFilter.FILTER_ACCEPT

The value of the constant **LSParserFilter.FILTER_ACCEPT** is **1**.

LSParserFilter.FILTER_REJECT

The value of the constant **LSParserFilter.FILTER_REJECT** is **2**.

LSParserFilter.FILTER_SKIP

The value of the constant **LSParserFilter.FILTER_SKIP** is **3**.

LSParserFilter.FILTER_INTERRUPT

The value of the constant **LSParserFilter.FILTER_INTERRUPT** is **4**.

Objects that implement the **LSParserFilter** interface:

Properties of objects that implement the **LSParserFilter** interface:

whatToShow

This read-only property is a **Number**.

Functions of objects that implement the **LSParserFilter** interface:

startElement(elementArg)

This function returns a **Number**.

The **elementArg** parameter is an object that implements the **Element** interface.

acceptNode(nodeArg)

This function returns a **Number**.

The **nodeArg** parameter is an object that implements the **Node** interface.

Objects that implement the **LSProgressEvent** interface:

Objects that implement the **LSProgressEvent** interface have all properties and functions of the **Event** interface as well as the properties and functions defined below.

Properties of objects that implement the **LSProgressEvent** interface:

input

This read-only property is an object that implements the **LSInput** interface.

position

This read-only property is a **Number**.

totalSize

This read-only property is a **Number**.

Objects that implement the **LSLoadEvent** interface:

Objects that implement the **LSLoadEvent** interface have all properties and functions of the **Event** interface as well as the properties and functions defined below.

Properties of objects that implement the **LSLoadEvent** interface:

newDocument

This read-only property is an object that implements the **Document** interface.

input

This read-only property is an object that implements the **LSInput** interface.

Objects that implement the **LSSerializer** interface:

Properties of objects that implement the **LSSerializer** interface:

domConfig

This read-only property is an object that implements the **DOMConfiguration** interface.

newLine

This property is a **String**.

filter

This property is an object that implements the **LSSerializerFilter** interface.

Functions of objects that implement the **LSSerializer** interface:

write(nodeArg, destination)

This function returns a **Boolean**.
The **nodeArg** parameter is an object that implements the **Node** interface.
The **destination** parameter is an object that implements the **LSOutput** interface.
This function can raise an object that implements the **LSException** interface.

writeToURI(nodeArg, uri)

This function returns a **Boolean**.
The **nodeArg** parameter is an object that implements the **Node** interface.
The **uri** parameter is a **String**.
This function can raise an object that implements the **LSException** interface.

writeToString(nodeArg)

This function returns a **String**.
The **nodeArg** parameter is an object that implements the **Node** interface.
This function can raise an object that implements the **DOMException** interface or the **LSException** interface.

Objects that implement the **LSOutput** interface:

Properties of objects that implement the **LSOutput** interface:

byteStream

This property is an object that implements the **Object** interface.

systemId

This property is a **String**.

encoding

This property is a **String**.

Objects that implement the **LSSerializerFilter** interface:

Objects that implement the **LSSerializerFilter** interface have all properties and functions of the **NodeFilter** interface as well as the properties and functions defined below.

Properties of objects that implement the **LSSerializerFilter** interface:

whatToShow

This read-only property is a **Number**.

Appendix D: Acknowledgements

Many people contributed to the DOM specifications (Level 1, 2 or 3), including participants of the DOM Working Group and the DOM Interest Group. We especially thank the following:

Andrew Watson (Object Management Group), Andy Heninger (IBM), Angel Diaz (IBM), Arnaud Le Hors (W3C and IBM), Ashok Malhotra (IBM and Microsoft), Ben Chang (Oracle), Bill Smith (Sun), Bill Shea (Merrill Lynch), Bob Sutor (IBM), Chris Lovett (Microsoft), Chris Wilson (Microsoft), David Brownell (Sun), David Ezell (Hewlett-Packard Company), David Singer (IBM), Dimitris Dimitriadis (Improve AB and invited expert), Don Park (invited), Elena Litani (IBM), Eric Vasilik (Microsoft), Gavin Nicol (INSO), Ian Jacobs (W3C), James Clark (invited), James Davidson (Sun), Jared Sorensen (Novell), Jeroen van Rotterdam (X-Hive Corporation), Joe Kesselman (IBM), Joe Lapp (webMethods), Joe Marini (Macromedia), Johnny Stenback (Netscape/AOL), Jon Ferraiolo (Adobe), Jonathan Marsh (Microsoft), Jonathan Robie (Texcel Research and Software AG), Kim Adamson-Sharpe (SoftQuad Software Inc.), Lauren Wood (SoftQuad Software Inc., *former Chair*), Laurence Cable (Sun), Mark Davis (IBM), Mark Scardina (Oracle), Martin Dürst (W3C), Mary Brady (NIST), Mick Goulish (Software AG), Mike Champion (Arbortext and Software AG), Miles Sabin (Cromwell Media), Patti Lutsky (Arbortext), Paul Grosso (Arbortext), Peter Sharpe (SoftQuad Software Inc.), Phil Karlton (Netscape), Philippe Le Hégaret (W3C, *W3C Team Contact and former Chair*), Ramesh Lekshmynarayanan (Merrill Lynch), Ray Whitmer (iMall, Excite@Home, and Netscape/AOL, *Chair*), Rezaur Rahman (Intel), Rich Rollman (Microsoft), Rick Gessner (Netscape), Rick Jelliffe (invited), Rob Relyea (Microsoft), Scott Isaacs (Microsoft), Sharon Adler (INSO), Steve Byrne (JavaSoft), Tim Bray (invited), Tim Yu (Oracle), Tom Pixley (Netscape/AOL), Vidur Apparao (Netscape), Vinod Anupam (Lucent).

Thanks to all those who have helped to improve this specification by sending suggestions and corrections (Please, keep bugging us with your issues!).

Many thanks to Elliotte Rusty Harold, Andrew Clover, Anjana Manian, Christian Parpart, Mikko Honkala, and François Yergeau for their review and comments of this document.

Special thanks to the DOM Conformance Test Suites contributors: Fred Drake, Mary Brady (NIST), Rick Rivello (NIST), Robert Clary (Netscape), with a special mention to Curt Arnold.

D.1 Production Systems

This specification was written in XML. The HTML, OMG IDL, Java and ECMAScript bindings were all produced automatically.

Thanks to Joe English, author of cost, which was used as the basis for producing DOM Level 1. Thanks also to Gavin Nicol, who wrote the scripts which run on top of cost. Arnaud Le Hors and Philippe Le Hégaret maintained the scripts.

After DOM Level 1, we used Xerces as the basis DOM implementation and wish to thank the authors. Philippe Le Hégaret and Arnaud Le Hors wrote the Java programs which are the DOM application.

Thanks also to Jan Kärrman, author of html2ps, which we use in creating the PostScript version of the specification.

Glossary

Editors:
Arnaud Le Hors, W3C
Robert S. Sutor, IBM Research (for DOM Level 1)

Some of the following term definitions have been borrowed or modified from similar definitions in other W3C or standards documents. See the links within the definitions for more information.

16-bit unit
The base unit of a `DOMString`. This indicates that indexing on a `DOMString` occurs in units of 16 bits. This must not be misunderstood to mean that a `DOMString` can store arbitrary 16-bit units. A `DOMString` is a character string encoded in UTF-16; this means that the restrictions of UTF-16 as well as the other relevant restrictions on character strings must be maintained. A single character, for example in the form of a numeric character reference, may correspond to one or two 16-bit units.

API
An *API* is an Application Programming Interface, a set of functions or methods used to access some functionality.

namespace well-formed
A node is a *namespace well-formed* XML node if it is a well-formed [p.55] node, and follows the productions and namespace constraints. If [*XML 1.0*] is used, the constraints are defined in [*XML Namespaces*]. If [*XML 1.1*] is used, the constraints are defined in [*XML Namespaces 1.1*].

read only node
A *read only node* is a node that is immutable. This means its list of children, its content, and its attributes, when it is an element, cannot be changed in any way. However, a read only node can possibly be moved, when it is not itself contained in a read only node.

schema
A *schema* defines a set of structural and value constraints applicable to XML documents. Schemas can be expressed in schema languages, such as DTD, XML Schema, etc.

well-formed
A node is a *well-formed* XML node if its serialized form, without doing any transformation during its serialization, matches its respective production in [*XML 1.0*] or [*XML 1.1*] (depending on the XML version in use) with all well-formedness constraints related to that production, and if the entities which are referenced within the node are also well-formed. If namespaces for XML are in use, the node must also be namespace well-formed [p.55] .

References

For the latest version of any W3C specification please consult the list of W3C Technical Reports available at http://www.w3.org/TR.

F.1 Normative references

[DOM Level 2 Core]
Document Object Model Level 2 Core Specification, A. Le Hors, et al., Editors. World Wide Web Consortium, 13 November 2000. This version of the DOM Level 2 Core Recommendation is http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113. The latest version of DOM Level 2 Core is available at http://www.w3.org/TR/DOM-Level-2-Core.

[DOM Level 3 Core]
Document Object Model Level 3 Core Specification, A. Le Hors, et al., Editors. World Wide Web Consortium, February 2004. This version of the Document Object Model Level 3 Core specification is http://www.w3.org/TR/2004/PR-DOM-Level-3-Core-20040205. The latest version of DOM Level 3 Core is available at http://www.w3.org/TR/DOM-Level-3-Core.

[DOM Level 2 Traversal and Range]
Document Object Model Level 2 Traversal and Range Specification, J. Kesselman, J. Robie, M. Champion, P. Sharpe, V. Apparao, L. Wood, Editors. World Wide Web Consortium, 13 November 2000. This version of the Document Object Model Level 2 Traversal and Range Recommendation is http://www.w3.org/TR/2000/REC-DOM-Level-2-Traversal-Range-20001113. The latest version of Document Object Model Level 2 Traversal and Range is available at http://www.w3.org/TR/DOM-Level-2-Traversal-Range.

[ECMAScript]
ECMAScript Language Specification, Third Edition. European Computer Manufacturers Association, Standard ECMA-262, December 1999. This version of the ECMAScript Language is available from http://www.ecma-international.org/.

[IANA-CHARSETS]
Official Names for Character Sets, K. Simonsen, et al., Editors. Internet Assigned Numbers Authority. Available at ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets.

[ISO/IEC 10646]
ISO/IEC 10646-2000 (E). Information technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane, as, from time to time, amended, replaced by a new edition or expanded by the addition of new parts. [Geneva]: International Organization for Standardization, 2000. See also International Organization for Standardization, available at http://www.iso.ch, for the latest version.

[Java]
The Java Language Specification, J. Gosling, B. Joy, and G. Steele, Authors. Addison-Wesley, September 1996. Available at http://java.sun.com/docs/books/jls

[OMG IDL]
"OMG IDL Syntax and Semantics" defined in *The Common Object Request Broker: Architecture and Specification, version 2*, Object Management Group. The latest version of CORBA version 2.0 is available at http://www.omg.org/technology/documents/formal/corba_2.htm.

[IETF RFC 2396]
Uniform Resource Identifiers (URI): Generic Syntax, T. Berners-Lee, R. Fielding, L. Masinter, Authors. Internet Engineering Task Force, August 1998. Available at http://www.ietf.org/rfc/rfc2396.txt.

[IETF RFC 3023]
XML Media Types, M. Murata, S. St.Laurent, and D. Kohn, Editors. Internet Engineering Task Force, January 2001. Available at http://www.ietf.org/rfc/rfc3023.txt.

[SAX]
Simple API for XML, D. Megginson and D. Brownell, Maintainers. Available at http://www.saxproject.org/.

[Unicode]
The Unicode Standard, Version 4, ISBN 0-321-18578-1, as updated from time to time by the publication of new versions. The Unicode Consortium, 2000. See also Versions of the Unicode Standard, available at http://www.unicode.org/unicode/standard/versions, for latest version and additional information on versions of the standard and of the Unicode Character Database.

[XML 1.0]
Extensible Markup Language (XML) 1.0 (Third Edition), T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, Editors. World Wide Web Consortium, 4 February 2004, revised 10 February 1998 and 6 October 2000. This version of the XML 1.0 Recommendation is http://www.w3.org/TR/2004/REC-xml-20040204. The latest version of XML 1.0 is available at http://www.w3.org/TR/REC-xml.

[XML 1.1]
XML 1.1, T. Bray, and al., Editors. World Wide Web Consortium, 4 February 2004. This version of the XML 1.1 Recommendation is http://www.w3.org/TR/2004/REC-xml11-20040204. The latest version of XML 1.1 is available at http://www.w3.org/TR/xml11.

[XML Information Set]
XML Information Set (Second Edition), J. Cowan and R. Tobin, Editors. World Wide Web Consortium, 4 February 2004, revised 24 October 2001. This version of the XML Information Set Recommendation is http://www.w3.org/TR/2004/REC-xml-infoset-20040204. The latest version of XML Information Set is available at http://www.w3.org/TR/xml-infoset.

[XML Namespaces]
Namespaces in XML, T. Bray, D. Hollander, and A. Layman, Editors. World Wide Web Consortium, 14 January 1999. This version of the Namespaces in XML Recommendation is http://www.w3.org/TR/1999/REC-xml-names-19990114. The latest version of Namespaces in XML is available at http://www.w3.org/TR/REC-xml-names.

[XML Namespaces 1.1]
Namespaces in XML 1.1, T. Bray, D. Hollander, A. Layman, and R. Tobin, Editors. World Wide Web Consortium, 4 February 2004. This version of the Namespaces in XML 1.1 Recommendation is http://www.w3.org/TR/2004/REC-xml-names11-20040204. The latest version of Namespaces in XML 1.1 is available at http://www.w3.org/TR/xml-names11/.

F.2 Informative references

[Canonical XML]
Canonical XML Version 1.0, J. Boyer, Editor. World Wide Web Consortium, 15 March 2001. This version of the Canonical XML Recommendation is

http://www.w3.org/TR/2001/REC-xml-c14n-20010315. The latest version of Canonical XML is available at http://www.w3.org/TR/xml-c14n.

[DOM Level 3 Events]

Document Object Model Level 3 Events Specification, P. Le Hégaret, T. Pixley, Editors. World Wide Web Consortium, November 2003. This version of the Document Object Model Level 3 Events specification is http://www.w3.org/TR/2003/NOTE-DOM-Level-3-Events-20031107. The latest version of Document Object Model Level 3 Events is available at http://www.w3.org/TR/DOM-Level-3-Events.

[JAXP]

Java API for XML Processing (JAXP). Sun Microsystems. Available at http://java.sun.com/xml/jaxp/.

[IETF RFC 2616]

Hypertext Transfer Protocol -- HTTP/1.1, R. Fielding, et al., Authors. Internet Engineering Task Force, June 1999. Available at http://www.ietf.org/rfc/rfc2616.txt.

[XML Schema Part 1]

XML Schema Part 1: Structures, H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, Editors. World Wide Web Consortium, 2 May 2001. This version of the XML Part 1 Recommendation is http://www.w3.org/TR/2001/REC-xmlschema-1-20010502. The latest version of XML Schema Part 1 is available at http://www.w3.org/TR/xmlschema-1.

F.2 Informative references

Index

"ignore-unknown-character-denormalizations"

"canonical-form"
"charset-overrides-xml-encoding"
"disallow-doctype" 14, 18
"discard-default-content" 29, 32, 37, 32
"format-pretty-print" 32, 32
"ignore-unknown-character-denormalizations" 14, 18
"infoset"
"namespaces"
"normalize-characters"
"resource-resolver"
"supported-media-types-only" 14, 19
"xml-declaration" 29, 32, 33

16-bit unit 10, 10, 23, 24, 36, 55

[attributes]

abort
acceptNode
ACTION_APPEND_AS_CHILDREN
ACTION_INSERT_AFTER
ACTION_INSERT_BEFORE
ACTION_REPLACE
ACTION_REPLACE_CHILDREN
API 9, 55
async

baseURI
busy
byteStream 23, 36

Canonical XML 32, 58
certifiedText
characterStream 23, 36
createLSInput
createLSOutput
createLSParser
createLSSerializer

DOM Level 2 Core 9, 11, 29, 57
DOM Level 2 Traversal and Range 27, 37, 37, 57
DOM Level 3 Core 11, 12, 14, 17, 20, 29, 32, 57
DOM Level 3 Events 28, 59
domConfig 17, 32
DOMImplementationLS

ECMAScript
encoding 23, 37

filter 19, 33
FILTER_ACCEPT
FILTER_INTERRUPT
FILTER_REJECT
FILTER_SKIP

IANA-CHARSETS 37, 57
IETF RFC 2396 20, 24, 23, 25, 37, 58
IETF RFC 2616 17, 23, 59
IETF RFC 3023 17, 58
input 29, 29
ISO/IEC 10646 10, 10, 23, 57

Java
JAXP 9, 59

load
LSException
LSInput
LSInputStream
LSLoadEvent
LSOutput
LSOutputStream
LSParser
LSParserFilter
LSProgressEvent
LSReader
LSResourceResolver
LSSerializer
LSSerializerFilter
LSWriter

MODE_ASYNCHRONOUS
MODE_SYNCHRONOUS

namespace well-formed
newDocument
newLine

OMG IDL

parse
PARSE_ERR
parseURI
parseWithContext
position
progress
publicId

read only node 20, 55
resolveResource

SAX 9, 24, 58
schema 13, 55
SERIALIZE_ERR
startElement
stringData
systemId 24, 37

totalSize

Unicode 10, 10, 23, 58

well-formed 28, 27, 29, 55
whatToShow 27, 37
write
writeToString
writeToURI

XML 1.0 13, 17, 23, 25, 32, 33, 37, 55, 55, 58
XML 1.1 17, 23, 29, 32, 33, 55, 55, 58
XML Information Set 15, 58
XML Namespaces 17, 55, 58
XML Namespaces 1.1 17, 55, 58
XML Schema Part 1 13, 25, 59