Using *Mathematica*, you can import and transform arbitrary XML documents, export notebooks as XML files, and work with popular XML formats such as SVG and MathML. This article gives an overview of *Mathematica*’s support for XML and includes detailed information on the large number of new XML-related functions and options.

**Introduction**

Starting with Version 4.2, *Mathematica* includes comprehensive support for XML, the metamarkup language developed by the World Wide Web Consortium (W3C) for describing structured documents and data. Using the new XML features introduced in Version 4.2, you can do any of the following.

- Import any arbitrary Extensible Markup Language (XML) document in the form of a *Mathematica* expression.
- Analyze the contents of an XML document or transform its structure using *Mathematica*’s sophisticated programming and symbolic manipulation abilities.
- Export the resulting expression back as an XML document to share it with other users and applications.
- Save *Mathematica* notebooks in an XML format using the new NotebookML document type definition (DTD) defined for this purpose.
- Import, export, and evaluate equations in MathML—the W3C standard for representing mathematical formulas on the web.
- Export graphics in Scalable Vector Graphics (SVG) format—the W3C standard for representing graphics on the web.

These new features make *Mathematica* a powerful development tool for creating and processing XML documents. They ensure complete interoperability between *Mathematica* and other XML applications and between notebooks and other XML document formats. This article provides detailed information on *Mathematica*’s XML-related capabilities.
Native XML Formats

Mathematica has built-in support for four XML formats: MathML, SVG, NotebookML, and ExpressionML. More details about each of these formats are given below.

- **MathML**

  Mathematical Markup Language (MathML) is an XML format developed by the W3C for describing the structure and symbolic meaning of mathematical formulas. It provides a standard way of displaying mathematical notation in web pages. Version 4.1 of Mathematica included limited support for import and export of MathML. In Version 4.2, the support was greatly expanded with many new functions for generating and manipulating MathML and for converting between MathML and the expressions used internally by Mathematica to represent mathematics.

  The new MathML features make Mathematica an excellent tool for authoring and editing MathML content. You can, for example, use Mathematica’s powerful typesetting system to create properly formatted equations and then copy and paste them in MathML format into an HTML document for display on the web. You can also import MathML equations from other applications and evaluate them using Mathematica.

- **SVG**

  SVG is an XML format developed by the W3C for describing two-dimensional graphics. SVG images can be rescaled without loss of resolution and are usually much smaller in size than comparable JPEG or GIF images. SVG files can also be manipulated with a scripting language to produce dynamic and interactive graphics. Using Mathematica, you can directly export any graphics present in a notebook, in SVG format.

- **NotebookML**

  NotebookML is an XML format developed by Wolfram Research for representing Mathematica notebooks. The tags and attributes used in NotebookML are specified by an XML DTD and correspond closely to the structures used in notebook expressions. Using NotebookML, you can save your notebooks as well-formed XML documents and then import them back into Mathematica in a completely lossless way.

  NotebookML provides a bridge between Mathematica and other XML applications, allowing you to use notebooks with standard web technologies such as Cascading Style Sheets (CSS) and Extensible Stylesheet Language Transformations (XSLT). For example, you can display a NotebookML document in a web browser by using a CSS style sheet to specify how each element in the document should be rendered. Support for MathML and SVG is built into NotebookML, in that you can choose to save all equations in a notebook as MathML and all graphics as SVG when saving the notebook as NotebookML.
ExpressionML

ExpressionML is a specialized subset of NotebookML. ExpressionML fragments can represent any Mathematica expression in an XML format. NotebookML uses ExpressionML fragments to represent Mathematica expressions that are embedded within a notebook structure.

SymbolicXML

What Is SymbolicXML?

SymbolicXML is the format used by Mathematica for representing XML documents. The conversion from XML to SymbolicXML translates the XML document into a Mathematica expression, while preserving its structure. Since both XML documents and Mathematica expressions have a tree structure, there is a natural mapping from one to the other. You can then manipulate the SymbolicXML expression using the standard techniques of Mathematica programming.

You can import XML data into Mathematica using the standard Import or ImportString functions. You can also control various details of the import process, such as how to treat whitespace, whether to recognize entities, or whether to validate against a DTD, by using the conversion options of the Import function.

The following command imports an XML data file into Mathematica.

Import["data.xml","XML"]
Unless the file you are importing is in an XML format recognized by *Mathematica*, such as MathML, the result is a SymbolicXML expression, `expr1`. You can then manipulate the expression using standard *Mathematica* commands to produce another SymbolicXML expression, `expr2`.

\[ expr1 \rightarrow expr2 \]

Finally, you can export the result as an XML file using the standard `Export` function.

\[ Export["newdata.xml", expr2, "XML"] \]

You can use conversion options to control various details of the export process, such as the format of the exported XML, using the conversion options of the `Export` function.

The combination of SymbolicXML and *Mathematica* programming provides a useful alternative to other techniques for manipulating XML documents, such as XSLT transformations or the SAX or DOM APIs used with a low-level programming language such as Java. *Mathematica* allows you to achieve the same level of flexibility and control in processing XML documents. The advantage is that you can leverage *Mathematica*’s advanced support for symbolic manipulation and numerical computation to do some very complex and sophisticated transformations that would be difficult or impossible to do using other methods.

For example, you can use pattern-matching techniques to extract specific parts of an XML document, perform numerical computations on the data, and then convert the results into 3D graphics for easy visualization. You can also define transformations to convert one type of XML application to another. For example, you can import a DocBook document as SymbolicXML and then convert it into XHTML format by defining suitable transformation rules to replace one set of element names with another set. For some specific examples of useful applications of SymbolicXML, see Transforming XML.

Support for SymbolicXML is well-integrated with NotebookML, ExpressionML, and MathML. For example, when importing an XML document as SymbolicXML, *Mathematica* recognizes if the document is in NotebookML, ExpressionML, or MathML format and automatically converts it into a notebook expression in the case of NotebookML, an expression in the case of ExpressionML, or a typeset box expression in the case of MathML. You can also override the default behavior and choose to import any of these XML flavors as SymbolicXML if you wish. There is also a large number of kernel functions for quickly and easily converting between strings, boxes, or expressions on the one hand, and NotebookML, MathML, or SymbolicXML on the other.

Note that if you prefer to manipulate XML documents using Java directly, you can still do so using the *J/Link* add-on package. This package integrates *Mathematica* fully with Java, enabling you to call Java commands from *Mathematica* or to call *Mathematica* kernel functions from Java programs. You can, thus, have access to both the computational abilities of *Mathematica* as well as the low-level programming features and classes of Java, and combine the two as needed.
**Representing Elements**

Each element in an XML document corresponds to an `XMLElement` command in SymbolicXML. An XML expression of the form

```xml
<element attribute = "value"> data <element>
```

has the following representation in SymbolicXML:

```symbolicxml
XMLElement[element, {attribute -> value}, {data}]
```

Each `XMLElement` command has three arguments:

- The first argument specifies the name of the element.
- The second argument specifies the attributes of the element. This argument is a list of zero or more rules, with each rule specifying a single attribute in the form: "attribute" -> "value".
- The third argument specifies the actual data contained in the element. This can be raw character data in the form of a string and/or child elements of the element being represented. Each child element is represented by its own `XMLElement` command. You can nest multiple `XMLElement` commands to the level necessary to replicate the nested structure of the original XML expression.

The names of all elements and attributes as well as any character data in the XML document are represented as strings in SymbolicXML. This is to prevent a large number of new symbols from being introduced into the *Mathematica* session, which could lead to possible naming conflicts.

Here is a simple XML fragment.

```xml
<book type="novel">Moby Dick</book>
```

Here is the representation of this fragment in SymbolicXML.

```symbolicxml
XMLElement["book", {"type" -> "novel"}, {"Moby Dick"}]
```

Here is a more complicated XML expression, showing several levels of nesting.

```xml
<book type='novel'>
<title>Moby Dick</title>
<author born='1819' died='1891'>
<name>
<first>Herman </first>
<last>Melville</last>
</name>
</author>
</book>
```

Here is the corresponding SymbolicXML expression.

```symbolicxml
XMLElement["book", {"type" -> "novel"}, {
XMLElement["title", {}], {"Moby Dick"}],
XMLElement["author", {"born" -> "1819", "died" -> "1891"}, {
```
### Handling Namespaces

If a namespace is specified in an XML element, the syntax of the corresponding SymbolicXML expression is slightly more complex. The exact syntax depends on whether the namespace is specified implicitly, as a default namespace, or explicitly, using a namespace prefix.

#### Using a default namespace

For any element that lies within a default namespace, the `XMLElement` expression is the same as it would be if no namespace was specified. However, the element in which the default namespace is declared has its `XMLElement` expression modified, as shown in the following example.

Here is a simple XHTML document with a default namespace declared on the `html` element.

```html
<html xmlns='http://www.w3.org/1999/xhtml'>
  <head> </head>
  <body>
    <p>Here is some text.</p>
  </body>
</html>
```

Here is the corresponding SymbolicXML expression.

```
XMLElement["html",{{"http://www.w3.org/2000/xmlns/","xmlns"}→"http://www.w3.org/1999/xhtml"},{
XMLElement["head",{},{}],
XMLElement["body",{},{
XMLElement["p",{},{"Here is some text."}]}]}
```

Note that the `XMLElement` expression representing the `html` element has a complex structure. Its second argument is:

```
{"http://www.w3.org/2000/xmlns/","xmlns"}→"http://www.w3.org/1999/xhtml"
```

This statement accomplishes two things:

- It identifies the attribute `xmlns` with the namespace defined by the URL `http://www.w3.org/2000/xmlns`, as required by the XML specification.
- It sets the value of the `xmlns` attribute to the URL `http://www.w3.org/1999/xhtml`, thus defining the default namespace.

In other words, when declaring a default namespace on an element, the syntax of the corresponding `XMLElement` structure is:
Here xmlns-url is the URL associated with the namespace of the *xmlns* attribute, and namespace-url is the URL of the default namespace being declared.

**Using an explicit namespace prefix**

If the namespace is specified explicitly on an element using a namespace prefix, the syntax of the SymbolicXML expression is modified, as shown in the following example.

Here is an XHTML document with some MathML markup embedded in it. The *xmlns:m* attribute in the *math* element binds the MathML namespace to the namespace prefix *m*. All the MathML element names are then written with this namespace prefix attached.

```
<html xmlns='http://www.w3.org/1999/xhtml'>
<head> <title>Test</title> </head>
<body>
 <p>Here is some math.</p>
 <p>
  <m:math xmlns:m='http://www.w3.org/1998/Math/MathML'>
   <m:mi>x</m:mi>
   <m:mo>+</m:mo>
   <m:mn>1</m:mn>
  </m:math>
 </p>
</body>
</html>
```

Here is the corresponding SymbolicXML expression.

```
XMLElement["html",{{"http://www.w3.org/2000/xmlns/","xmlns"} \to 
"http://www.w3.org/1999/xhtml"},XMLElement[head,{},{}],
XMLElement["body",{},{
  XMLElement["p",{},{"Here is some math."}],
  XMLElement["p",{},{]
ns/","m"} \to "http://www.w3.org/1998/Math/MathML"},
XMLElement[{{"http://www.w3.org/1998/Math/MathML","mi"},{{"x"}],
  XMLElement[{{"http://www.w3.org/1998/Math/MathML","mo"},{{"+"]},
  XMLElement[{{"http://www.w3.org/1998/Math/MathML","mn"},{{"1"} 
  }]}]}]
}

There are two features to note here.

- The first attribute of the XMLElement structure for the top-level *math* element is
  {{"http://www.w3.org/2000/xmlns/","m"} \to 
  "http://www.w3.org/1998/Math/MathML". This associates the MathML namespace with the prefix *m*.}
• The `XMLElement` structure for each MathML element is of the form `XMLElement[{url, element}, {}, {data}]`, where `url` identifies the MathML namespace. This is the SymbolicXML equivalent of writing an element name with the namespace prefix attached.

## Representing Other Objects

The `XMLObject` expression is used as a container for parts of an XML document other than elements, such as comments, processing instructions, or declarations. It is also used as a container for the entire document itself. This structure has the syntax `XMLObject[object][data]`, where `object` describes the type of object being represented and `data` specifies the details of the object. There are six types of objects that can be specified as the first argument, each corresponding to a specific type of XML construct.

- Declaration
- Comment
- Document
- Doctype
- ProcessingInstruction
- CDATASection

### Declaration

The `XMLObject["Declaration"]` expression is used to represent the XML declaration that typically appears at the start of an XML document. This has the following syntax.

```
XMLObject["Declaration"]["Version"->"1.0", option -> value]
```

These two options are allowed.

- "Standalone" takes the value "yes" if the document references an external DTD and "no" otherwise.
- "Encoding" specifies the character encoding used in the document. Not all encodings will be honored on export. If an encoding that *Mathematica* cannot export is specified, an error message is produced and the encoding is changed in the document.

Here is a typical XML declaration.

```
<?xml version="1.0" encoding="ascii" standalone="yes"?>
```

Here is the corresponding SymbolicXML expression.

```
XMLObject["Declaration"]["Version"->"1.0", "Encoding"->"ascii" "Standalone"->"Yes"]
```
**Comment**

The `XMLObject["Comment"]` expression is used to represent XML comments. It has the following syntax.

```
XMLObject["comment"]["string"]
```

Here is an example of an XML comment.

```
<!-- Created on 3/6/02. -->
```

Here is the corresponding SymbolicXML expression.

```
XMLObject["Comment"]["Created on 3/6/02."]
```

**Document**

The most important `XMLObject` is `XMLObject["Document"]`. It is used as a container for the entire document and has the following syntax.

```
XMLObject["Document"][{prolog}, document tree, {epilog}]
```

The prolog may contain an `XMLObject["Declaration"]`, followed by optional processing instructions and DTD declarations. The epilog contains either processing instructions or comments.

Here is an example of a simple document consisting of an XML declaration, a comment, and a single element.

```
<?xml version='1.0'?>
<!--this is a sample file-->
<root/>
```

Here is the corresponding SymbolicXML expression.

```
XMLObject["Document"][{XMLObject["Declaration"]["Version"->
"1.0"],XMLObject["Comment"]["this is a sample file"],XMLElement["root",{},{}],{}}]
```

The only option for `XMLObject["Document"]` is "Valid". This option is set automatically by the parser. If the document was validated on import and validation succeeded, then the option "Valid"→ `True` will be included in the `XMLObject` expression. If validation was attempted but failed, then "Valid"→ `False` will be included in the `XMLObject`. If validation was not attempted, then the option is omitted from the `XMLObject` expression.
Doctype

The `XMLObject["Doctype"]` expression is used to represent XML document type declarations. It has the following syntax.

XMLObject["Doctype"] [name, option → value]

These three options are allowed.

- "System" specifies a DTD in the local file system, either as a relative pathname or a URL.
- "Public" specifies a standardized name that is used to publicly identify the DTD.
- "Internal" specifies an internal DTD subset. Its value is a string that contains the data in the internal DTD subset.

Here is a Doctype declaration with all three options.

```xml
<!DOCTYPE catalog PUBLIC "-//FOO//DTD catalog 1.1//EN"
  "www.foo.com/example/catalog.dtd"
  [internal DTD stuff]>
```

Here is the corresponding SymbolicXML expression.

XMLObject["Doctype"]["catalog", "Public"→"-//FOO//DTD catalog 1.1//EN", "System"→"www.foo.com/example/catalog.dtd", "InternalSubset"→"internal DTD stuff"]

For more details on XML Doctype declarations, see the W3C XML specification at www.w3.org/TR/REC-xml.

ProcessingInstruction

The `XMLObject["ProcessingInstruction"]` expression is used to represent XML processing instructions. This has the following syntax.

XMLObject["ProcessingInstruction"] [target string, optional data string]

It is common to use attribute-like syntax in processing instructions. These pseudo-attributes are not parsed but are returned as raw strings. Here is a processing instruction that specifies a style sheet.

```xml
<?xml-stylesheet href="mystyle.css" type="text/css"?>
```

Here is the corresponding SymbolicXML expression. Notice that the double quotes around the attribute values are escaped to distinguish them from the double quotes around the argument as a whole.

XMLObject["ProcessingInstruction"]["xml-stylesheet", "href="mystyle.css" type="text/css"]
CDATASection

The `XMLObject["CDATASection"]` expression is used to represent CDATA sections. CDATA is a W3C abbreviation for Character Data. CDATA sections are used in an XML document as a wrapper for raw character data to avoid having to escape special characters such as " and <. These characters would normally have to be indicated as `&quote;` and `&lt;`, respectively. CDATA sections are used in XML to enclose character data that would require a lot of escaping, such as programs or math expressions.

Here is a simple fragment from an XML document containing a CDATA section.

```xml
<![CDATA[ 5 < 7 << 2*10^123]]>
```

Here is the corresponding SymbolicXML expression.

```mathematica
XMLObject["CDATASection"][" 5 < 7 << 2*10^123"]
```

By default, `CDATASection` object wrappers are not preserved on import, and only the contents of the CDATA section are retained. To preserve the `CDATASection` wrappers, you must explicitly set the conversion option, `"PreserveCDATASections" -> True`. More information about conversion options is given below.

### Importing XML

#### Functions for Importing XML

- **Import**

  You can import XML data into *Mathematica* using the standard `Import` function, which has the following syntax.

  ```mathematica
  Import[file]
  Import[file, format]
  ```

  The first argument of the function specifies the file to be imported. You can also specify an optional second argument to control the form of the output. For importing XML data, the relevant file formats are: "XML", "NotebookML", "ExpressionML", "MathML", and "SymbolicXML".

  With "XML" as the import format, any XML formats that *Mathematica* does not recognize are returned as SymbolicXML. The formats that *Mathematica* does recognize—NotebookML, ExpressionML, and MathML—are treated slightly differently. A NotebookML file is imported as a notebook expression. An ExpressionML file is imported as the corresponding cell expression. A MathML file is returned as the corresponding box expression.
With "SymbolicXML" as the import format, even NotebookML, ExpressionML, and MathML files are imported as SymbolicXML. This gives you a way to override the XML format from being automatically interpreted.

Here is a file containing the MathML encoding for \( x^2 \).

```
!! eqn.mml

<math xmlns="http://www.w3.org/1998/Math/MathML">
  <semantics>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <annotation-xml encoding="MathML-Content">
      <apply>
        <power/>
        <ci>x</ci>
        <cn type="integer">2</cn>
      </apply>
    </annotation-xml>
  </semantics>
</math>
```

If you import the file and specify "XML" as the second argument, the equation is converted into a box expression.

```
Import["eqn.mml", "XML"]

FormBox[TagBox[TagBox[SuperscriptBox[x, 2], MathMLPresentationTag, AutoDelete → True], AnnotationsTagWrapper[TagBox[SuperscriptBox[x, 2], MathMLContentTag, AutoDelete → True]], AutoDelete → True], TraditionalForm]
```

If you specify "SymbolicXML" as the second argument, the equation is imported as a SymbolicXML box expression.
If `Import` is used with only one argument, *Mathematica* processes the data in the file based on its file extension. Any file with a `.xml` extension is imported as XML. This means that if it is in one of the XML formats explicitly supported by *Mathematica*, namely NotebookML, ExpressionML, or MathML, the file will be interpreted in the appropriate way. All other XML formats are imported as SymbolicXML.

*Mathematica* also recognizes the `.mml` extension for MathML files and the `.nbml` extension for NotebookML files. In the following example, we import a file with the `.mml` extension.

```mathematica
Import["eqn.mml", "SymbolicXML"]
```

You can display the above box expression as conventional mathematical notation by using `DisplayForm`.

```mathematica
DisplayForm[%]
```

You can control the various details of the import process, such as how to treat whitespace, whether to recognize entities, or whether to validate against a DTD, by specifying conversion options to the `Import` function.

- **ImportString**

You can also use the standard `ImportString` function to import XML data from a string. This is useful when you want to generate XML data directly within *Mathematica* instead of importing the data from an external file. The `ImportString` function has the following syntax.

```mathematica
ImportString[string, format]
```
With "XML" as the import format, any XML formats that Mathematica does not recognize are returned as SymbolicXML. Here is an example of a simple XML expression converted to SymbolicXML using ImportString.

```
ImportString[
  "<person gender='male'><name>Joe Smith</name></person>" , "XML"]
```

```
XMLObject[Document][{},XMLElement[person,
  {gender → male}, [XMLElement[name, {}, {Joe Smith}]}}], {}]
```

The formats that Mathematica does recognize—NotebookML, ExpressionML, and MathML—are treated slightly differently. A NotebookML file is imported as a notebook expression. An ExpressionML file is imported as the corresponding cell expression. A MathML file is returned as the corresponding box expression. Here is an example of importing a simple MathML expression. Notice that the MathML markup is automatically converted to a Mathematica box expression.

```
ImportString["<math><mi>x</mi><mo>+</mo><mn>1</mn></math>" , "XML"]
```

```
FormBox[TagBox[RowBox[{x, +, 1}],
  MathMLPresentationTag, AutoDelete → True], TraditionalForm]
```

You can stop the automatic interpretation of imported files based on file extension by specifying "SymbolicXML" as the second argument. In this example, the imported file is returned as SymbolicXML rather than the usual box expression.

```
ImportString[
  "<math><mi>x</mi><mo>+</mo><mn>1</mn></math>" , "SymbolicXML"]
```

```
XMLObject[Document][{}, XMLElement[math, {}, {XMLElement[mi, {}, {x}],
  XMLElement[mo, {}, {+}], XMLElement[mn, {}, {1}]}}], {}]
```

You can control the various details of the import process, such as how to treat whitespace, whether to recognize entities, or whether to validate against a DTD, by specifying conversion options to the ImportString function.

- **XMLGet**

The XMLGet function can be used to import an XML document as SymbolicXML. It is very similar to the Import function, in that XMLGet[file] is equivalent to Import[file, "SymbolicXML"]. The advantage of using XMLGet is that, unlike Import, it can retrieve files over the web. Hence, it is useful if you want to import an XML file posted at a URL.
For example, the following command retrieves stock quotes from a website and returns the data as SymbolicXML.

```
XML`Parser`XMLGet[
 "http://www.webservicex.net/stockquote.asmx/GetQuote?symbol=IBM"
]
```

```
XMLObject[Document][
 {XMLObject[Declaration][Version → 1.0, Encoding → utf-8]},
 XMLElement[string, {{http://www.w3.org/2000/xmlns/}, xmlns →
 http://www.webserviceX.NET/},
 {<StockQuotes><Stock><Symbol>IBM</Symbol><Last>88.51</Last><
 Date>1/14/2003</Date><Time>4:00pm</Time><Change>+1.00</Change><
 Open>87.51</Open><High>88.51</High><Low>87.22</Low><Volume>
 7304300</Volume><MktCap>149.6B</MktCap><PreviousClose>87.51</
 PreviousClose><PercentageChange>+1.14%</PercentageChange><
 AnnRange>54.01 - 120.55</AnnRange><Earns>3.18</Earns><P-E>27.52</
 P-E><Name>INTL BUS MACHINE</Name></Stock></StockQuotes>], {}]
```

Note that XMLGet exists only in the XML`Parser` context. Hence, you must use the full name of the function, XML`Parser`XMLGet, when doing an evaluation. To use the function without the context name prefix, you must first add the XML`Parser` context to your context path.

XMLGet also accepts an optional second argument, which specifies a preinitialized parser object.

```
XMLGet[file, XMLParserObject]
```

Initializing the parser involves loading a DTD into memory either from a URL or a local file. This only needs to be done once in each kernel session. Subsequent references to the DTD are then processed much faster because the DTD has already been read and parsed.

You can also specify conversion options for XMLGet to control its behavior. The conversion options for XMLGet are the same as the ones for Import. However, the syntax for specifying conversion options is slightly different. The conversion options can be specified directly in the XMLGet function, such that

```
XMLGet[file, option1 → value1, option1 → value2,...]
```

is equivalent to

```
Import[file, "SymbolicXML", ConversionOptions → {option1 → value1, option2 → value2,...}]}
```

- **XMLGetString**

  The XMLGetString function can be used to import an XML string as SymbolicXML. It is very similar to the ImportString function, in that XMLGetString[string] is equivalent to ImportString[string, "SymbolicXML"].
Note that `XMLGetString` exists only in the `XML` context. Hence, you must use the full name of the function, `XML` `Parser` `XMLGet`, when doing an evaluation. To use the function without the context name prefix, you must first add the `XML` `Parser` context to your context path.

The advantage of using `XMLGetString` is that it accepts a preinitialized parser object as its second argument.

`XMLGetString[file, XMLParserObject]`

Initializing the parser involves loading a DTD into memory either from a URL or a local file. This only needs to be done once in each kernel session. Subsequent references to the DTD are then processed much faster because the DTD has already been read and parsed.

For example, the following command loads the XML package and then preinitializes the parser according to the XHTML DTD located at the specified URL. In this example, the preinitialized parser is given the name `XHTMLParser`.

```
XHTMLParser = XML`Parser`InitializeXMLParser[ "html",  
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"]
```

Now that the parser is initialized, we import an XML string. The string is validated with respect to the DTD stored in `XHTMLParser` by setting `"ValidateAgainstDTD" → True`. The option `Valid → True` in the output indicates that the XML input string was valid XML with respect to the XHTML DTD.

```
XML`Parser`XMLGetString[  
   "<html><head><title>test</title></head><body><p>Here is some text.</p></body></html>",  
   XHTMLParser, "ValidateAgainstDTD" → True]
```
You can also specify conversion options for `XMLGetString` to control the various details of the import process. The conversion options for `XMLGetString` are the same as those for `ImportString`. As with `XMLGet`, the conversion options can be specified directly in the `XMLGetString` function, that is:

```
XMLGetString[string, option1 → value1, option1 → value2,...]
```

is equivalent to

```
ImportString[string, "SymbolicXML", ConversionOptions → {option1 → value1, option1 → value2, ...
```

For more information on the conversion options available for importing XML, see Import Conversion Options.

### Entities and Validation

An XML document can contain any characters included in the Unicode character set. When importing an XML document into `Mathematica`, all numeric Unicode character entity references are automatically resolved into the corresponding `Mathematica` character.

```plaintext
ImportString["<hand>
  <card suit='♠' value='2'/>
  <card suit='♥' value='10'/>
  <card suit='♡' value='6'/>
  <card suit='♣' value='4'/>
  <card suit='♣' value='5'/>
</hand>", "XML"]
```

```
XMLObject[Document][{},
  XMLElement[hand, {},  
    XMLElement[card, [suit → ♠, value → 2], {}],
    XMLElement[card, [suit → ♥, value → 10], {}],
    XMLElement[card, [suit → ♡, value → 6], {}],
    XMLElement[card, [suit → ♣, value → 4], {}],
    XMLElement[card, [suit → ♣, value → 5], {}]], {}]
```

Other entities that are not built into XML are resolved according to the rules present in the DTD.
In addition to simply converting an XML document to a SymbolicXML expression, Import can validate the XML data to ensure that it conforms to a content model defined by a DTD. So long as the document is well formed, a SymbolicXML expression will be returned. If the document is not valid, warning messages will be issued and the document wrapper will indicate the invalid nature of the document with the option Valid → False.

You can control the various aspects of how entities are treated and whether the document is validated or not by using the conversion options for the Import function.

## Conversion Options

### Introduction

The standard ConversionOptions feature of Import gives you more control over the import process.

The syntax for specifying a conversion option is as follows.

```
Import[file, ConversionOptions → {option → value}]
```

Multiple conversion options can be specified by making the right-hand side of ConversionOptions a list of lists. There are nine conversion options available for importing XML data.

- "NormalizeWhitespace"
- "AllowRemoteDTDAccess"
- "AllowUnrecognizedEntities"
- "ReadDTD"
- "ValidateAgainstDTD"
- "IncludeDefaultedAttributes"
- "IncludeEmbeddedObjects"
- "IncludeNamespaces"
- "PreserveCDATASections"
NormalizeWhitespace

This conversion option controls how whitespace in the document being imported is processed. Whitespace is defined as a space, tab, or newline character.

If "NormalizeWhitespace"→True, all the whitespace inside an element is normalized. This means that all leading and trailing whitespace is stripped and any interior whitespace is reduced to a single whitespace character. "NormalizeWhitespace"→True is the default setting for this option.

If "NormalizeWhitespace"→False, then all whitespace is preserved as it was in the original XML document.

If "NormalizeWhitespace"→Automatic, then ignorable whitespace is removed and nonignorable whitespace is preserved. Whitespace is ignorable when it occurs in places where character data is not permitted according to the content model specified by the DTD. The primary use of ignorable whitespace is to add indentation for formatting purposes.

<table>
<thead>
<tr>
<th>option</th>
<th>value</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;NormalizeWhitespace&quot;</td>
<td>True</td>
<td>all the whitespace inside an element is normalized</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>all whitespace in the original XML document is preserved</td>
</tr>
<tr>
<td></td>
<td>Automatic</td>
<td>ignorable whitespace is removed and nonignorable whitespace is preserved</td>
</tr>
</tbody>
</table>

The option "NormalizeWhitespace" and its possible values.

Here is an example of whitespace handling with the default setting "NormalizeWhitespace"→True.

```
ImportString["<foo> a b </foo>", "XML"]
```

```
XMLElement[foo, {}, {a, b}], {}]
```

Setting "NormalizeWhitespace"→False preserves the whitespace as it appears in the original string.

```
ImportString["<foo> a b </foo>", "XML",
ConversionOptions→{"NormalizeWhitespace" → False}]
```

```
XMLElement[foo, {}, { a, b }], {}]
```
Note: If the option \texttt{"NormalizeWhitespace" -> False} is specified, pattern matching on the resulting SymbolicXML expression may become problematic because of the intervening whitespace.

**AllowRemoteDTDAccess**

This conversion option controls whether the parser may access the network in order to retrieve DTDs. If \texttt{"AllowRemoteDTDAccess" -> True}, the parser will automatically access the network to retrieve DTDs. If \texttt{"AllowRemoteDTDAccess" -> False}, then remote DTDs will not be retrieved, but local DTDs may still be used.

<table>
<thead>
<tr>
<th>option</th>
<th>value</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{&quot;AllowRemoteDTDAccess&quot;}</td>
<td>True</td>
<td>the parser will automatically access the network to retrieve DTDs</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>remote DTDs will not be retrieved, but local DTDs can still be used</td>
</tr>
</tbody>
</table>

- The option \texttt{"AllowRemoteDTDAccess"} and its possible values.

If \texttt{"AllowRemoteDTDAccess" -> False} and the document refers to a remote DTD, the parse will fail and an error message will be generated unless the conversion option \texttt{"ReadDTD"} is also set to \texttt{False}.

**AllowUnrecognizedEntities**

This option determines what the parser will do if undefined entity references are encountered in the XML document.

<table>
<thead>
<tr>
<th>option</th>
<th>value</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{&quot;AllowUnrecognizedEntities&quot;}</td>
<td>True</td>
<td>any undefined entities are wrapped in special entity delimiter characters and no error messages are reported</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>an error message is reported and the parse fails</td>
</tr>
<tr>
<td></td>
<td>\texttt{Automatic}</td>
<td>an error message is reported for any unrecognized entity, and the entity is wrapped in special entity delimiter characters</td>
</tr>
</tbody>
</table>

- The option \texttt{"AllowUnrecognizedEntities"} and its possible values.

The following examples contain an undefined entity called \texttt{"dogs"}. If \texttt{"AllowUnrecognizedEntities" -> False}, then an error message is reported and the parse fails.
ImportString["<!DOCTYPE root [<!ELEMENT root (#PCDATA)><!ENTITY cats 'Himalayans'>]> 
<root>
 &dogs;
</root>", "XML",
ConversionOptions -> {"AllowUnrecognizedEntities" -> False}]

XML`Parser`XMLGet::parseError :
 Entity 'dogs' was not found at Line: 1 Character: 88 in /tmp/m0000083951

Import::probableDTDError :
 There was an error parsing the XML document. This may be due to undeclared entities in the XML document, or an error loading the DTD. Check that entities are correctly declared or use the "AllowUnrecognizedEntities" conversion option.

$Failed

If "AllowUnrecognizedEntities"->Automatic, an error message is reported for any unrecognized entity, and the entity is wrapped in special entity delimiter characters. However, this does not interrupt the importing and parsing of the XML data. Automatic is the default setting for this option.

ImportString["<!DOCTYPE root [<!ELEMENT root (#PCDATA)><!ENTITY cats 'Himalayans'>]> 
<root>
 &dogs;
</root>", "XML",
ConversionOptions -> {"AllowUnrecognizedEntities" -> Automatic}]

XML`Parser`XMLGet::nonFatalParseError :
 Entity 'dogs' was not found at Line: 1 Character: 88 in /tmp/m0000093951

XMLElement[Document][[XMLElement[Doctype][root, InternalSubset -> <!ELEMENT root (#PCDATA)><!ENTITY cats 'Himalayans'>]],
XMLElement[root, {}, {&dogs&}], {}, Valid -> False]

If "AllowUnrecognizedEntities"->True, then any undefined entities are wrapped in special entity delimiter characters and no error messages are reported.
### ReadDTD

This conversion option determines whether an external DTD subset is read or not. The most important uses of a DTD are to define a content model for validation and to define character entities.

<table>
<thead>
<tr>
<th>option</th>
<th>value</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ReadDTD&quot;</td>
<td>True</td>
<td>external DTDs are read</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>external DTDs are ignored</td>
</tr>
</tbody>
</table>

The option "ReadDTD" and its possible values.

Since reading the DTD can directly affect the contents of the document, "ReadDTD"→True is the default setting for this option. Setting "ReadDTD"→False can improve the efficiency, but you should only make this change if you are certain that no information is required from the DTD. "ReadDTD" is ignored if you are using a preinitialized parser.

Setting "ReadDTD"→False is the only way to prevent the parser from attempting to read the DTD. Setting "AllowRemoteDTDAccess"→False will prevent network access and setting "ValidateAgainstDTD"→False will prevent validation from happening, but neither of these options will prevent an error caused by the parser failing to read the DTD.

### ValidateAgainstDTD

This conversion option determines whether the XML document is validated or not.
The option "ValidateAgainstDTD" and its possible values.

If the document is valid, the parser will set the `XMLObject["Document"]` option "Valid"→True. If the document is invalid, the parser will generate validity error messages and will set "Valid"→False.

The following is an example of trying to parse a document that is not valid by setting "ValidateAgainstDTD"→True. The parser generates error messages because the document is not valid.
On the other hand, if the document is valid, then no messages are generated and "Valid"→True is included in the output.

Parsing both of the examples above with "ValidateAgainstDTD"→False generates no error messages, nor does it add a "Valid" option to XMLObject["Document"].
XML\`Parser\`XMLGetString["<!DOCTYPE root [ 
  <!ELEMENT root EMPTY>
]
=root>
  <child/>
</root>", "ValidateAgainstDTD" → False] // InputForm

XMLObject["Document"][
  XMLObject["Doctype"]["root", 
    "InternalSubset" -> 
      " <!ELEMENT root EMPTY> "]},
XMLElement["root", {}], 
{XMLElement["child", {}, {}]}, {}]

XML\`Parser\`XMLGetString["<!DOCTYPE root [ 
  <!ELEMENT root EMPTY>
]
=root>
</root>", "ValidateAgainstDTD" → False] // InputForm

XMLObject["Document"][
  XMLObject["Doctype"]["root", 
    "InternalSubset" -> 
      " <!ELEMENT root EMPTY> "]},
XMLElement["root", {}], 
{XMLElement["child", {}, {}]}, {}]

With "ValidateAgainstDTD"→True, validation is attempted even if there is no DOCTYPE declaration.

XML\`Parser\`XMLGetString["<root/>", "ValidateAgainstDTD" → True]

\texttt{XML\`Parser\`XMLGetString::nonFatalParseError : Unknown element 'root' at Line: 1 Character: 8}

\texttt{XML\`Parser\`XMLGetString::nonFatalParseError : Root element different from DOCTYPE at Line: 1 Character: 8}

XMLObject[Document][{}, XMLElement[root, {}, {}], {}, Valid → False]

To have validation on only when there is a DOCTYPE Declaration, use the default setting "ValidateAgainstDTD"→Automatic. In the following example, no DTD is specified so the parser does not attempt to validate the XML string.
Here the parser tries to validate the input string because a DTD is specified explicitly.

Note that even when using a preinitialized parser, "ValidateAgainstDTD"→Automatic will not validate unless there is a DOCTYPE declaration in the document.

- **IncludeDefaultedAttributes**

  This conversion option determines whether attributes that are specified by the DTD as default attributes are included in the SymbolicXML expression. "IncludeDefaultedAttributes"→False is the default setting for this option because the default values for the attributes will be known to application developers, and, therefore, it is unnecessary to include the values in the SymbolicXML expression. Setting "IncludeDefaultedAttributes"→True will include them in the SymbolicXML expression.
The option "IncludeDefaultedAttributes" and its possible values.

Here is a simple example to illustrate how this option works. For brevity, let us assign a variable to represent the XML fragment.

```xml
xmlString = "<!DOCTYPE test [<!ELEMENT test EMPTY>
<!ATTLIST test \
	movie \t\tCDATA \t#FIXED \t\"The Lord of The Rings.\"
>]>
<test/>
";
```

This converts the XML fragment into SymbolicXML.

```mathematica
ImportString[xmlString, "SymbolicXML"]
```

```mathematica
XMLObject[Document][{XMLObject[Doctype][test,
    InternalSubset -> <!ELEMENT test EMPTY> <!ATTLIST test
      movie CDATA #FIXED 'The Lord of The Rings.' >]},
   XMLElement[test, {}, {}], {}, Valid -> True]
```

If you want the default attributes to be included in the imported SymbolicXML, set "IncludeDefaultedAttributes" → True.

```mathematica
ImportString[xmlString, "SymbolicXML",
  ConversionOptions -> {"IncludeDefaultedAttributes" -> True}]
```

```mathematica
XMLObject[Document][{XMLObject[Doctype][test,
    InternalSubset -> <!ELEMENT test EMPTY> <!ATTLIST test
      movie CDATA #FIXED 'The Lord of The Rings.' >]},
   XMLElement[test, {movie -> The Lord of The Rings.}, {}],
    {}, Valid -> True]
```

Though default attributes are defined in a DTD, including them in the expression is not the same as validation; thus, default attributes can be included even with "ValidateAgainstDTD" → False.
ImportString[xmlString, "SymbolicXML", ConversionOptions -> 
   {"IncludeDefaultedAttributes" -> True, "ValidateAgainstDTD" -> False}]

XMLObject[Document][[XMLObject[Doctype][test, 
   InternalSubset -> <!ELEMENT test EMPTY> <!ATTLIST test 
   Movie CDATA #FIXED 'The Lord of The Rings.' >]], 
   XMLElement[test, {Movie -> The Lord of The Rings.}, {}, {}]

- IncludeEmbeddedObjects

This conversion option determines the treatment of comments and processing instructions that occur inside 
the document tree.

<table>
<thead>
<tr>
<th>option</th>
<th>value</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;IncludeEmbeddedObjects&quot;</td>
<td>All</td>
<td>all the embedded objects will be included in the document tree</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>no embedded objects are included</td>
</tr>
<tr>
<td></td>
<td>Comments</td>
<td>only embedded comments are included</td>
</tr>
<tr>
<td></td>
<td>ProcessingInstructions</td>
<td>only embedded processing instructions are included</td>
</tr>
</tbody>
</table>

▲ The option "IncludeEmbeddedObjects" and its possible values.

As before, we set a variable to represent a simple XML fragment to facilitate further examples.

"IncludeEmbeddedObjects"→All will include all embedded objects in the body of the XML 
document.

xmlexample = "<!−−Comment−−> 
   <?Instruction?> 
   <root> 
   <blank/> 
   </root>";

With "IncludeEmbeddedObjects"→All, all the embedded objects will be included in the document tree.

XML`Parser`XMLGetString[xmlexample, "IncludeEmbeddedObjects" -> All]

XMLObject[Document][[XMLObject[Comment][Comment], 
   XMLObject[ProcessingInstruction][Instruction]], 
   XMLElement[root, {}, {XMLElement[blank, {}, {}]}, {}]}
Since comments and processing instructions are not intended to affect applications that use the XML
document, they are usually excluded from the document tree. Including them runs the risk of hampering
pattern matching. Hence, the default setting is "IncludeEmbeddedObjects" → None.

ProcessingInstructions and Comments

Setting "IncludeEmbeddedObjects" → "ProcessingInstructions" or
"IncludeEmbeddedObjects" → "Comments" will include only the embedded processing
instructions or comments, respectively. You can also set "IncludeEmbeddedObjects" →
{"Comments", "ProcessingInstructions"} so that a list of the embedded comments and
processing instructions will be included.

- IncludeNamespaces

This conversion option determines how namespaces are handled.
The option "IncludeNamespaces" and its possible values.

We will set a variable to represent a simple XML fragment with namespaces to facilitate our examples.

```xml
xmlString = "<root xmlns='http://mynamespace.com'
    xmlns:same='http://mynamespace.com'
    xmlns:foo='http://anothernamespace.com'>
  <child attr1='a' same:attr2='b' foo:attr3='c'/>
  <foo:child/>
  <same:child/>
</root>"
```

**True**

"IncludeNamespaces"→True reports the namespace information for each element and attribute via a list of the form \{namespace, localname\}. This form is verbose, but it is more faithful to the data model of the XML document. Additionally, this form may be easier to use for pattern matching.

```plaintext
XML`Parser`XMLGetString[xmlString, "IncludeNamespaces" -> True]
```

```
XMLElement[Document][{}, XMLHttpRequest[{{"http://mynamespace.com", root},
  {"http://www.w3.org/2000/xmlns/", same} -> "http://mynamespace.com",
  {"http://www.w3.org/2000/xmlns/", foo} -> "http://anothernamespace.com"},
  {XMLElement[{{"http://mynamespace.com", child},
    {{"attr1"} -> a, {{http://mynamespace.com\", attr2} -> b,
      {"http://anothernamespace.com", attr3} -> c}, {}},
    XMLElement[{{"http://anothernamespace.com", child}, {}, {}]},
    XMLElement[{{"http://mynamespace.com\", child}, {}, {}}}, {}]
```
False

"IncludeNamespaces"→False only reports the local name of each element or attribute. While this setting should not be used in any serious XML application, it is useful for applications that have only a single namespace because it makes the SymbolicXML expression easier to read. Note that the names of all the child elements appear to be identical when parsed this way. Consequently, this option value cannot be trusted whenever multiple namespaces are used.

```
XML`Parser`XMLGetString[xmlString, "IncludeNamespaces" → False]
```

```
XMLObject[Document][{},
XMLElement[root, {xmlns → http://mynamespace.com,
  same → http://mynamespace.com\, foo → http://anothernamespace.com},
{XMLElement[child, {attr1 → a, attr2 → b, attr3 → c}, {}],
XMLElement[child, {}, {}], XMLElement[child, {}, {}]}], {}]
```

Automatic

With the default value "IncludeNamespaces"→Automatic, the namespace is determined by means of scoping. If the namespace of an element is the same as the default namespace, then the name is represented as a single string for the local name. If the namespace of an element is different, then the name is represented by a list with the structure \{namespace, localname\}.

Here, we see that the only element whose name is represented by a two-string list is the one in namespace http://anothernamespace.com. The other elements are implicitly contained in the http://mynamespace.com namespace. Attributes are not compacted because according to the W3C specification, the attributes and the elements have different namespace scoping.

```
XML`Parser`XMLGetString[xmlString]
```

```
XMLObject[Document][{}, XMLElement[root,
  {{[http://www.w3.org/2000/xmlns/, xmlns} → http://mynamespace.com,
    [http://www.w3.org/2000/xmlns/, same} → http://mynamespace.com\,
    [http://www.w3.org/2000/xmlns/, foo} → http://anothernamespace.com},
  {XMLElement[child, {attr1 → a, [http://mynamespace.com\, attr2 → b,
    [http://anothernamespace.com, attr3 → c}, {}],
  XMLElement[[http://anothernamespace.com, child], {}, {}],
  XMLElement[[http://mynamespace.com\, child], {}, {}]}], {}]
```
Unparsed

Because the XML namespace recommendation, which extends XML, was made after the initial XML recommendation, there are some documents that use names in a non-namespace-compliant fashion. "IncludeNamespaces" → "Unparsed" is provided to allow parsing of these documents. With this value, the name is always represented as a single string: the exact string that appears in the XML file. Unless absolutely necessary, this option value should not be used.

```
XML`Parser`XMLGetString[xmlString, "IncludeNamespaces" -> "Unparsed"]
```

```
XMLElement[child, {attr1 -> a, same:attr2 -> b, foo:attr3 -> c}, {}],
XMLElement[foo:child, {}, {}],
XMLElement[same:child, {}, {}}}], {}]
```

■ PreserveCDATASections

This option controls whether the distinction between CDATA sections and regular character data is maintained on import. The value can be either True or False. CDATA sections are meant as a convenience for document authors; for most applications, they should not be treated differently from ordinary data. This means that preserving CDATA sections can make pattern matching difficult. For this reason, "PreserveCDATASections" → False is the default setting.

```
XMLElement[child, {attr1 -> a, same:attr2 -> b, foo:attr3 -> c}, {}],
XMLElement[foo:child, {}, {}],
XMLElement[same:child, {}, {}}]], {}]
```

■ The option "PreserveCDATASections" and its possible values.

Here is an example of the default behavior of PreserveCDATASections.

```
XMLElement[child, {attr1 -> a, same:attr2 -> b, foo:attr3 -> c}, {}],
XMLElement[foo:child, {}, {}],
XMLElement[same:child, {}, {}}]], {}]
```

```
XMLElement[child, {attr1 -> a, same:attr2 -> b, foo:attr3 -> c}, {}],
XMLElement[foo:child, {}, {}],
XMLElement[same:child, {}, {}}]], {}]
```

To preserve CDATA sections, specify "PreserveCDATASections" → True.
Transforming XML

Introduction

Mathematica is uniquely suited for processing symbolic expressions because of its powerful pattern matching abilities and its large collection of built-in structural manipulation functions. This section provides a few examples to illustrate the use of Mathematica for processing XML data.

When you import an arbitrary XML document into Mathematica, it is automatically converted into a SymbolicXML expression. The advantage of converting XML data into SymbolicXML is that you can directly manipulate SymbolicXML using any of Mathematica’s built-in functions.

The following command converts an XML string into a SymbolicXML expression.

```
data = ImportString["<colors><blue/><blue/><red/><blue/></colors>", "XML"]
```

Here we use a simple transformation rule to remove the unwanted “red” element from the list.

```
newdata = data /. XMLElement["red", ___] \[Rule] Sequence[]
```

You can use ExportString to convert SymbolicXML into native XML syntax, which was designed to be easy to read.
Visualizing the XML Tree

Many XML tools display an XML document as a collapsible tree, where the nodes correspond to the elements of the document. This example shows how to produce a similar visualization using cell grouping in a Mathematica notebook.

We will do this by recursively traversing the SymbolicXML expression, and for each XElement object, creating a CellGroupData expression that contains cells for each of that XElement object’s attributes and children. Each nested CellGroupData will be indented from the previous one. We start with the function to process an XElement object.

```mathematica
XMLElement[tag_, attributes_, data_, m_Integer] :=
  Cell[CellGroupData[
    Cell[TextData[
      StyleBox[tag, FontFamily -> "Swiss",
        FontWeight -> "Bold", FontSize -> 15]],
      Sequence @@ (XMLNote[#1, m] & /@ attributes,
      Sequence @@ (XMLNote[#1, m + 30] & /@ data )], Open],
    CellMargins -> {{m, Inherited}, {Inherited, Inherited}}]
```

Notice that we use the integer m for indentation. When we map XMLNote onto the XElement object’s children, we pass a larger value for m, increasing the indentation for the child elements.

A CellGroupData expression contains a list of cells. In the above definition, we have only created a cell for the XElement x. However, we have then mapped XMLNote onto the attribute list. Since this returns a list, we need to use Apply[Sequence] to the result in order to merge that list into the CellGroupData expression’s list of cells. We then do the same thing to the children of the XElement.

However, we have not yet defined XMLNote to work on attributes. The attributes of an XElement object are stored in SymbolicXML as rules. In most cases, the rule contains two strings: the key and the value. However, when namespaces are involved, the first element of the rule may be a list containing two strings: the namespace and the key. We will need to make two definitions to handle the attributes.
We will need one more definition in order to process simple SymbolicXML expressions. The text nodes in an XML document are stored simply as String objects in SymbolicXML. Thus, we need a definition that handles String objects.

```
XMLNote[s_String, m_Integer] :=
  Cell[s, Background -> GrayLevel[0.9],
    CellMargins -> {(m + 25, Inherited), {Inherited, Inherited}}]
```

With these definitions in place, we can construct a simple notebook to visualize a basic XML document.

```
d = ImportString[ExportString[Sqrt[1/2], "MathML"], "SymbolicXML"]
```

```
XMLElement[math, [http://www.w3.org/2000/xmlns/, xmlns] ->
  http://www.w3.org/1998/Math/MathML],
  [XMLElement[semantics, {}], [XMLElement[mfrac, {}], [XMLElement[mn, {}, {1}], XMLElement[msgrt, {}], [XMLElement[mn, {}, {2}]]]]],
  XMLElement[annotation-xml, {encoding -> MathML-Content},
    [XMLElement[apply, {}], [XMLElement[times, {}], {}], XMLElement[cn, {type -> integer}, {1}], XMLElement[apply, {}], [XMLElement[power, {}], {}, XMLElement[apply, {}], [XMLElement[power, {}], {}], XMLElement[cn, {type -> integer}, {2}], XMLElement[apply, {}], [XMLElement[sep, {}], {}, 2]]],
    XMLElement[cn, {type -> integer}, {-1}]]]]]], {}]
```
Because the default value of the option "IncludeEmbeddedObjects" is None, we did not alter comments, processing instructions, or anything else that would be stored in an XMLObject. Adding definitions for these is not difficult and would be a good exercise in processing SymbolicXML.

The notebook produced as the result of the above evaluation is shown here.

[XML data representation]

- **Manipulating XML Data**

XML applications are used for more than just document layout. XML is also an excellent format for storing structured data. Many commercial database vendors are now adding XML support to their products, allowing you to work with databases using XML as an intermediate format.

Mathematica is well suited for extracting and manipulating information from XML documents. To illustrate this, let us manipulate an XML file containing data on major league baseball players. You can download this file from www.mathematica-journal.com.
We first import this file into Mathematica as a SymbolicXML expression.

```
AL = Import["ALHitters.xml"];
```

Each player’s information is stored in a `PlayerRecord` element. We can easily extract this with `Cases`.

```
players = Cases[AL, XMLElement["PlayerRecord", _, _], Infinity];

Length[players]
294
```

As we can see, the XML document contains records for 294 players. Since we do not want to sift through all the American League hitters, we will just take a look at the Yankees. Inside each `PlayerRecord` element, there is a `TEAM` element that specifies a player’s team. By passing a slightly more sophisticated pattern to `Cases`, we can extract a list of all players on the Yankees team.

```
yankees = Cases[AL, 
    XMLElement["PlayerRecord", _, 
    {___, XMLElement["TEAM", _, {"NYY"}], ___}], Infinity];

Length[yankees]
21
```

The variable `yankees` now contains a list of SymbolicXML expressions for all the Yankees players. To see the syntax of each `PlayerRecord`, we extract the first element of `yankees`. 
We can see that the player’s name is stored in the `PLAYER` element of each `PlayerRecord` element. Suppose we just want to look at the names of the Yankees hitters we have already extracted. We can extract the name from one `PlayerRecord` easily enough.

```mathematica
First[Cases[First[yankees],
  XMLElement["PLAYER", _, {name_}] -> name, 2]]
```

Bernie Williams

We can then use `Map` to extract all the names from `yankees`.

```mathematica
Map[First[Cases[First[#],
  XMLElement["PLAYER", _, {name_}] -> name, 2]] &,
  yankees]
```
Map[
First[Cases[#, 
XMLElement["PLAYER", _, {name_}] \[\rightarrow\] name, 2]] &,
yankees]

{Bernie Williams, Derek Jeter, David Justice, Jorge Posada, 
Paul Oneill, Tino Martinez, Chuck Knoblauch, Jose Canseco, 
Scott Brosius, Luis Polonia, Shane Spencer, Glenallen Hill, 
Clay Bellinger, Jose Vizcaino, Luis Sojo, Chris Turner, Ryan Thompson, 
Alfonso Soriano, Lance Johnson, Felix Jose, Roberto Kelly}

Alternately, we could have used Cases on yankees with an appropriate pattern.

Cases[yankees, 
XMLElement["PlayerRecord", _, 
{___, XMLElement["PLAYER", _, {name_}], ___}] \[\rightarrow\] name]

{Bernie Williams, Derek Jeter, David Justice, Jorge Posada, 
Paul Oneill, Tino Martinez, Chuck Knoblauch, Jose Canseco, 
Scott Brosius, Luis Polonia, Shane Spencer, Glenallen Hill, 
Clay Bellinger, Jose Vizcaino, Luis Sojo, Chris Turner, Ryan Thompson, 
Alfonso Soriano, Lance Johnson, Felix Jose, Roberto Kelly}

SymbolicXML is a general-purpose format for expressing arbitrary XML data. In some cases, you may find it more useful to convert SymbolicXML into a different type of Mathematica expression. This type of conversion is easy to do using pattern matching. In the following example, we import an XML file containing data about baseball pitchers and translate the resulting SymbolicXML expression into a list of Mathematica rules. You can download the data file used here from www.mathematica-journal.com.

NL = Import["NLPitchers.xml"]; 

myTransform[XMLElement[_, _, children_]] := 
Apply[Sequence, Map[myTransform, children]]; 

myTransform[s_String] := s; 

myTransform[XMLObject["Document"][_, root_, _]] := 
{myTransform[root]};
Here, we have transformed the SymbolicXML expression for a PlayerRecord node into a simpler expression. All the information about the player is stored in a list of Mathematica Rule with Pitcher as the head.

In addition to transforming the data into a different expression syntax as above, we can also modify the data and leave the overall expression in SymbolicXML. This way we can alter our data, but still export it to an XML file for use with other applications. As an example, we will work with the salaries of our American League hitters. First, we delete any PlayerRecord entries where the salary is not available.

Next, we create a function to extract name–salary pairs from our PlayerRecord data. We will then sort these pairs by salary and look at the top ten.

```
myTransform[PlayerAttribute[XMLElement[name_, _, {value_}]]] :=
    name → value;

myTransform[XMLElement["PlayerRecord", _, children_]] :=
    Pitcher @@ Map[myTransform[PlayerAttribute[#]] &, children];

First[myTransform[NL]]

Pitcher[PLAYER → Randy Johnson, TEAM → ARI, POS → SP, GP → 35, GS → 35,
       STATUS → A, W → 19, L → 7, S → 0, IP → 248.2, ERA → 2.64, ParkAdj → 0.97441,
       PAERA → 2.57, _90ERA → 5.94, _50ERA → 4.54, RPper9inn → 3.4,
       RP → 93.0, RunRatio → 0.64, PAERAWinsOver162 → 123.461, Games → 27.6,
       PAERAWins → 21.0, PAERAloses → 6.6, AVG → 0.213, OBA → 0.276, ShutOut → 3,
       H → 202, R → 89, ER → 73, HR → 23, K → 347, WP → 5, BLK → 2, HBP → 6, E → 3,
       Salary → 13350000, DollarsPerRun → 143563, PictureIndex → 7762]

AL = AL /. XMLElement["PlayerRecord", __, {___,
        XMLElement["Salary", {}, {"#N/A"}], __}] → Sequence[];

Salaries[x_] := Cases[x,
        XMLElement["PlayerRecord", __, {___,
        XMLElement["PLAYER", __, {name_}], __, 
        XMLElement["Salary", __, {s_}], __}] →
        {name, ToExpression[s], Infinity}]
```
As a simple example of how to change the data in our SymbolicXML expression, we will create a function that doubles players’ salaries.

```
DoubleSalary[s_] := ToString[2*ToExpression[s]]
```

```
AL = AL /. XMLElement["Salary", {}, {sal_}] :> XMLElement["Salary", {}, {DoubleSalary[sal]}];
```

```
richer = Take[Sort[Salaries[AL], (#1[[2]] > #2[[2]]) &], 10] // TableForm
```

```
Albert Belle  25737340
Bernie Williams  24714286
Mo Vaughn  22333334
Raul Mondesi  20000000
Derek Jeter  20000000
Rafael Palmeiro  17241842
Ivan Rodriguez  17200000
Jim Thome  16350000
Juan Gonzalez  15000000
Kenny Lofton  15000000
```

**Visualizing XML Data**

- Creating a 3D Graphic from an XML File

The following example illustrates how to use Mathematica programming and SymbolicXML to visualize data in XML format. The molecule description markup language (MoDL) is an XML application that describes molecules. For details, see www.oasis-open.org/cover/modl.html. In this example, we convert a MoDL description of the methane molecule into a Mathematica 3D graphic.
The following is the MoDL file, which contains the description of the methane molecule. You can download this file from www.mathematica-journal.com.

```
!! methane.xml

<?xml version="1.0"?>
<modl>
<head animation="on" clockperiod="5s" stepsize="1s" loop="true">
  <meta name="title" content="Methane Dance"/>
  <DEFINE name="C">
    <atom radius="0.3" color="1 1 0"/>
  </DEFINE>
  <DEFINE name="H">
    <atom radius="0.25" color="1 0 0"/>
  </DEFINE>
  <DEFINE name="CH4">
    <molecule>
      <atom type="C" id="c1" position="0 0 0.4"/>
      <atom type="H" id="h1" position="0.5 0.1 -0.4"/>
      <atom type="H" id="h2" position="-1 0.1 0.3"/>
      <atom type="H" id="h3" position="0.3 0.7 1"/>
      <atom type="H" id="h4" position="0.2 -0.9 0.8"/>
      <bond atom1="c1" atom2="h1"/>
      <bond atom1="c1" atom2="h2"/>
      <bond atom1="c1" atom2="h3"/>
      <bond atom1="c1" atom2="h4"/>
    </molecule>
  </DEFINE>
</head>
<body>
  <molecule type="CH4" id="m"/>
  <TRANSLATE object="m" t="0.2" position="-3 1 -2"/>
  <TRANSLATE object="m" t="0.4" position="-1 0 -3"/>
  <TRANSLATE object="m" t="0.6" position="1 -1 -1"/>
  <TRANSLATE object="m" t="0.8" position="1 0 1"/>
  <TRANSLATE object="m" t="1" position="0 0 0"/>
  <ROTATE object="m" t="0.2" axis="1 0 0"/>
  <ROTATE object="m" t="0.4" axis="0 0 1" angle="-1.571"/>
  <ROTATE object="m" t="0.6" axis="0 -1 0"/>
  <ROTATE object="m" t="0.8" axis="-1 0 0" angle="-0.78"/>
  <ROTATE object="m" t="1" axis="0 0 1"/>
</body>
</modl>
```

Here, we import the file into Mathematica in the form of a SymbolicXML expression.
methane = Import["methane.xml"];  

In order to convert the resulting SymbolicXML expression into a `Graphic3D` expression, we will need the standard package `Graphics`\`Shapes`.

```mathematica
<< Graphics`Shapes`
```

The following code defines a function called `MoDLToGraphics3D` that turns the SymbolicXML expression into a `Graphics3D` expression. This function relies on a number of auxiliary functions that are defined in a later part of this section, which deals with the details of implementation.

```mathematica
MoDLToGraphics3D[modl_] := 
Block[
  
defs = Cases[modl, XMLElement["DEFINE", ___], Infinity],
  body = First[Cases[modl, XMLElement["body", ___], Infinity]],
  moldef, themols, theatoms,
  ProcessDefinition /@ defs;
  themols = 
    FlattenCases[body, 
      XMLElement["molecule", {___, "type" \[Rule] t_, ___}, ___] :> moldef[t],
      Infinity];
  theatoms = Cases[body, 
    XMLElement["atom", {___, "type" \[Rule] t_, ___, "position" \[Rule] p_, ___}, ___] :> Append[moldef[t], MolStringListToList[p]], Infinity];
  Graphics3D[MolToGraphics[\[Join][themols, theatoms], Boxed \[Rule] False]]
]
```

Applying this function to the original SymbolicXML expression generates a 3D graphic representing the methane molecule.
The details of implementation of the `MoDLToGraphics3D` function, which performs the actual transformation from SymbolicXML to a 3D graphic, are provided here.

**Implementation Details**

Notice that the original MoDL file contains a head and a body. In the head, a number of definitions are made that are used throughout the body. We have extracted these definitions into the variable `defs`. We then map the function `ProcessDefinition` across the list of definitions. The function `ProcessDefinition` constructs a Mathematica expression out of a definition and stores it in the variable `moldef`, which is dynamically scoped inside of `MoDLToGraphics3D`.

A `DEFINE` element in the head typically defines either an atom or a molecule. First, consider an atom definition.

```mathematica
First[Cases[methane,
   XMLElement["DEFINE", _, {XMLElement["atom", __]],
   Infinity]]]
```

```mathematica
XMLElement[DEFINE, {name -> C},
   {XMLElement[atom, {radius -> 0.3, color -> 1 1 0}, {}]}]
```

The `DEFINE` element essentially associates a unique key (in this case `C`) to an atom element. The `atom` element specifies its color and radius. We will turn this into a Mathematica expression of the form
Atom[radius, color]. We will then store it in moldef[name], where name is the key specified in the name attribute of the DEFINE element.

```
ProcessDefinition[XMLElement["DEFINE", {"name" -> name_String},
   {XMLElement["atom", {a__}, {}]}]] :=
   (moldef[name] = Atom[GetRad[a], GetColor[a]]);
```

In this case, a is the entire sequence of attributes of the atom element. The GetRad and GetColor functions, which we define later, extract the radius and color from this sequence. For now, assume that GetRad returns a number and that GetColor returns an RGBColor expression. We now need to process definitions of molecule elements. Like the atom definitions, molecules are given a unique key in the name attribute of the DEFINE element. The molecule element then contains atom elements and bond elements.

```
First[Cases[methane,
   XMLElement["DEFINE", __, {XMLElement["molecule", __]]],
   Infinity]]
```

```
XMLElement[DEFINE, {name -> CH4}, {XMLElement[molecule, {}],
   {XMLElement[atom, {type -> C, id -> c1, position -> 0 0 0.4}, {}],
   XMLElement[atom, {type -> H, id -> h1, position -> 0.5 0.1 -0.4}, {}],
   XMLElement[atom, {type -> H, id -> h2, position -> -1 0.1 0.3}, {}],
   XMLElement[atom, {type -> H, id -> h3, position -> 0.3 0.7 1}, {}],
   XMLElement[atom, {type -> H, id -> h4, position -> 0.2 -0.9 0.8}, {}],
   XMLElement[bond, {atom1 -> c1, atom2 -> h1}, {}],
   XMLElement[bond, {atom1 -> c1, atom2 -> h2}, {}],
   XMLElement[bond, {atom1 -> c1, atom2 -> h3}, {}],
   XMLElement[bond, {atom1 -> c1, atom2 -> h4}, {}]}]]
```

The atom elements contain three attributes: type, id, and position. The type attribute references the key from previous atom definitions. The id attribute is a unique key for this instance of the type of atom defined. In other words, what was defined previously in the atom definitions were types of atoms, like carbon or hydrogen. The atom elements inside a molecule element represent a distinct atom of some previously defined type.

The molecule element also contains bond elements. These have two attributes: atom1 and atom2. These reference the id of the atom elements in that molecule expression.

When we call ProcessDefinition on a molecule definition, we will want to store a list of the atoms and bonds in moldef.
In the definition of `ProcessDefinition`, `subdef` is the list of atoms and bonds. We map `ProcessSubdef` onto this list. That is, what we assign to `moldef[name]` is a list of the result of `ProcessSubdef` on each atom and bond element in the molecule. When `ProcessSubdef` is called on an atom, it extracts that atom’s type from `moldef`, appends the position to that expression, and stores the results under `moldef[id]`. When `ProcessSubdef` is called on a bond, it simply returns a `Bond` expression containing the positions of the two atoms it references.

We need an auxiliary function before we define `GetRad`, `GetColor`, and `GetPos`. Since positions are written as space-separated lists of numbers in MoDL, we first write a function that turns this string into a `Mathematica` list.

```mathematica
MolStringListToList[molpos_String] := Module[{stream = StringToStream[molpos], thelist}, thelist = Read[stream, {Number, Number, Number}]; Close[stream]; thelist];
```

The functions `GetPos`, `GetRad`, and `GetColor` should take in a sequence of attributes of any length and create a list from the position attribute. In SymbolicXML, attributes are stored as `Mathematica` rules. Both `GetPos` and `GetColor` will need to use `MolStringListToList`. `GetRad` needs only to convert a string to a number.

```mathematica
GetPos[___, "position" -> p_, ___] := MolStringListToList[p];
GetRad[___, "radius" -> r_, ___] := ToExpression[r];
```
GetColor[___, "color" -> c_, ___] := 
    RGBColor @@ MolStringListToList[c];

GetColor[___] := GrayLevel[1];

Here is the definition of MoDLToGraphics3D again for reference.

MoDLToGraphics3D[modl_] :=
    Block[
        {defs = Cases[modl, XMLElement["DEFINE", ___], Infinity],
        body = First[Cases[modl, XMLElement["body", ___], Infinity]],
        moldef, themols, theatoms},
        ProcessDefinition/@defs;
        themols =
            Flatten@Cases[body,
                XMLElement["molecule", {___, "type" -> t_, ___}, ___] -> moldef[t],
                Infinity];
        theatoms = Cases[body,
                XMLElement["atom", {___, "type" -> t_, ___, "position" -> p_, ___}, ___] -> Append[moldef[t], MolStringListToList[p]],
                Infinity];
        Graphics3D[MolToGraphics/@Join[themols, theatoms], Boxed -> False]
    ]

Block scopes the variables defs, body, moldef, themols, and theatoms. We already discussed moldef, and defs simply contains a list of the DEFINE elements. The function body just contains the body element of the SymbolicXML expression. That leaves themols and theatoms.

After ProcessDefinition is mapped to defs, themols is defined. Molecules in the body have a type attribute, which references the key of the molecule type defined in the head. The Cases statement then matches the molecules in the body and returns that molecule’s type definition in moldef.

body = First[Cases[methane, XMLElement["body", ___], Infinity]];
Flatten@Cases[body,
XMLElement["molecule", {___, "type" \[\rightarrow\] t_, ___}, ___] \[\[\rightarrow\] moldef[t],
Infinity]

{Atom[0.3, RGBColor[1, 1, 0], {0, 0, 0.4}],
Atom[0.25, RGBColor[1, 0, 0], {0.5, 0.1, -0.4}],
Atom[0.25, RGBColor[1, 0, 0], {-1, 0.1, 0.3}],
Atom[0.25, RGBColor[1, 0, 0], {0.3, 0.7, 1}],
Atom[0.25, RGBColor[1, 0, 0], {0.2, -0.9, 0.8}],
Bond[{0, 0, 0.4}, {0.5, 0.1, -0.4}], Bond[{0, 0, 0.4}, {-1, 0.1, 0.3}],
Bond[{0, 0, 0.4}, {0.3, 0.7, 1}], Bond[{0, 0, 0.4}, {0.2, -0.9, 0.8}]]

In our example, we only have one molecule in the body. If more molecules existed, the lists of `Atom` and `Bond` expressions would be merged together by `Flatten`. As we will see, the `Graphics3D` expression is simply made by drawing each `Atom` and `Bond`. Also, the body may contain other atoms as well. The definition of `theatoms` simply matches these elements, reads their type from `moldef`, and appends their positions. Thus, `theatoms` would contain a list of more atoms to be drawn.

The last line of `MoDLToGraphics3D` joins the `Atom` and `Bond` expressions in `themols` with the `Atom` expressions in `theatoms`. It then maps `MolToGraphics` onto this list. `MolToGraphics` is simply a function that returns a sphere for `Atom` expressions and a line for `Bond` expressions. Of course, we also need to define `MolToGraphics`. The definition is straightforward, provided you are familiar with `Graphics` and `Graphics` expressions in `Mathematica`.

```mathematica
MolToGraphics[Bond[pts__]] := {Thickness[0.01], Line[{pts}]};

MolToGraphics[Atom[r_, col_, pos_]] :=
{SurfaceColor[col], EdgeForm[],
TranslateShape[Sphere[r, 10, 10], pos]};
```

The result is a 3D graphic of methane or any other molecule you have defined in MoDL.
Comparing XSLT and Mathematica

In many situations, there is a need to transform a document from one XML format into another. One popular technique used for this purpose is XSLT transformations. However, Mathematica’s pattern matching and transformation abilities allow you to do similar transformations simply by importing the original document and then manipulating the resulting SymbolicXML expression. This section gives examples of some basic XSLT transformations and explains how to do the equivalent transformations in Mathematica.

A Simple Template

Let us consider a very simple example. Say our XML dialect uses the `code` tag to enclose program code. Typically, this is displayed in a monospace font. If we were to convert such a document to XHTML, we would probably want to use the `pre` tag for program code. The following XSLT template would do this.

```xml
<xsl:template match="code">
  <pre class="code">
    <xsl:value-of select="."/>
  </pre>
</xsl:template>
```

In Mathematica, you can create a function to do the same.

```mathematica
Show[MoDLToGraphics3D[methane]]
```
Inserting Attribute Values

Now consider an XML application that uses the `termdef` element to indicate the definition of a new term. Again, we will convert this to XHTML. We would like to anchor the definition with a named `a` element so that we can link directly to that location in the document. Assuming we have templates to handle whatever string formatting is inside the `termdef` element, we can use the following XSLT.

```xml
<xsl:template match="termdef">
    <span class="termdef">
        <a name="{@id}">
            [Definition:]&nbsp;&nbsp;
            <xsl:apply-templates/>
        </a>
    </span>
</xsl:template>
```

Notice that the `name` attribute in the resultant XHTML gets the value of the `id` attribute of the original `termdef` element. In *Mathematica*, you can do the following.

```mathematica
xslt[XMLElement["termdef", __, {"id" -> id_}, ___]] :=
    XMLElement["pre", {"class" -> "code"}, {value_}];
```

Using Predicates

Let us consider a more complicated example that involves XPath predicates. Assume we would like to match a `note` element, but only if it either has a `role` attribute set to `example` or if it contains an `eg` element as a child. Let us look at an XSLT template, and then explain what it does.

```xml
<xsl:template match="note[@role='example' or child::eg]">
    <div class="exampleOuter">
        <div class="exampleHeader">Example</div>
        <xsl:if test="*[1][self::p]">
            <div class="exampleWrapper">
                <xsl:apply-templates select="*[1]"/>
            </div>
        </xsl:if>
        <div class="exampleInner">
            <xsl:apply-templates select="eg"/>
        </div>
    </div>
</xsl:template>
```
The first `<xsl:if>` element checks to see if the first child element is a `<p>` element. If it is, then `<xsl:apply-templates>` is called on that child. This is similar to calling `Map` across the results of `Cases`. In the second `<xsl:if>` element, we check if there are `<p>` child elements beyond the first child. If so, `<xsl:apply-templates>` is called on those. Here is the corresponding `Mathematica` code.

```
xslt[
XMLElement["note", atts: {___, "role" -> "example", ___}, contents] |
XMLElement["note", atts_,
    contents: {___, XMLElement["eg", ___], ___}] :=
XMLElement["div", {"class" -> "exampleOuter"}, {
XMLElement["div", {"class" -> "exampleHeader"}, {"Example"}],
If[MatchQ[contents, {XMLElement["p", ___], ___]],
XMLElement["div",
    {"class" -> "exampleWrapper"}, xslt[contents[[1]]]],
Sequence[]],
XMLElement["div", {"class" -> "exampleInner"},
Map[xslt, Cases[contents, XMLElement["eg", ___]]]],
Sequence @@ Map[xslt, Cases[Drop[contents, 1],
XMLElement["p", ___]]]]];
```

```
{a, If[False, bob, Sequence @@ {}], b}
{a, b}
```

- **Traversing Upwards**

So far, all the examples we have given in XSLT have had a very simple implementation in `Mathematica` using SymbolicXML. In each of these cases, however, we were selecting expressions that were nested inside of the given expression. What if we wanted to select an ancestor or sibling? The following examples show how this can be done in `Mathematica`.

To clarify the problem and find a solution, we have to realize that an XML document is just a stream of characters that follows a specific syntax. Tools for manipulating XML documents treat XML according to
some model. In the case of XSLT (and its path-selection language, XPath), this model is that of a tree. Since Mathematica is a list-based language, it treats XML as nested expression lists.

While these two models are similar, they have important differences. Most notably, in nested lists you do not inherently have any concept of the containing list. Technically, any transformation that can be done with axis types, like ancestor, can be done without them. However, it is often convenient to traverse up the XML document.

Let us look at an example and then discuss how to implement the same behavior in Mathematica. While it will involve a slightly different technique than we have used above, it will nonetheless be rather simple. Consider the following XML document.

```mathematica
myxml = XML`Parser`XMLGetString["<document>
  <body>
  <p>This is a paragraph.</p>
  <p>This paragraph contains a bibliography reference:
  <bibref ref="mybibref"/>
  </p>
  </body>
  <bibliography>
  <bibl id="mybibref">My Bibliography Reference</bibl>
  </bibliography>
</document>"]
```

We will assume we simply want to have a template that matches bibref elements and replaces them with the text inside of the corresponding bibl element. In XSLT, we would write the following template.

```xml
<xsl:template match="bibref">
  <xsl:param name="ref"/>
  <xsl:value-of select="@ref"/>
</xsl:template>
```

The problem with using the same approach in Mathematica is that once we have matched a bibref element, we no longer have any information about the elements containing it. As a remedy, we will instead pass an expression containing the entire SymbolicXML expression. Notice that the bibref element in question can be obtained from
Rather than pass the `XMLElement` expression, we can pass this expression wrapped in `Hold`. That way, we can easily obtain the `bibref` element by calling `ReleaseHold`, and we can access ancestors by dropping indices from the `Part` expression. However, we will need to write a pattern matching function so that we can match these in definitions of functions.

```mathematica
XMLMatchQ[expr_, patt_] := And[
  MatchQ[expr, Hold[Part[___]]],
  MatchQ[ReleaseHold[expr], patt]];
```

```mathematica
XMLMatchQ[Hold[Part[myxml, 2, 3, 1, 3, 2, 3, 2]],
XMLElement["bibref", _, _]]
```

```mathematica
True
```

The *Mathematica* transformation then becomes relatively simple.

```mathematica
Clear[xslt]
```

```mathematica
xslt[xml_? (XMLMatchQ[#, XMLElement["bibref", {___}, ___]] &)] :=
  Module[{ref},
    ref = "ref" /. ReleaseHold[xml][[2]]; First[Cases[Part[xml, 1, 1][[2]],
      XMLElement["bibliography", _, {___},
        XMLElement["bibl", {"id" -> ref}, {s_}, ___]] -> s, 2]]]
```

```mathematica
xslt[Hold[Part[myxml, 2, 3, 1, 3, 2, 3, 2]]]
```

My Bibliography Reference
Converting a Notebook to HTML

Suppose you need to export a notebook in a specific XML format (apart from standard formats listed under the Save As Special menu). One option would be to export to NotebookML and then use some external tool (e.g., XSLT rules) to transform to the desired form of XML. But often it is just as easy to perform the manipulation within Mathematica, after first converting the notebook expression into SymbolicXML. As an example, let us recreate a simplified version of the File → Save As Special → HTML menu’s functionality.

You can download the sample notebook used in this example from www.mathematica-journal.com. The following commands import the notebook expression so we can manipulate it.

```mathematica
nb = NotebookOpen[ToFileName[Directory[], "Shakespeare.nb"]];
data = NotebookGet[nb];
```

(More notebook code follows.)
Our method will be to define a recursive function, \texttt{transform}, to process the original (notebook) expression from top to bottom, similar to the templates of XSLT. First, we establish a default definition to discard anything not explicitly matched by other patterns. (Given our “top-down” approach, perhaps this should be the last definition, but we place it here to reduce extraneous output in the intermediate results.)

\begin{verbatim}
Clear[transform];
transform[___] := Sequence[];
\end{verbatim}

The above definition uses \texttt{Sequence[]} for the following reason: since \texttt{transform} will be applied recursively, the best “null” result is one that can be dropped in the midst of a list of arguments without disrupting the syntax.

We start with the notebook expression itself.

\begin{verbatim}
transform[Notebook[contents_List, ___]DDD]:=
XMLElement["html", {}, {XMLElement["body", {}, transform/@content]}];
\end{verbatim}

The following points are worth noting.

\begin{itemize}
  \item The argument pattern must be robust enough to accept all variants. (Even though the notebook options are discarded in this conversion, a \texttt{BlankNullSequence(___)} is included to allow for them.)
  \item The only thing done with the \texttt{contents} argument is to pass it back to \texttt{transform}.
  \item The third argument is always a \texttt{List}. Forgetting this is a common pitfall.
  \item Notice that we have dropped the \texttt{head} section that is usually included in an HTML document.
\end{itemize}

The same general theme is followed for the remaining definitions.

Next, we discard cell-grouping information because the HTML has no use for it.

\begin{verbatim}
transform[Cell[CellGroupData[contents_List, ___]] :=
Sequence@@(transform/@contents);
\end{verbatim}

\textit{Mathematica} sectional heads are translated to their HTML counterparts.

\begin{verbatim}
transform[Cell[title_String, "Title"] :=
XMLElement["h1", {}, {title}];
transform[Cell[title_String, "Section"] :=
XMLElement["h2", {}, {title}];
\end{verbatim}
The Text cells introduce a complication: the contents of a Mathematica Text cell can be either a simple string or a TextData-wrapped list, if the text has additional information, such as font changes, specified. Thus, we need a definition for both cases.

\[
\text{transform[Cell[contents_String,"Text"]]} := \\
\text{XMLElement["p",{},\{contents\}] for strings,}
\]

\[
\text{transform[Cell[TextData[contents_List],"Text"]]} := \\
\text{XMLElement["p",{},\{transform/@contents\}] for lists.}
\]

Simple strings should just be passed on as is. Once again, in keeping with a top-down style, this should be placed later in the sequence of definitions. But placing it earlier helps make the intermediate results more meaningful.

\[
\text{transform[contents_String]} := \text{contents;}
\]

Finally, we deal with (simple) font changes.

\[
\text{transform[StyleBox[contents_String, FontSlant \rightarrow "Italic"]]} := \\
\text{XMLElement["i",{},\{contents\}] for italics,}
\]

\[
\text{transform[StyleBox[contents_String, FontWeight \rightarrow "Bold"]]} := \\
\text{XMLElement["b",{},\{contents\}] for bolds.}
\]

Here is the final product.

\[
\text{result = transform[data]}
\]

\[
\text{XMLElement[html,{},}
\]

\[
\text{\{XMLElement[body,{},
\text{\{XMLElement[h1,{},\{Quotations\}],}
\text{XMLElement[h2,{},\{Shakespeare\]}, XMLElement[p,{},{Now is the}
\text{winter of our discontent made glorious summer by this ,}
\text{XMLElement[i,{},\{Son of York\}],. And all the clouds that}
\text{lowered on our house, in the Ocean's deep bosum buried.}],}
\text{XMLElement[p,{},{Forgive me, though bleeding piece of}
\text{earth, that I am meek and gentle with these butchers.}
\text{Thou art the ruines of the noblest man that ever}
\text{lived in the tide of time. Woe to the hands ,}
\text{XMLElement[b,{},\{that shed this costy blood}\}],!\}]}\}]]}
\]

You can get output in a more human-readable form by using \texttt{ExportString}. 

---

The Mathematica Journal © 2003 Wolfram Media, Inc. XMLFeatures.nb 8/13/03
We can verify that this is well-formed XML.

```
SymbolicXMLErrors[result]
```

{}  

And, of course, the SymbolicXML can be exported to a file, suitable for viewing with a web browser.

```
Export["Quotes.html", result, "XML"]
```

Quotes.html

An alternative to a recursive function is to apply a list of replacement rules using ReplaceRepeated.
The two methods produce identical results.

\[\text{result1} == \text{result2}\]

True

Here is how the two methods differ.

- Since the recursion occurs implicitly via \text{ReplaceRepeated}, the latter implementation is cleaner in spots. In particular, the handling of \text{Text} cells: the \text{TextData} rule can be separated from the \text{Cell} rule. The same could be accomplished for the recursive function, but at the cost of additional patterns for the various forms that \text{contents} might take (e.g., \_List versus \_String and so on). \text{ReplaceRepeated}, by acting on all subexpressions, obviates this need.

- There is no default rule for the second method. Any unhandled parts of the original Mathematica expression will pass through unchanged, probably rendering invalid XML.

Finally, we use \text{Clear} to remove the definitions of all the symbols.
Clear[nb, data, transform, result, nb2html, result2];

\[ \text{Verifying SymbolicXML Syntax} \]

You can use the function `XML`\(^{-}\)\(\text{SymbolicXMLErrors} \)` to find errors with a SymbolicXML expression. This function returns a part specification that you can use with functions like `Part` or `Extract` to access the problematic part of your SymbolicXML expression. Let us return to the American League hitters from our earlier example.

```
AL = Import["ALHitters.xml", "SymbolicXML"];
```

We saw earlier that there were `PlayerRecord` nodes for which the `Salary` node contained the string 
 "#N/A". Suppose we decide that any player whose salary is not available to us, must be making $1,000,000. However, when we do our transformation, we make an easily overlooked mistake.

```
AL = AL /.
 XMLElement["Salary", {}, "#N/A"] :> XMLElement["Salary", {}, "1000000"];
```

We have now created incorrect SymbolicXML. We have put the string "1000000" as the third element of the `XMLElement` expression, rather than using a list containing that string. Suppose we did not know this, though, and that later on we find that `Export` produces errors when we try to write our modified XML to a file. We can use `SymbolicXMLErrors` to find the problematic expressions.

```
ALerrors = XML`SymbolicXMLErrors[AL] // Short
```

\[ \text{XMLElement::contentsList : 1000000 in XMLElement[Salary, {}, 1000000] is not a List of contents. The third item in an XMLElement must be a List of contents, even if it is an empty List.} \]

\[ \text{XMLElement::contentsList : 1000000 in XMLElement[Salary, {}, 1000000] is not a List of contents. The third item in an XMLElement must be a List of contents, even if it is an empty List.} \]

\[ \text{XMLElement::contentsList : 1000000 in XMLElement[Salary, {}, 1000000] is not a List of contents. The third item in an XMLElement must be a List of contents, even if it is an empty List.} \]

\[ \text{General::stop : Further output of XMLElement::contentsList will be suppressed during this calculation.} \]

\[ \{2, 3, 39, 3, 35\}, \{2, 3, 69, 3, 35\}, \ll \text{85\rr}, \{2, 3, 294, 3, 35\}\]
Notice that the output is preceded by several `XMLElement::name` error messages. This indicates that something is wrong with the input. The output of `SymbolicXMLErrors`, however, tells us exactly where something went wrong. `ALerrors` now contains a list of part specifications where the errors occurred.

Here we examine the first error.

```
First[ALerrors]
{2, 3, 39, 3, 35}
```

```
Part[AL, Sequence @@ First[ALerrors]]
XMLElement[Salary, {}, 1000000]
```

```
Cases[FoldList[Part, AL, First[ALerrors]],XMLElement[s_, __] \[Rule] s]
{AmericanLeagueIndividualHitters, PlayerRecord, Salary}
```

This problem is easy enough to fix.

```
Part[AL, Sequence @@ First[ALerrors]] =
Part[AL, Sequence @@ First[ALerrors]] /. 
XMLElement["Salary", p_, s_String] \[Rule] XMLElement["Salary", p, {s}]
XMLElement[Salary, {}, {1000000}]
```

We can see that the rest of the errors are of the same nature.

```
Extract[AL, Rest[ALerrors]] // Short
{XMLElement[Salary, {}, {1000000}], <<85>>, XMLElement[Salary, {}, {1000000}]}
```

By using `Map`, we can fix the rest of the errors in the same way.

```
Map[(Part[AL, Sequence @@ #] = Part[AL, Sequence @@ #] /. 
XMLElement["Salary", p_, s_String] \[Rule] XMLElement["Salary", p, {s}]) \&, 
ALerrors] // Short
{XMLElement[Salary, {}, {1000000}], <<86>>, XMLElement[Salary, {}, {1000000}]}
```

We can then verify that we have fixed the error using `SymbolicXMLErrors` again.
Exporting XML

Functions for Exporting XML

Export

You can export XML data from *Mathematica* using the standard `Export` function.

\[
\text{Export}[\text{file}, \text{expr}]
\]

\[
\text{Export}[\text{file}, \text{expr}, \text{format}]
\]

The first argument of the function specifies the file to which the data should be exported. The second argument specifies the data to be exported. For exporting XML data, this can be a SymbolicXML expression or any other *Mathematica* expression. You can also specify an optional third argument to control the form of the output. For exporting XML data, the relevant file formats are "XML", "NotebookML", "ExpressionML", "MathML", and "SVG".

With "XML" as the export format, all expressions are imported as NotebookML or ExpressionML.

\[
\text{Export}[\text{"test.xml"}, x^2, \text{"XML"}]
\]

`test.xml`

\[
\begin{align*}
&< ?xml version='1.0'?> \\
&< !DOCTYPE Expression SYSTEM 'http://www.wolfram.com/XML/notebookml.dtd'> \\
&< Expression xmlns:mathematica = 'http://www.wolfram.com/XML/' > \\
&< xmlns = 'http://www.wolfram.com/XML/' > \\
&< Function > \\
&< Symbol > Power < /Symbol > \\
&< Symbol > x < /Symbol > \\
&< Number > 2 < /Number > \\
&< /Function > \\
&< /Expression >
\end{align*}
\]

With "MathML" specified as the export format, the same expression is written out as MathML.
If `Export` is used with only two arguments, *Mathematica* determines the export format based on the filename extension. The ".xml" extension is associated with XML. Hence, `Export[filenname.xml, expr]` is equivalent to `Export[filenname.xml, expr, "XML"],` as seen in the example below.
The .mml extension is associated with MathML. Hence, `Export[filename .mml, expr]` is equivalent to `Export[filename .mml, expr, "MathML"]`, as seen in the example below.

```mathematica
Export["test.mml", x^2]
```

```xml
!! test.mml
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <annotation-xml encoding='MathML-Content'>
      <apply>
        <power>
          <ci>x</ci>
        </power>
      </apply>
    </annotation-xml>
  </semantics>
</math>
```

You can control the various details of the export process using the conversion options feature of the `Export` function.

The following commands delete the test files created by evaluating the commands in this section.

```mathematica
DeleteFile[{ToFileName[{Directory[]}, "test.xml"]}];
DeleteFile[{ToFileName[{Directory[]}, "test.mml"]}];
```

- **ExportString**

  You can convert `Mathematica` expressions into XML strings using the `ExportString` function. This function has the following syntax.

  ```mathematica
  ExportString[expr, format]
  ```

  For exporting as XML, the relevant formats are "XML", "NotebookML", "ExpressionML", "MathML", and "SVG".

  This command produces a SymbolicXML expression.
ImportString["<root><child/></root>", "SymbolicXML"]

XMLObject[Document][{},
XMLElement[root, {}],
XMLElement[child, {}]]

If the SymbolicXML expression is supplied as the first argument of ExportString, the resulting output is ordinary XML.

ExportString[%, "XML"]

<root>
  <child/>
</root>

If the first argument is some other type of expression, the output is in the form of ExpressionML.

ExportString["some text", "XML"]

<?xml version='1.0'?>
<!DOCTYPE Expression SYSTEM 'http://www.wolfram.com/XML/notebookml1.dtd'>
<Expression xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML'/>
<String>some text</String>
</Expression>

You can control various details of the export process using the conversion options feature of the ExportString function.

Conversion Options

Introduction

The standard ConversionOptions feature of Export can be used for controlling the export process. The syntax for specifying a conversion option is as follows.

Export[file, expr, format, ConversionOptions -> {option1 -> value1, option2 -> value2, ...}]

Multiple conversion options can be specified by making the right-hand side of ConversionOptions a list of lists. For exporting XML data, the following conversion options are available.

- "Annotations"
- "AttributeQuoting"
- "CheckXML"
Annotations

This conversion option controls which “annotations” are added to the output MathML. The value of this option is a list whose elements can be any combination of the following: "DocumentHeader", "XMLDeclaration", or "DOCTYPEDeclaration". The order of the elements in the list is irrelevant.

XMLDeclaration

When "XMLDeclaration" is one of the annotations, then an XML declaration is included in the header. That is, the statement `<?xml version="1.0"?>` appears in the header.

```
ExportString[x, "XML",
ConversionOptions ->
{"Annotations" -> {"DocumentHeader", "XMLDeclaration"}}]

<?xml version='1.0'?>
<Expression xmlns:mathematica='http://www.wolfram.com/XML'/
    xmlns='http://www.wolfram.com/XML'/>
<Symbol>x</Symbol>
</Expression>
```

DOCTYPEDeclaration

When "DOCTYPEDeclaration" is one of the annotations, then an XML document type declaration of the form `<!DOCTYPE ... >` appears in the header. This is a statement that specifies the DTD for the XML application in which the output is written.

```
ExportString[x, "XML",
ConversionOptions ->
{"Annotations" -> {"DocumentHeader", "DOCTYPEDeclaration"}}]

<!DOCTYPE Expression SYSTEM '
    http://www.wolfram.com/XML/notebookml1.dtd'>
<Expression xmlns:mathematica='http://www.wolfram.com/XML'/
    xmlns='http://www.wolfram.com/XML'/>
<Symbol>x</Symbol>
</Expression>
```
DocumentHeader

With the setting, "Annotations"→{"DocumentHeader", "XMLDeclaration", "DOCTYPEDeclaration"}, a header containing an XML declaration and a document type declaration for the MathML DTD are automatically added to the output.

```
ExportString[x, "XML", 
ConversionOptions -> {"Annotations" -> 
{"DocumentHeader", "XMLDeclaration", "DOCTYPEDeclaration"}]}

<?xml version='1.0'?>
<!DOCTYPE Expression SYSTEM 'http://www.wolfram.com/XML/notebookml1.dtd'>
<Expression xmlns:mathematica='http://www.wolfram.com/XML/'
xmlns='http://www.wolfram.com/XML'/>
<Symbol>x</Symbol>
</Expression>
```

When "Annotations" does not contain "DocumentHeader", then the output has no header. This is true even if the "Annotations" contains other elements such as "XMLDeclaration" or "DOCTYPEDeclaration". Thus "DocumentHeader" is an overall switch that controls whether the structure has a header or not.

```
ExportString[x, "XML", 
ConversionOptions -> 
{"Annotations" -> {"XMLDeclaration", "DOCTYPEDeclaration"}]}

<Expression xmlns:mathematica='http://www.wolfram.com/XML/'
xmlns='http://www.wolfram.com/XML'/>
<Symbol>x</Symbol>
</Expression>
```

The "Annotations"→"DocumentHeader" is useful for controlling the form of SymbolicXML generated. For instance, you can explicitly add an XElement[Document] to the SymbolicXML output, as shown below.

```
XML`MathML`ExpressionToSymbolicMathML[x + 2, 
"Annotations" -> {"DocumentHeader"}, "Formats" -> "PresentationMathML"]

XMLObject[Document][{},
{XMLElement[mrow, {}], {XMLElement[mi, {}, {x}]],
XMLElement[mo, {}, {+}], XMLElement[mn, {}, {2}]}}}], {}]
```
Here, "Annotations"→{"DocumentHeader"} is not specified so the XMLElement[Document] is omitted from the output.

```
XML`MathML`ExpressionToSymbolicMathML[
  x + 2, "Formats" → "PresentationMathML"]
```

XMLElement[math, {xmlns → http://www.w3.org/1998/Math/MathML},
  {XMLElement[mrow, {}], {XMLElement[mi, {}, {x}],
    XMLElement[mo, {}, {+}], XMLElement[mn, {}, {2}]}},]

- **AttributeQuoting**

This conversion option determines whether attribute values are enclosed by single quotes or double quotes.

<table>
<thead>
<tr>
<th>option</th>
<th>value</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;AttributeQuoting&quot;</td>
<td>'</td>
<td>attribute values are enclosed by single quotes</td>
</tr>
<tr>
<td></td>
<td>&quot;</td>
<td>attribute values are enclosed by double quotes</td>
</tr>
</tbody>
</table>

The option "AttributeQuoting" and its possible values.

With the default setting, "AttributeQuoting" → "'", attribute values are enclosed in single quotes. This ensures that there is no conflict with Mathematica strings, which are typically enclosed in double quotes.

```
ExportString[XMLElement["foo", {"color" → "blue"}, {"some data"}],
  "XML", ConversionOptions → {"AttributeQuoting" → ""}]
```

For certain applications, you might prefer attribute values to be enclosed in double quotes. You can achieve this by setting "AttributeQuoting" → "\". Note that the double quote character must be preceded by a forward slash to escape it.

```
ExportString[XMLElement["foo", {"color" → "blue"}, {"some data"}],
  "XML", ConversionOptions → {"AttributeQuoting" → "\"}]
```

- **CheckXML**

This conversion option determines whether the SymbolicXML expression being exported is first checked for errors. By default this option is set to True.
The option "CheckXML" and its possible values.

You can set this option to False if you are confident the SymbolicXML is correct, because checking the XML for errors can cause processing delays. The following example shows the delay produced by checking a small file for errors.

```
largeSymbolicXML = Import["test.xml", "SymbolicXML"];

Timing[Export["myFile", largeSymbolicXML, "XML"];
{16.0833 Second, Null}

Timing[Export["myFile", largeSymbolicXML, 
  "XML", ConversionOptions -> {"CheckXML" -> False}];]
{13.7667 Second, Null}
```

The error checking provided by this option can be quite useful because a small error may completely ruin the exported form of a large SymbolicXML expression. With the option on, small errors can often be fixed. Here the SymbolicXML has an error, but it is fixed to give a reasonable result.

```
Export["myFile", XMLElement["foo"], "XML"]
XMLElement::argrx : XMLElement called with 1 arguments; 3 arguments are expected.
XMLElement::autoFix : There is an error in the SymbolicXML expression being exported. An automated attempt has been made to fix the error, however the fix may yield XML different to that desired. The function XML`SymbolicXMLErrors will detail the errors in the SymbolicXML expression.
myFile

!!myFile
<foo/>
```

On the other hand, with "CheckXML"→False, nothing is output to the file.
Export["myFile", XMLElement["foo"],
"XML", ConversionOptions -> {"CheckXML" -> False}]
This is a long paragraph. It is made of several sentences. It keeps going and going. It doesn't look very nice on one long line, but if it is indented, it is much easier to read."

With "ElementFormatting" → None, no extra indentation is added.

With "ElementFormatting" → Automatic, elements with mixed content (strings as content) are not indented, while elements with element-only content are indented.

We saw that with "ElementFormatting" → All, long strings are line wrapped. This can be used to produce output similar to HTML. On the other hand, ElementFormatting → Automatic produces one long line of text.
This is a long paragraph. It is made of several sentences. It keeps going and going. It doesn't look very nice on one long line, but if it is indented, it is much easier to read.

Advanced users can also specify a function to determine the formatting. The function is passed a two-element list, \{namespace, localName\}. The function should return \textbf{True} when indenting is wanted, \textbf{False} when no indenting is wanted, and \textbf{Automatic} for cases where the element-only content should be indented and mixed content should not be indented.

\begin{verbatim}
myIndentFunction["", "indent"] := True;
myIndentFunction["", "noIndent"] := False;
myIndentFunction["", "autoIndent"] := Automatic;
\end{verbatim}

When exporting XML documents, it is sometimes desirable to represent special characters using named character entities. The \textbf{"Entities"} conversion option supports output of these named character entities.
The option "Entities" and its possible values.

You can also specify a list of entities as the value of this option as a list. For example, if you want to export both HTML and MathML entities, you could use this setting.

"Entities"→{"HTML", "MathML"}

If neither the "HTML" nor "MathML" settings are used, all characters are still output correctly in XML. However, they may be numeric entities or encoded in UTF-8.

Here we use the "HTML" setting to turn an α in the input into the named character entity &alpha;.

```
xmlString = ExportString[XMLElement["foo", {}, {"α"}],
   "XML", ConversionOptions→{"Entities"→"HTML"}]
<foo>&alpha;</foo>
```

You can also enter your own list of character replacement rules to be used. If you do this, then you are also responsible for including some basic escaping required by XML. For example:

```
"Entities"→{
   "&"→"&amp;",
   "<"→"&lt;",
   ">"→"&gt;",
   "\""→"&quot;",
   "'"→"&apos;",
   "h"→"&lowerCaseH;"
}
```

If you specify any value for Entities, it is your responsibility to ensure that appropriate entity declarations are present. For example, by using the "HTML" setting, you can easily generate XML with HTML entities. In this example, the Icelandic character “thorn” is exported as the corresponding character entity reference.

```
xmlString = ExportString[XMLElement["foo", {}, {"b"}],
   "XML", ConversionOptions→{"Entities"→"HTML"}]
<foo>&thorn;</foo>
```

Here, the thorn entity is not declared, and so the character is exported as a numeric character reference.
NamespacePrefixes

This option lets you generate XML markup with a specific namespace declaration and namespace prefixes. The option is specified in the form

"NamespacePrefixes" → {url → prefix}

where url and prefix are strings specifying the URL of the namespace and the namespace prefix. In the following example, the "NamespacePrefix" option is used to generate presentation markup with each MathML element having a namespace prefix "mml" associated with the MathML namespace.

```
ExportString[XMLElement["foo", {}, {"p"}], "XML"]

<foo>&#254;</foo>
```

```
■ NamespacePrefixes

This option lets you generate XML markup with a specific namespace declaration and namespace prefixes. The option is specified in the form

"NamespacePrefixes" → {url → prefix}

where url and prefix are strings specifying the URL of the namespace and the namespace prefix. In the following example, the "NamespacePrefix" option is used to generate presentation markup with each MathML element having a namespace prefix "mml" associated with the MathML namespace.

```
Export["test.mml", x^2, "MathML",
ConversionOptions -> {"Formats" -> {"PresentationMathML"},
"NamespacePrefixes" -> {"http://www.w3.org/1998/Math/MathML" -> "m"}}]

test.mml
```

```xml
<math xmlns:m='http://www.w3.org/1998/Math/MathML'>
<m:msup>
  <m:mi>x</m:mi>
  <m:mn>2</m:mn>
</m:msup>
</math>
```

■ Working with NotebookML

□ Introduction

NotebookML is an XML format for describing Mathematica notebooks. It involves a mapping of a notebook expression to a similar XML tree structure. The names of elements and attributes in NotebookML are chosen to match the names of the corresponding parts of the original notebook expression. Here is an example of a simple notebook expression.

```
Notebook[{{Cell["Here is some text!","Text"]}}]
```
Here is the same expression in NotebookML.

```xml
<Notebook>
  <List>
    <Cell>
      <String>This is a cell.</String>
      <Style>
        <String>Section</String>
      </Style>
    </Cell>
  </List>
</Notebook>
```

Note the direct correspondence between the parts of the notebook expression and their XML counterparts. The conversion is done on the FullForm of the notebook expression. For example, while a list can be denoted by `{}` in a notebook, the underlying representation of the list is still `List[...]`. Hence, in NotebookML, a list would be represented by a `List` element of the form `<List>...</List>`.

NotebookML is useful for exporting complete notebooks in XML format. However, you can also export individual cells, mathematical formulas, or other types of content in a notebook as XML, using what is called ExpressionML. This is a subset of NotebookML that enables you to save arbitrary Mathematica expressions in XML format. Here is the FullForm for a mathematical formula.

```
Sin[x + 1] // FullForm
```

```
Sin[Plus[1, x]]
```

Here is the ExpressionML representation of the same formula. You can generate the NotebookML or ExpressionML for any type of notebook expression, using `Export` or `ExportString`.

```
<Expression>
  <Function>
    <Symbol>Sin</Symbol>
    <Function>
      <Symbol>Plus</Symbol>
      <Number>1</Number>
      <Symbol>x</Symbol>
    </Function>
  </Function>
</Expression>
```

NotebookML and ExpressionML are both 100% well-formed, standards-compliant XML. Hence, they make it easy to integrate Mathematica notebooks or parts of notebooks into any XML framework or workflow. The namespace for NotebookML and ExpressionML is specified by the URL www.wolfram.com/XML. Also, the DTD for NotebookML and ExpressionML is located at the URL www.wolfram.com/XML/DTD/notebookml1.dtd.
Syntax of NotebookML

- Strings, Numbers, and Symbols

The `String`, `Number`, and `Symbol` elements are the only NotebookML elements that can directly contain character data. All other NotebookML elements are either empty elements or can only contain other elements. For example, here is a simple cell in `Text` style.

```
Cell["This is a text cell","Text"]
```

Here is the corresponding NotebookML.

```xml
ExportString[Cell["This is a text cell", "Text"], "XML"]
```

```xml
<xml version='1.0'?>
<!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookml1.dtd'>
<Cell class='Text'
    CreatedBy='Mathematica 4.2'
    xmlns:xhtml='http://www.w3.org/1999/xhtml'
    xmlns:mathematica='http://www.wolfram.com/XML'/>
   <String>This is a text cell</String>
   <Style>
     <String>Text</String>
   </Style>
</Cell>
```

Note that each string in the cell expression is represented by a `String` element in NotebookML. One benefit of using a `String` element to describe string data is that it provides a clearer indication to XML-processing applications that the whitespace inside the string data is significant.

In addition to strings, the contents of a cell can contain numbers or symbols. These structures are represented in NotebookML using the elements `Number` and `Symbol`. For example, the `Mathematica` expression

```
CellFrame[{1,0},{0,1}]
```

has the following NotebookML representation.

```xml
<CellFrame>
   <List>
      <Number>1</Number>
      <Number>0</Number>
   </List>
   <List>
      <Number>0</Number>
      <Number>1</Number>
   </List>
</CellFrame>
```
Symbol elements lack any special structure. As an example, consider the following Mathematica code fragment.

\texttt{Sin[x]}

The corresponding NotebookML is shown below. The Function element is used to enclose the names of all built-in functions.

\begin{verbatim}
<Function>
  <Symbol>Sin</Symbol>
  <Symbol>x</Symbol>
</Function>
\end{verbatim}

In notebooks, two-dimensional structures such as fractions, subscripts, superscripts, and so on, are represented by box expressions. Here is the expression for a cell containing both text and a box expression.

\begin{verbatim}
Cell[{TextData["Here is some math: ",
       Cell[BoxData[SuperscriptBox["x", "2"])]], "Text"]
\end{verbatim}

Here is the NotebookML representation of the above cell. Note the use of the TextData, BoxData, and SuperscriptBox elements in NotebookML to represent the corresponding parts of the cell expression.
ExportString[Cell[TextData[{"Here is some math: ", 
Cell[BoxData[SuperscriptBox["x", "2"]]]}], "Text"], "XML"]

<?xml version='1.0'?>
<!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookml1.dtd'>
<Cell class='Text'
    CreatedBy='Mathematica 4.2'
    xmlns:xhtml='http://www.w3.org/1999/xhtml'
    xmlns:mathematica='http://www.wolfram.com/XML/'
    xmlns='http://www.wolfram.com/XML'>
    <TextData>
        <List>
            <String>Here is some math: </String>
            <Cell>
                <BoxData>
                    <List>
                        <SuperscriptBox>
                            <String>x</String>
                            <String>2</String>
                        </SuperscriptBox>
                    </List>
                </BoxData>
            </Cell>
        </List>
    </TextData>
</Cell>

- **Options**

In *Mathematica*, various properties of notebooks, cells, and the contents of a cell are described by options. A notebook expression with options specified has the following underlying structure.

\[
\text{Notebook}[\{\text{cell1, cell2, cell3, ...}, \text{options}\}]
\]

Each option has the general form \text{name} \rightarrow \text{value}, where the left-hand side specifies the option name and the right-hand side specifies the value of that option. In NotebookML, an option is represented using the form shown below.

\[
\text{<Option>}
    \text{<Symbol>name</Symbol>}
    \text{NotebookML representation of value}
\text{</Option>}
\]
For example, this notebook expression will include a ruler in the toolbar at the top of the displayed window.

Notebook[{{Cell["Some text", "Text"], WindowToolbars -> {"RulerBar"}}]

Here is the corresponding NotebookML.

```
ExportString[Notebook[{{Cell["Some text", "Text"], WindowToolbars -> {"RulerBar"}}], "XML"]
```

```xml
<?xml version='1.0'?>
<!DOCTYPE Notebook SYSTEM 'http://www.wolfram.com/XML/notebookml1.dtd'>
<Notebook xmlns:xhtml='http://www.w3.org/1999/xhtml'
    xmlns:mathematica='http://www.wolfram.com/XML/
    xmlns='http://www.wolfram.com/XML/
<List>
    <Cell class='Text'>
        <String>Some text</String>
        <Style>
            <String>Text</String>
        </Style>
    </Cell>
</List>
<Options>
    <Option>
        <Symbol>WindowToolbars</Symbol>
        <Function>
            <Symbol>List</Symbol>
            <String>RulerBar</String>
        </Function>
    </Option>
</Options>
</Notebook>
```

Multiple options are grouped together in an Options element. Although this is not necessary, it is useful (for ease of programming and efficiency reasons) in the context of many XML applications like XSLT. For example, the cell expression:

```
Cell["Some text", FontSize->8, FontWeight="Bold"]
```

has the following representation in NotebookML.
ExportString[
  Cell["Some text", FontSize -> 8, FontWeight -> "Bold"], "XML"]

<!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookml1.dtd'>
<Cell CreatedBy='Mathematica 4.2'>
  xmlns:xhtml='http://www.w3.org/1999/xhtml'
  xmlns:mathematica='http://www.wolfram.com/XML/
  xmlns='http://www.wolfram.com/XML'/>
<String>Some text</String>
</Cell>

### Importing NotebookML

NotebookML files can be imported through the kernel using `Import`. The first argument of `Import` is the filename of the file you are importing. If the file carries the `.xml` or `.nbml` extension and is in NotebookML format, then `Import` will automatically convert the file into a notebook expression. Here is an example.

```mathematica
Import["test.xml"]
```

Notebook[Cell[Hello. This is a test notebook., Text],
  FrontEndVersion -> 4.2 for Microsoft Windows,
  ScreenRectangle -> {{0., 1024.}, {0., 695.}}, WindowSize -> {496., 599.},
  WindowMargins -> {{18., Automatic}, {Automatic, 23.}}]

You can also specify the import format explicitly, by using "SymbolicXML" or "NotebookML" as the second argument to `Import`. This is useful when the file you are importing does not have a `.xml` or `.nbml` extension to indicate the nature of its contents.

The import process is much faster, especially for large files, if you specify "NotebookML" as the import format. However, you should only use this form if you are sure that the file is a NotebookML document.

```mathematica
Import["test.xml", "NotebookML"]
```
You can also import the file as SymbolicXML using the following command.

```
Import["test.xml", "SymbolicXML"];
```

## Processing NotebookML

### Converting between NotebookML and SymbolicXML

Suppose you have written a technical paper in *Mathematica* and now want to submit the paper to a journal that only accepts submissions in a specific XML format, say DocBook. You, therefore, need to transform your notebook into DocBook format. One way to do this is to use XSLT to transform the NotebookML output from *Mathematica* into DocBook.

![XSLT Diagram](image)

However, you can easily perform the same transformation completely within *Mathematica* by using SymbolicXML as an intermediary. *Mathematica's* pattern matching and symbolic processing can make the task quite easy. A schematic of the process might look like the following.

![SymbolicXML Diagram](image)

Here is a simple example of the type of manipulations you can perform. This command defines a rule for replacing tab characters ('\t') with four nonbreaking spaces. This is something one might want to do before presenting work on the web.

```mathematica
fixCharRule = {XMLElement["String", a_, {s_String}] :> XMLElement["String", a, {StringReplace[s, \"\t\" -> "\t\n"]}]};
```

This applies the rule to a notebook called `nb` and exports the result as an XML file.
nb = Notebooks[[2]]
NotebookObject[<<test.nb>>]

Export["changedchars.xml", ToSymbolicXML[nb] /. fixCharRule]
changedchars.xml

■ Preprocess

Preprocessing allows any function to be applied to the notebook before the XML conversion functions are applied to it. For example, one might want to ensure that all closed cell groups are open before proceeding with the conversion. This can be done with the following command.

openCellGroups[eachCell_] := eachCell //.
   {CellGroupData[cells_, Closed] -> CellGroupData[cells, Open]};

One could then preprocess the notebook by using openCellGroups in the following way.

Export["test2.xml", openCellGroups[NotebookGet[nb]], "XML"]
test2.xml

■ Postprocess

Postprocessing provides a handle on the SymbolicXML object, representing the NotebookML before it is exported. This gives you total freedom to manipulate the SymbolicXML in any fashion you choose. However, it is your responsibility to output legitimate SymbolicXML. Mathematica will try to automatically correct minor errors but if there is a more serious error, the export process will fail.

Here, we use a postprocessing rule to replace characters.

changeChars[XMLObject["Document"][pre_, root_, postopts___]] :=
   XMLObject["Document"][pre, changeChars[root], postopts];

changeChars[XMLElement[name_, atts_, contents_]] :=
   XMLElement[name, atts, Flatten[changeChars /@ contents]];
changeChars[s_String] := Characters[s] /. charRules

charRules = Dispatch[{"a" -> "Mathematica is cool"}];

You could then postprocess the symbolic NotebookML by using changeChars in the following way.

Export["test3.xml", 
  changeChars[XML`ToSymbolicXML[NotebookGet[nb]]], "XML"]

test2.xml

Of course, you can use both pre- and postprocessing together.

Export["test4.xml", changeChars[
  XML`ToSymbolicXML[openCellGroups[NotebookGet[nb]]]], "XML"]

test2.xml

Exporting NotebookML

Introduction

You can export notebooks as NotebookML using the Export function. Just as when exporting a notebook as HTML, when you export a notebook as NotebookML, graphics and boxes are automatically saved as GIF images.

The first argument for Export specifies the file to export to, and the second argument specifies the data to be exported. If you specify the file extension of the exported document as ".xml", then Export will automatically generate NotebookML. For example, the following will export the current notebook as a file named "anothertest.xml" in NotebookML.

Export["anothertest.xml", InputNotebook[]]

another.xml

If you export a Cell with the file extension .xml, Export will automatically generate ExpressionML. The following example will export the cell as ExpressionML.
### Conversion Options for Export

You can use `ConversionOptions` to control the behavior of `Export` and `ExportString`. There are four conversion options available for exporting NotebookML.

- **"Annotations"** includes or excludes XML declarations, DOCTYPE declarations, and style advisories in any desired combination.
- **"BoxFormats"** specifies the export formats for various typeset box objects in the document.

```plaintext
Export["celltest.xml", Cell["Some text", "Text"]]
celltest.xml

If the filename for the exported data does not carry the .xml extension, then you must specify "XML" or "NotebookML" as the second argument for `Export`. The data for export is then specified as the third argument. Here is an example.

```plaintext
Export["moretests", InputNotebook[], "XML"]
moretests

You can also export data using `ExportString`. This will return NotebookML for a notebook expression or ExpressionML for a cell expression without assigning the output a particular filename. The first argument for `ExportString` is the data to be exported, and the second argument is the desired export format. In this example, the notebook expression is exported as NotebookML.

```plaintext
ExportString[Notebook[{Cell["Some text", "Text"]}], "NotebookML"]

```
• "GraphicsFormats" specifies the export formats for graphics objects in the document.
• "Stylesheets" will associate a style sheet (CSS or XSLT) for presentation on the web.

Annotations

The Annotations option takes a list whose elements can be any combination of the following values: "DocumentHeader", "DOCTYPEDeclaration", "XMLDeclaration", and "StyleAdvisories". "DocumentHeader" determines if any header information should be added at the beginning of the NotebookML file. The "DOCTYPEDeclaration" and "XMLDeclaration" settings only take effect if "DocumentHeader" is specified as one of the annotations.

If "DOCTYPEDeclaration" is included in the list of annotations, then a DOCTYPE declaration is included in the header of the exported document. The default setting for "Annotations" includes "DOCTYPEDeclaration", as shown in the following example.

ExportString[Cell["Some text", "Text"], "XML"]

```xml
<?xml version='1.0'?>
<!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookml1.dtd'>
<Cell class='Text' 
  CreatedBy='Mathematica 4.2'
  xmlns:xhtml='http://www.w3.org/1999/xhtml'
  xmlns:mathematica='http://www.wolfram.com/XML/
  xmlns='http://www.wolfram.com/XML'/>
<String>Some text</String>
</Cell>
```

If you exclude "DOCTYPEDeclaration" from the list of Annotations values, the DOCTYPE declaration is omitted from the header of the exported document. Excluding the DOCTYPE declaration can sometimes shorten processing time taken to read the resulting XML document into an application, because the parser does not have to reference a DTD.

If "XMLDeclaration" is included in the list of annotation values, an XML declaration is also included in the header of the exported document. The default setting for "Annotations" includes "XMLDeclaration", as shown in the following example.
ExportString[Cell["Some text", "Text"], "XML"]

<?xml version='1.0'?>
<!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookml1.dtd'>
<Cell class='Text' CREATEDBY='Mathematica 4.2'
xmlns:xhtml='http://www.w3.org/1999/xhtml'
xmlns:mathematica='http://www.wolfram.com/XML/'
xmlns='http://www.wolfram.com/XML'/>
<String>Some text</String>
</Cell>

If "XMLDeclaration" is excluded from Annotations, it is also omitted from the header of the exported document. Excluding the XML declaration is useful if a user wishes to create a NotebookML fragment for insertion in another XML document.

ExportString[Cell["some text", "Text"], "XML", ConversionOptions ->
{"Annotations" -> {"DOCTYPEDeclaration", "StyleAdvisories"}}]

<Cell class='Text' CREATEDBY='Mathematica 4.2'
xmlns:xhtml='http://www.w3.org/1999/xhtml'
xmlns:mathematica='http://www.wolfram.com/XML/'
xmlns='http://www.wolfram.com/XML'/>
<String>some text</String>
</Cell>

If "StyleAdvisories" is included in the list of annotations, class attributes are returned for Cells and StyleBoxes that have styles associated with them. The default setting for "Annotations" includes "StyleAdvisories", as shown in the following example.
ExportString[Cell["Some text", "Text"], "XML"]

<?xml version='1.0'?>
<!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookml1.dtd'>
<Cell class='Text'
    CreatedBy='Mathematica 4.2'
    xmlns:xhtml='http://www.w3.org/1999/xhtml'
    xmlns:mathematica='http://www.wolfram.com/XML/
                xmlns='http://www.wolfram.com/XML'>
    <String>Some text</String>
    <Style>
        <String>Text</String>
    </Style>
</Cell>

If you exclude "StyleAdvisories" from Annotations, the class attributes associated with Cell are omitted.

ExportString[Cell["Some text", "Text"], "XML", ConversionOptions ->
{"Annotations" -> {"DOCTYPEDeclaration", "XMLDeclaration"}}]

<Cell CreatedBy='Mathematica 4.2'
    xmlns:xhtml='http://www.w3.org/1999/xhtml'
    xmlns:mathematica='http://www.wolfram.com/XML/
                xmlns='http://www.wolfram.com/XML'>
    <String>Some text</String>
    <Style>
        <String>Text</String>
    </Style>
</Cell>

**BoxFormats**

**BoxFormats** exports box data as NotebookML, GIF, or MathML. This option can be set to one or more of the following values: Automatic, "GIF", or "MathML". The default setting is "BoxFormats"→Automatic.

With "BoxFormats"→Automatic, the boxes will be losslessly exported as NotebookML. This is the only box format setting that is guaranteed to be lossless.

With "BoxFormats"→"GIF", the box data will be represented as a GIF file. The exported NotebookML document will reference the exported GIF file by using an XHTML tag such as `<img src="file.gif">`. 
With "BoxFormats"→"MathML", the box data will be exported as MathML, which will be embedded inside the NotebookML.

If BoxFormats has more than one value, then the BoxData parent element will have multiple child elements associated with it, one element per format.

For example, given some BoxData expression:

BoxData[{{}]

and the following values for BoxFormats

BoxFormats → {Automatic, GIF, MathML}

a NotebookML expression of the following form is generated.

```xml
<BoxData>
  <List>
    ...
  </List>
  <xhtml:img src="Images/myFile_1.gif"/>
  <math>
    ...
  </math>
</BoxData>
```

**GraphicsFormats**

The "GraphicsFormats" option changes the format of notebook graphics into another format for export. This is often useful since some external applications do not support Mathematica notebook graphics. This option can be set to one or more of the following values: Automatic, "Bitmap", "GIF", "Metafile", "PICT", "PostScript", or "QuickTime".

Automatic is the default setting for this option. It is necessary to set "GraphicsFormats"→Automatic to ensure lossless import and export of a notebook.

For the "Bitmap", "Metafile", "PICT", "PostScript", and "QuickTime" values, the original notebook graphic will be converted to the specified graphic type and then exported.

For the "GIF" setting, an external GIF file is created from the original notebook graphic. An `<xhtml:img src>` element is inserted in the exported file. Here is an example.

"GraphicsFormats" → {Automatic, "GIF"}

```xml
<GraphicsData>
  <PostScript>({some postscript data...})</PostScript>
  <xhtml:img src="Images/myFile_1.gif"/>
</GraphicsData>
```
If "GraphicsFormats" has more than one value, then multiple child elements are created for each specified format inside the GraphicsFormats parent element.

**Stylesheets**

You can associate your NotebookML documents with a style sheet (CSS or XSLT) by using the Stylesheets option. This option takes a list of rules that represent the pseudo-attributes in the XML-style sheet processing instruction. For more information about the various pseudo-attributes, see: www.w3.org/TR/xml-stylesheet.

The following is a list of rules for "Stylesheets".

- href
- type
- title
- media
- charset
- alternate

Each of these rules must have a string as its value. For example, the following option

```
"StyleSheets" -> {"href" -> "mystyle.css", "type" -> "text/css"}
```

would result in the following XML statement.

```xml
<?xml-stylesheet href="mystyle.css" type="text/css"?>
```

Here is an example that uses this conversion option.

```
ExportString[Cell["Some text"], "XML", ConversionOptions -> 
{"StyleSheets" -> {"href" -> "mystyle.css", "type" -> "text/css"}}]
```

```
<?xml version='1.0'?>
<!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookml1.dtd'>
<?xml-stylesheet href="mystyle.css" type="text/css"?>
<Cell CreatedBy='Mathematica 4.2'
    xmlns:xhtml='http://www.w3.org/1999/xhtml'
    xmlns:mathematica='http://www.wolfram.com/XML/'
    xmlns='http://www.wolfram.com/XML'/>
<String>Some text</String>
</Cell>
```

"Stylesheets" can also take a list of rules, each of which may take a list of values as a sublist. Each sublist corresponds to one xml-stylesheet processing instruction.
Using a CSS style sheet, NotebookML can be displayed in many current generation browsers. For example, *Internet Explorer* 5 or later and *Netscape* 6 or later have built-in support for CSS. The style sheet can even mimic the behavior of *Mathematica*’s environments (like Working, Printout, Presentation, etc.). To save a notebook as NotebookML with a CSS style sheet, use the `Export` function with a conversion option of "StyleSheets" pointing to the relevant style sheet. Here is an example.

```mathematica
Export("NotebookML.xml", InputNotebook[],
ConversionOptions -> {"StyleSheets" -> 
  {"type" -> "text/css", "href" -> "css/presentation.css"})
```

If you save a notebook as NotebookML and use a CSS style sheet that contains definitions for the various elements, the resulting file can be rendered in a web browser. The advantage of this approach (instead of simply converting the notebook to HTML) is that you only need to create a single document, which can be viewed either in web browsers or in *Mathematica*. A non-*Mathematica* user can view the document in any web browser. But a *Mathematica* user will be able to open and edit the document as a notebook, evaluate the input, manipulate the graphics, and so on.
Working with MathML

Introduction

MathML is an XML-based markup language for representing mathematics. It was developed by the W3C to provide an effective way to display math in web pages and to facilitate the transfer and reuse of mathematical content between applications. MathML can encode information about both the meaning and the appearance of mathematical notation. For example, a MathML equation can be copied out of a web page and directly pasted into an application like Mathematica for evaluation.

Wolfram Research was a key participant in the development of MathML and is committed to supporting this important web technology. Mathematica includes full support for MathML 2.0, the latest version of the language. You can import MathML equations into a Mathematica notebook and evaluate them, or export equations from a notebook as MathML and paste them into an HTML document for viewing in a web browser. There are also several kernel commands for converting between MathML and the boxes and expressions used internally by Mathematica to represent mathematics.

Syntax of MathML

Overview

Since it is an XML application, the syntax rules of MathML are defined by the XML specification. Each MathML expression consists of a series of elements, written in the angle bracket syntax similar to HTML. Each element can take several attributes. The allowed elements and attributes are determined by the MathML DTD.

All MathML elements fall into one of three categories: interface elements, presentation elements, and content elements. Interface elements, such as the top-level math element, determine how a MathML expression is embedded in other XML documents.

Presentation elements encode information about the appearance of a mathematical expression, that is, its two-dimensional structure. For example, the mrow, mfrac, msqrt, and msub elements represent a row, a fraction, a square root, and a subscripted expression, respectively.

Content elements encode information about the logical meaning of a mathematical expression. For example, plus and sin represent addition and the trigonometric sine function, and apply represents the operation of applying a function.

A given equation can be represented in several different ways in MathML.

- Presentation elements only. This type of markup is called presentation markup. It is useful in situations where only the display of mathematics is important. For example, you can use presentation markup for displaying equations in a web page.
Content elements only. This type of markup is called content markup. It is useful in situations where it is important to encode mathematical meaning. For example, you can use content markup in a web page to allow students to copy equations from their browser and paste it into Mathematica for evaluation.

A combination of content and presentation elements. This type of markup is called combined markup and is used when you want to encode both the appearance and meaning of equations. For example, you can use combined markup to specify a nonstandard notation for a common mathematical construct or to associate a specific mathematical meaning with a certain type of notation that usually has a different meaning.

Using Mathematica, you can generate presentation markup, content markup, or combined markup for any equation.

**Presentation MathML**

Presentation MathML consists of about 30 elements and 50 attributes, which encode the visual two-dimensional structure of a mathematical expression. For example, the Mathematica typeset expression \( x + 1 \) would have the following MathML representation.

\[
x + 1
\]

```xml
<math>
  <mrow>
    <mi>x</mi>
    <mo>+</mo>
    <mn>1</mn>
  </mrow>
</math>
```

The entire expression is enclosed in a `math` element. This must be the root element for every instance of MathML markup. The above example also uses four other presentation elements.

- **mrow** displays its sub-elements in a horizontal row.
- **mi** represents an identifier, such as the name of a function or a variable.
- **mo** represents an operator or a delimiter.
- **mn** represents a number.

Identifiers, operators, and numbers are each represented by a different type of element because each type of object has slightly different typesetting conventions for fonts, spacing, and so on. For example, variables are typically rendered in an italic font, numbers are displayed in a normal font, and operators are rendered with extra space around them, depending on whether they occur in a prefix, postfix, or infix position.

In addition to the `mi`, `mn`, and `mo` elements, there are presentation elements corresponding to common notational structures, such as fractions, square roots, subscripts, superscripts, and matrices. Any given formula can then be represented by decomposing it into its constituent parts and replacing each notational
construct by the corresponding presentation elements. For example, the typeset expression \( \frac{\sqrt{x}}{y^2 - 1} \) would have the following MathML representation.

\[
\frac{\sqrt{x}}{y^2 - 1}
\]

Here, the `mfrac`, `msqrt`, and `msup` elements represent a fraction, a square root, and a superscripted expression. Each of these elements takes a fixed number of child elements, which have a specific meaning based on their position. These child elements are called `arguments`. For example, both the `mfrac` and `msup` elements take two arguments with the following syntax.

```xml
<math>
  <mfrac>
    <msqrt>
      <mi>x</mi>
    </msqrt>
    <mrow>
      <msup>
        <mi>y</mi>
        <mn>2</mn>
      </msup>
      <mo>-</mo>
      <mn>1</mn>
    </mrow>
  </mfrac>
</math>
```

As before, the `mrow` element is used to enclose other elements that need to be displayed in a horizontal row. For example, the typeset expression \( \int_{0}^{\infty} e^{-x} \, dx \) would have the following MathML representation.

```xml
<math>
  <mrow>
    <msubsup>
      <mo>&int;</mo>
      <mn>0</mn>
      <mi>&inf;</mi>
    </msubsup>
    <msup>
      <mi>&exp;</mi>
      <mrow>
        <mo>-</mo>
        <mi>x</mi>
      </mrow>
    </msup>
  </mrow>
</math>
```
Here, the limits of the integral are shown using the presentation element \texttt{msubsup}, which takes three arguments, with the following syntax.

\[
<\texttt{msubsup} \ base \ subscript \ superscript \ /\texttt{msubsup}>
\]

Another notable feature is that the symbols representing the integral sign, the exponential, and the differential \texttt{d} are represented using the character entities \texttt{\textbackslash int}, \texttt{\textbackslash exp}, and \texttt{\textbackslash dd}, respectively. These are among approximately two thousand special symbols defined by the MathML DTD. These can be included in a document using a named entity reference or a character entity reference that uses the Unicode character code for that symbol.

The \texttt{mstyle} element is used for applying styles to an equation. Any attributes specified in an \texttt{mstyle} element are inherited by all its child elements. Hence, you can use this element to specify properties like the font size and color for an equation, as in the example below. Note the use of the entity \texttt{\$InvisibleTimes;} to denote multiplication.

\[
x^2 - 2x + 1
\]

The above examples are intended only to illustrate how presentation markup works through a sampling of some of its elements. To see a complete listing of all the presentation elements and attributes, see the MathML specification at www.w3.org/TR/MathML2.
Content MathML

Content MathML consists of about 140 elements and 12 attributes, which encode the logical meaning of a mathematical expression. The content elements $\text{ci}$ and $\text{cn}$ are used to represent identifiers and numbers, respectively. They are analogous to the $\text{mi}$ and $\text{mn}$ elements in presentation markup. For example, the typeset expression $x + 1$ would have the following content MathML representation.

\[
x + 1
\]

\[
<\text{math}>
<\text{apply}>
<\text{plus}/>
<\text{ci}>x</\text{ci}>
<\text{cn}>1</\text{cn}>
</\text{apply}>
</\text{math}>
\]

The $\text{apply}$ element is used to apply operators or functions to expressions. The first argument of the $\text{apply}$ element is usually an empty element indicating an operator or function. The remaining arguments represent one or more expressions to which the first argument is applied. In the above example, the first argument of the $\text{apply}$ function is the empty element $\text{plus}$, which denotes addition.

The $\text{type}$ attribute of $\text{cn}$ describes the type of number encoded. It can take values $\text{real}$, $\text{integer}$, $\text{rational}$, $\text{complex-polar}$, $\text{complex-cartesian}$, and $\text{constant}$. The empty element $\text{sep}$ is used to separate different parts of a number, such as the numerator and denominator of a fraction or the real and imaginary parts of a complex number. For example:

\[
\frac{1}{2}
\]

\[
<\text{math}>
<\text{cn type="rational">1<sep/>2</cn>
</\text{math}>
\]

\[
3 + 4i
\]

\[
<\text{math}>
<\text{cn type="complex-cartesian">3<sep/>4</cn>
</\text{math}>
\]

The majority of content elements are empty elements representing specific operators or functions. The various elements are organized into groups named after the following specific elementary subfields of mathematics:

- Arithmetic, Algebra, and Logic
- Elementary Functions
There are elements corresponding to most operators and functions that are encountered in high school mathematics. For example, basic arithmetic operators are represented by \[\text{plus}, \text{minus}, \text{times}, \text{divide},\] and \[\text{power}.\]

\[
\frac{\sqrt{x}}{y^2 - 1}
\]

Integrals are specified using the \[\text{int}\] element. The variable of integration is represented using the element \[\text{bvar}\]. The upper and lower limits of integration are usually specified using the elements \[\text{lowlimit}\] and \[\text{uplimit}\].

\[
\int_{a}^{b} x \, dx
\]
The `interval` element is used to specify closed and open intervals. It takes the attribute `closure`, which can take the values `closed`, `open`, `closed-open`, and `open-closed` corresponding to the four types of intervals possible. The default value for `closure` is `closed`.

\[ [a, b] \]

\[
\int_a^b x \, dx
\]

You can also use the `interval` element to specify the limits of a definite integral as an alternative to using `uplimit` and `lowlimit`.
The \textbf{matrix} and \textbf{matrixrow} elements are used to represent a matrix and a row of a matrix, respectively. The \textbf{eq} element is used to express equality.

\[ M = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \]

The above examples are intended only to illustrate how content markup works through a representative sampling of some of its elements. To see a complete listing of all the content elements and attributes, see the MathML specification at www.w3.org/TR/MathML2.

\section*{Importing MathML}

There are two ways to import MathML equations into \textit{Mathematica}.

- Copy and paste MathML equations from another application, such as a web browser, directly into a notebook. When you paste a valid MathML expression into a notebook, \textit{Mathematica} brings up a dialog box asking if you want to paste the literal markup or interpret it. If you choose to interpret the markup, it is automatically converted into a \textit{Mathematica} expression.

- Use the \texttt{Import} command with "\texttt{MathML}" as the import format.
By default, MathML markup is imported as a Mathematica box expression. You can convert the boxes into an expression using the ToExpression command.

```
ToExpression[%]
x^2
```

### MathML Conversion Options

The standard ConversionOptions feature of the Import function can be used for greater control over the export process. The syntax for specifying a conversion option is as follows.

```
Import[file, expr, "MathML", ConversionOptions -> {option1 -> value1, option1 -> value2, ...}]
```

For more information on the conversion options available for importing MathML, see XML Import Conversion Options.

### Generating MathML

Mathematica includes several functions for generating MathML from the boxes and expressions used internally by Mathematica to represent equations. You can enter an equation in a notebook using palettes, menus, or keyboard shortcuts and then convert it into MathML using one of these conversion functions. All the MathML conversion functions are located in the XML`MathML` context.

You can use BoxesToMathML to generate MathML from a box structure. By default, this generates presentation markup only.

```
XML`MathML`BoxesToMathML[SuperscriptBox["x", "2"]]
```

```
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <msup>
    <mi>x</mi>
    <mn>2</mn>
  </msup>
</math>
```
Alternatively, you can use `ExpressionToMathML` to convert a typeset equation in a notebook into MathML. By default, this generates combined markup with both the presentation markup and content markup for the equation enclosed in a `semantics` element.

```xml
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <annotation-xml encoding='MathML-Content'>
      <apply>
        <power>
          <ci>x</ci>
          <cn type='integer'>2</cn>
        </power>
      </apply>
    </annotation-xml>
  </semantics>
</math>
```

The `annotation-xml` element is used to provide additional information of a type specified by its encoding attribute. Here, the encoding attribute has the value "MathML-Content", indicating that the `annotation-xml` element contains content MathML.

You can use the option "Formats" to generate either presentation MathML or content MathML only. You can also set the option "Annotations"→{} to suppress the header information.

```xml
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <mrow>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <mo>+</mo>
    <mn>1</mn>
  </mrow>
</math>
```
Note that `ExpressionToMathML` evaluates its first argument before converting it to MathML. Hence, if you supply an expression that can be simplified on evaluation, you may get unexpected results. For example, suppose you want to generate the presentation markup for the following definite integral.

\[
\int_{0}^{\infty} e^{-x} \, dx
\]

1

Because the integral evaluates to give the result 1, the following command generates the MathML representation of 1 instead of the integral.

```math
XML`MathML`ExpressionToMathML[\int_{0}^{\infty} e^{-x} \, dx, "Annotations" \rightarrow \{\}, "Formats" \rightarrow \{"PresentationMathML"\}]
```

<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <mn>1</mn>
</math>

To get the MathML representation of the integral, you must force the integral to remain unevaluated by wrapping the `Unevaluated` function around it.
XML`MathML`ExpressionToMathML[
Unevaluated[\int_0^\infty e^{-x} \, dx], "Annotations" \rightarrow \{
"Formats" \rightarrow \{"PresentationMathML"\}\}
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <mrow>
    <msubsup>
      <mo>\int</mo>
      <mn>0</mn>
      <mi>\infty</mi>
    </msubsup>
    <msup>
      <mi>e</mi>
      <mo>-</mo>
      <mi>x</mi>
    </msup>
    <mi>dx</mi>
  </mrow>
</math>

**Setting Options**

These are all the functions that generate MathML as output.

- `BoxesToMathML`
- `ExpressionToMathML`
- `BoxesToSymbolicMathML`
- `ExpressionToSymbolicMathML`
- `Import`

These functions all accept the following options.

- "Annotations"
- "Formats"
- "NamespacePrefixes"
Using these options, you can control various features of the generated MathML, such as including an XML
declaration or DTD declaration, generating presentation markup, content markup, or both, and using an
explicit namespace declaration and namespace prefix.

You can specify the options explicitly each time you evaluate one of the MathML functions. Alternatively,
you can use the `SetOptions` command to change the default values of the options for a particular
function. The option values you set are then used for all subsequent evaluations of that function.

For example, evaluating the following command ensures that the output for all subsequent evaluations of
`ExpressionToMathML` will, by default, generate presentation MathML only and omit the header
information in the output.

```
SetOptions[XML`MathML`ExpressionToMathML,
  "Annotations" → {}, "Formats" → {"PresentationMathML"}]
```

### Exporting MathML

#### Introduction

You can use `Mathematica`’s sophisticated typesetting capabilities to create properly formatted equations
and then convert them into MathML for display on the web. There are several ways to export mathematical
expressions from a `Mathematica` notebook as MathML.

- Use the `Edit` > `Copy As` > `MathML` menu command. This copies the selected expression into the
  Clipboard in MathML format. This is a convenient way to copy a specific mathematical formula
  from a notebook and paste it into an HTML document.

- Use the `File` > `Save As Special` > `HTML+MathML` menu command. This converts your entire
  notebook into HTML with all equations in the notebook saved as MathML. The equations are
  embedded in the HTML file in the form of MathML “data islands,” which can be displayed by a
  web browser, either directly or using a special plug-in.

- Use the `File` > `Save As Special` > `XML (NotebookML+MathML)` menu command. This converts
  your entire notebook into NotebookML with all equations in the notebook saved as MathML.

- Use the `Export` function, with "MathML" as the export format or with .mml as the file
  extension.

```
Export["test", x^2, "MathML"]
```

```
test
```
The resulting output contains both presentation markup and content markup for the expression, enclosed in a `semantics` element. You can choose to generate either presentation markup or content markup by changing the value of the option "Formats". Note that the `xmlns` attribute is added to the top-level `math` element to provide information about the namespace of the enclosed elements.
Symbols for MathML Elements

Certain content elements in MathML do not have a direct analog in Mathematica. Therefore, a few symbols are specially defined in the XML`MathML` context to represent these elements. These symbols are listed below.

<table>
<thead>
<tr>
<th>XML<code>MathML</code>Symbols`*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality</td>
</tr>
<tr>
<td>CartesianProduct</td>
</tr>
<tr>
<td>Codomain</td>
</tr>
<tr>
<td>ConditionedIntegrate</td>
</tr>
<tr>
<td>ConditionedLimit</td>
</tr>
<tr>
<td>ConditionedList</td>
</tr>
<tr>
<td>ConditionedMax</td>
</tr>
<tr>
<td>ConditionedMin</td>
</tr>
<tr>
<td>ConditionedProduct</td>
</tr>
<tr>
<td>ConditionedSet</td>
</tr>
<tr>
<td>ConditionedSum</td>
</tr>
<tr>
<td>CovariantDerivativeIndex</td>
</tr>
</tbody>
</table>

MathML Conversion Options

The standard ConversionOptions feature of the Export function can be used for greater control over the export process. The syntax for specifying a conversion option is as follows.

Export[file, expr, "MathML", ConversionOptions -> {option1 -> value1, option2 -> value2, ...}]

The following conversion options are specially useful when exporting MathML.

- "Annotations" controls whether to include an XML declaration or DTD declaration in the exported file.
- "Formats" controls whether to export presentation MathML, content MathML, or both.
- "NamespacePrefixes" controls whether to include a namespace prefix for each MathML element.
- "UseUnicodePlane1Characters" controls whether to include characters belonging to plane 1 of Unicode or to replace them with corresponding plane 0 characters.

Annotations

This option controls which annotations are added to the output MathML. The value of this option is a list whose elements can be "DocumentHeader", "XMLDeclaration", or "DOCTYPEDeclaration". The order of the elements in the list is not relevant.
**XMLDeclaration**

When "XMLDeclaration" is one of the annotations, then an XML declaration is included in the header. This means that the statement `<?xml version="1.0"?>` appears in the header.

```
Export["test.mml", x^2, "MathML",
    ConversionOptions -> {"Formats" -> {"PresentationMathML"},
    "Annotations" -> {"DocumentHeader", "XMLDeclaration"}}]
```

```
!! test.mml

<?xml version='1.0'?>
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <msup>
    <mi>x</mi>
    <mn>2</mn>
  </msup>
</math>
```

**DOCTYPEDeclaration**

When "DOCTYPEDeclaration" is one of the annotations, then an XML document type declaration of the form `<!DOCTYPE ... >` appears in the header. This is a statement that specifies the DTD for the XML application in which the output is written.

```
Export["test.mml", x^2, "MathML",
    ConversionOptions -> {"Formats" -> {"PresentationMathML"},
    "Annotations" -> {"DocumentHeader", "DOCTYPEDeclaration"}}]
```

```
!! test.mml

<!DOCTYPE ... >
```
When "Annotations" does not contain "DocumentHeader", then the output MathML has no header. This is true even if the "Annotations" contains other elements, such as "XMLDeclaration" or "DOCTYPEDeclaration". Thus, "DocumentHeader" is an overall switch that controls whether the structure has a header or not.
Export["test.mml", x^2, "MathML",
ConversionOptions = {"Formats" -> {"PresentationMathML"},
 "Annotations" -> {"DOCTYPEDeclaration", "XMLDeclaration"}}]

test.mml

!! test.mml

< math xmlns = 'http://www.w3.org/1998/Math/MathML' >
 < msup >
  < mi > x < / mi >
  < mn > 2 < / mn >
 < / msup >
 < / math >

 Formats

This option controls which type of MathML markup is generated. The value of the option is a list that can have as its elements "PresentationMathML", or "ContentMathML", or both. The default setting is "Formats"->{"PresentationMathML", "ContentMathML"}, which generates both presentation and content MathML in parallel.

The following command exports presentation MathML only.

ExportString[x^2, "MathML",
ConversionOptions -> {"Formats" -> {"PresentationMathML"}}]

<math xmlns='http://www.w3.org/1998/Math/MathML'>
 <msup>
  <mi>x</mi>
  <mn>2</mn>
 </msup>
</math>

This command exports content MathML only.
NamespacePrefixes

This conversion option lets you generate XML markup with a specific namespace declaration and namespace prefixes. The option is specified in the form

"NamespacePrefixes" → {url → prefix}

where url and prefix are strings specifying the URL of the namespace and the namespace prefix. In the following example, the "NamespacePrefix" option is used to generate presentation markup with each MathML element having a namespace prefix "mml" associated with the MathML namespace.

UseUnicodePlane1Characters

This conversion option controls whether characters belonging to plane 1 of Unicode should be replaced with similar characters in plane 0. This option is useful because currently most browsers cannot properly display plane 1 characters.
The option "UseUnicodePlane1Characters" and its possible values.

With the default setting, "UseUnicodePlane1Characters" → True, special characters that belong to plane 1 of Unicode (such as Gothic, scripted, and double-struck characters), are written out with their plane 1 numeric character codes. Here is an example involving the Gothic “g” character.

```
ExportString[g, "MathML"]
```

With the setting, "UseUnicodePlane1Characters" → False, any special character that belongs to plane 1 of Unicode, is replaced by a corresponding plane 0 character with a suitable value of the `mathvariant` attribute specified.

```
ExportString[g, "MathML", ConversionOptions → {"UseUnicodePlane1Characters" → False}]
```
Acknowledgments

Jason Harris and Thomasanna Hail were helpful in reviewing and editing the content of this article. The examples in the section on transforming XML were contributed by Buddy Ritchie, Chris Hill, and Shaun McCance.

Additional Material

The following data files, used in some of the examples, are available at www.mathematica-journal.com.

ALHitters.xml. An XML file containing data on baseball hitters.

NLPitchers.xml. Another XML file containing data on baseball pitchers.

methane.xml. A file specifying the structure of the methane molecule in MoDL syntax.

Shakespeare.nb. A notebook containing a quotation from Shakespeare.

About the Author


Pavi Sandhu
Technical Writing Manager
Wolfram Research, Inc.
pavi@wolfram.com