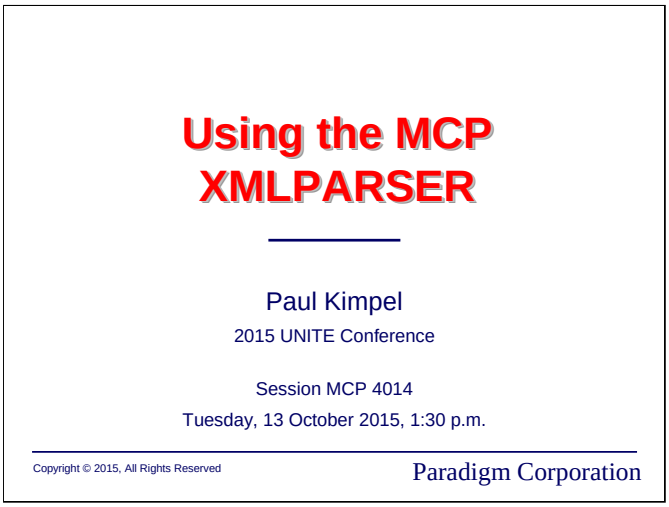



# Using the MCP

# XMLPARSER

2015 UNITE Conference
St. Louis, Missouri

Session MCP 4014

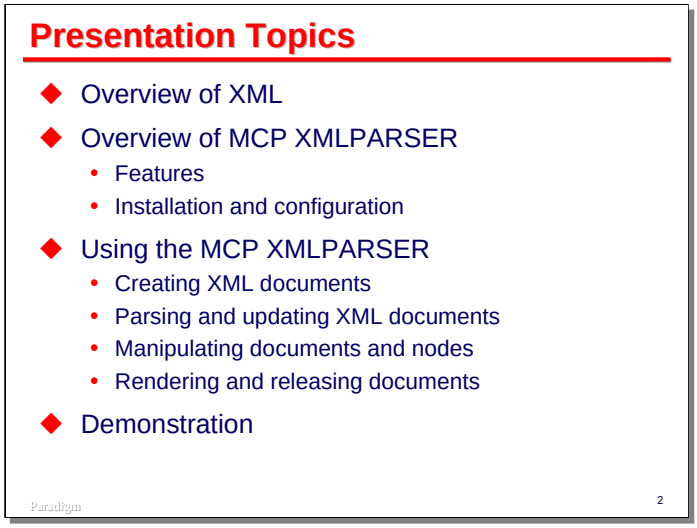
Tuesday, 13 October 2015, 1:30 p.m.

Paul Kimpel

Paradigm Corporation
San Diego, California

http://www.digm.com

e-mail: paul.kimpel@digm.com




I have a customer that does a lot of mailing. They were looking for a way to standardize and validate their mailing addresses as those addresses are entered and modified in their system. The U.S. Postal Service offers a free web service (part of its WebTools offering) that will do that on an interactive basis.

The customer needed a solution that would work for COBOL applications. Communicating with the USPS service was easily handled by the MCP HTTPCLIENT API, but the service required that both the input data and the response be coded in XML. Constructing an XML request in COBOL is not all that difficult, but parsing an XML response, especially in COBOL, can be quite a challenge. We started looking at the MCP XMLPARSER API, and found that it could do the job. Hence this presentation.

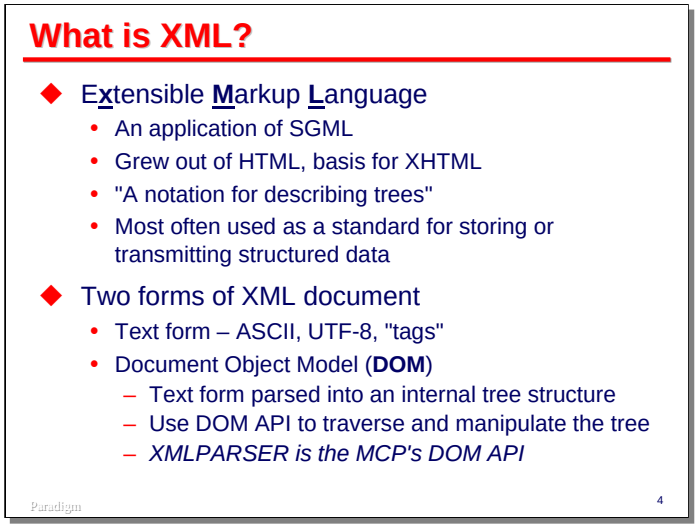
I will start with a brief overview of XML, focusing on a few key terms and concepts.

Next, I will give an overview of the XMLPARSER, its features, and how you install and configure it in the MCP environment.

The main part of the presentation will be devoted to discussing the XMLPARSER API and how you use it to process XML documents in a COBOL program. I will discuss the creation of new XML documents from scratch, parsing and updating existing XML documents, manipulating documents and the nodes of their DOM tree, rendering XML text out of the XMLPARSER, and releasing documents when you are finished with them.

I will conclude the presentation with demonstrations of a couple of XML-enabled COBOL programs.


To begin, let us discuss just what XML is, along with some terms and concepts we will need to understand the XMLPARSER and how to use it.




XML is an acronym for the E**x**tensible **M**arkup **L**anguage. This language is an application of SGML, the Standard Generalized Markup Language, which IBM began developing in the 1960s.

XML grew out of HTML (originally another application of SGML, but divorced from it with the advent of HTML 5.0). As HTML started to become popular in the late 1990s, people started to mine HTML documents for the data they contained. They found that what works well for producing a document in the visual realm did not work so well in the data parsing and extraction realm. Thus XML began as an attempt to define a notation for representing structured data rather than the prose text and images to which HTML was more suited.

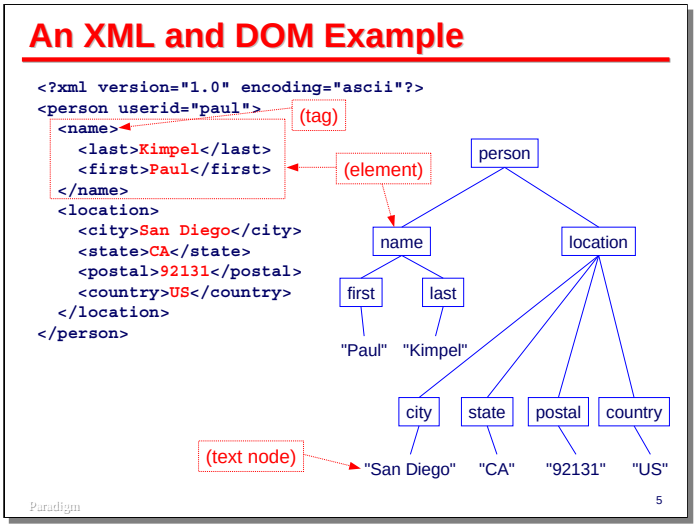
The most succinct description of XML that I have heard is that it is a "notation for describing trees." As we will see, XML documents are tree-structured, and carry their data payload largely in the leaves of the trees.

Today, XML is the predominant way to represent, store, and transmit structured data.

When we talk about an "XML document," we can mean two related, but entirely separate things. The obvious one is the textual representation of the document, with its tags, greater-than and less-than brackets, attribute name/value pairs, etc. These documents are typically encoded in ASCII or one of the ASCII-derived character sets, such as UTF-8.

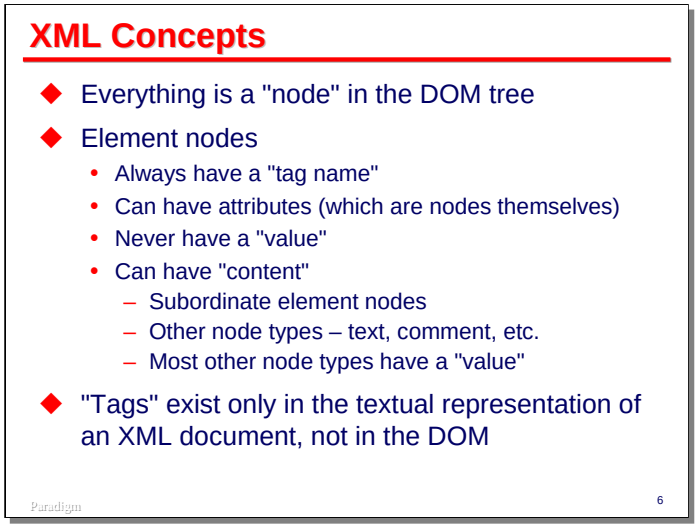
The second form of document can be thought of both as a conceptual view of the structure of the data and as the form the document takes inside software designed to manipulate the document. This form is called the Document Object Model, or DOM. It represents the document internally as a tree, and provides an API to navigate and manipulate the nodes of the tree.

The XMLPARSER is simply the MCP's DOM API for XML documents.




This slide illustrates the relationship between the textual and DOM forms of an XML document. On the left we see the textual representation of an XML data structure. On the right is the equivalent DOM tree.

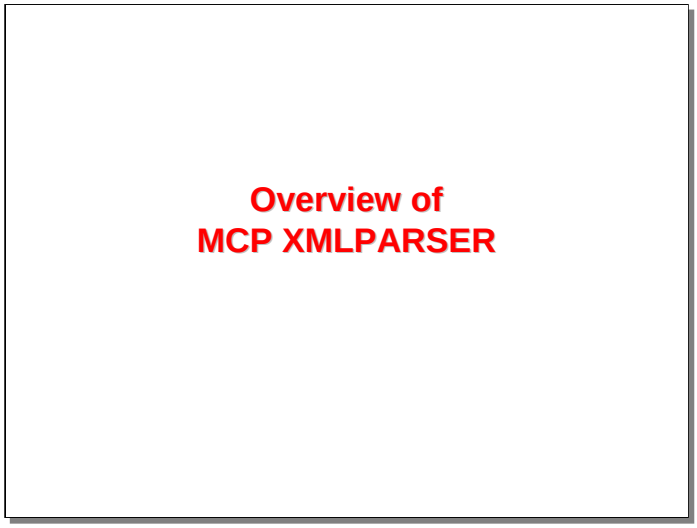
Note that tags exist only in the textual representation. The combination of an opening and closing tag defines an "element," which is represented as a node in the tree. Elements can contain other nodes, which may be subordinate elements or nodes of various other types. One of the most common types of nodes is a text node. Text nodes carry the literal data embedded in the XML document.



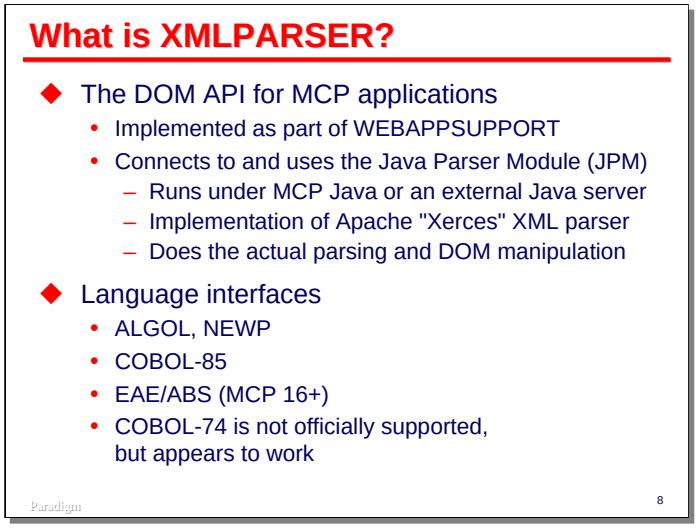

Everything in an XML document is represented as a node. Elements are one type of node, and have some special properties:

- Elements always have a "tag name." This is the name of the corresponding tag in the textual representation.
- Elements optionally can have "attributes," which are name=value pairs that provide metadata about the element and its contents. The <person> element in the previous example has a "**`userid`**" attribute, the value of which is "**`paul`**".
- Elements never have a "value." Text and most other types of nodes do have values.
- Instead of a value, elements may have "content," which are subordinate (child) nodes of the element node. Most commonly, elements contain text nodes and other elements.

Note that the only remnant of an XML tag that carries over to the DOM tree is the tag name, which becomes a property of the element node. The role of opening and closing tags in the textual document is replaced by parent-child relationships in the DOM tree.




With that background, let us now discuss the features, installation, and configuration of the MCP XMLPARSER API.

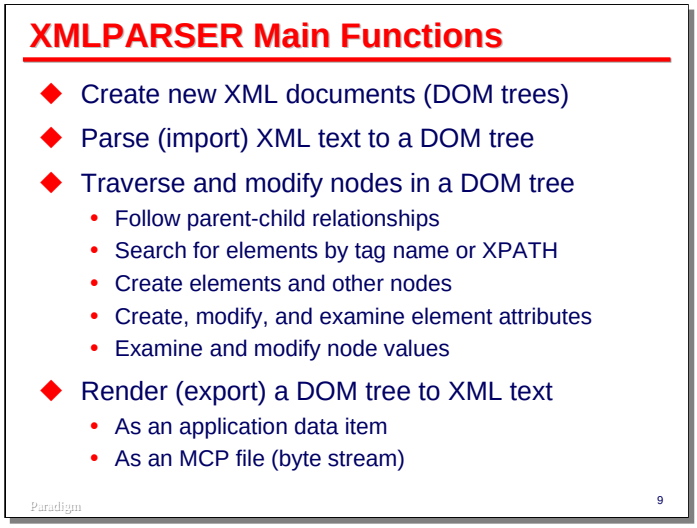



The XMLPARSER is nothing more than a DOM API adapted for use with MCP programming languages. It is implemented as part of the WEBAPPSUPPORT library, which in turn is a component of the Custom Connect Facility (CCF).

WEBAPPSUPPORT does not contain the code for parsing and manipulating XML documents, however. That function is performed by the Java Parser Module, or JPM. The JPM is an implementation of the Apache Foundation's Xerces XML parser. It is written in Java and runs either under MCP Java or on an external Windows server that hosts Java.

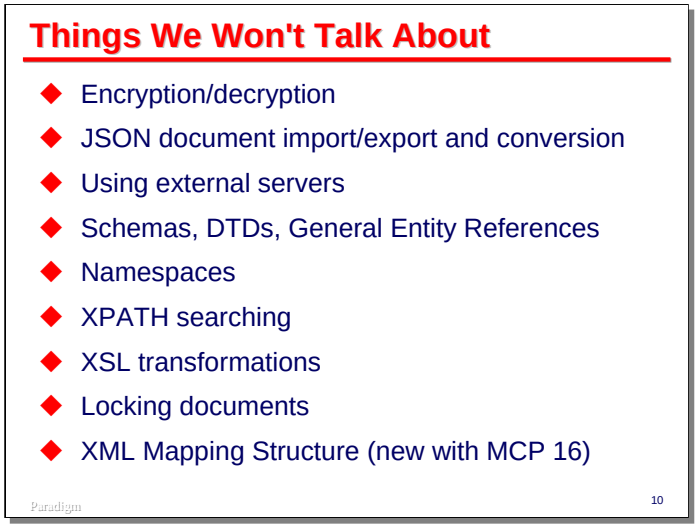
The DOM is stored and manipulated within the JPM, not the MCP environment. WEBAPPSUPPORT provides the mechanism to connect to the JPM and interface it to MCP applications.

The XMLPARSER API is supported in the Algol languages, NEWP, and COBOL-85. Starting with MCP 16, there is also an interface for EAE/ABS applications. COBOL-74 is not officially supported, but it appears to work.

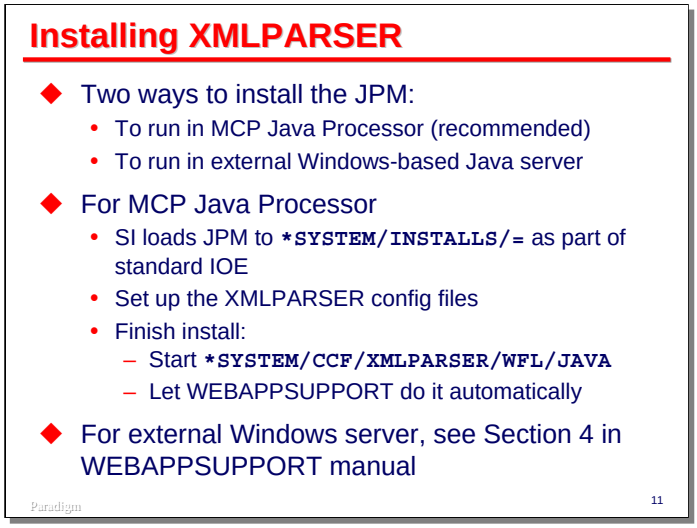



The XMLPARSER is a relatively large API, with over 50 methods accessible to MCP applications. At a high level, its abilities fall into a few general areas:

- You can create new XML documents (i.e., internal DOM trees) from scratch.
- You can parse (import) an XML text document into the API and create a DOM tree from it.
- Once you have a DOM tree, you can traverse and modify the nodes in it. You can follow parent-child-sibling relationships within the tree, search for elements by their tag name, search using a more general facility called XPATH, create new elements and other types of nodes, manipulate element attributes, and examine and modify node values.
- After you have created or modified a DOM tree, you can render (export) it to a textual XML document. The text can be retrieved either as a data item in your application, or in an MCP byte-stream file.




XML is a very rich facility, and has many capabilities that we simply do not have time to talk about in this presentation. The MCP XMLPARSER is a fairly complete API and supports all of the items listed on this slide. If you are interested in them, they are all documented in the WEBAPPSUPPORT reference manual.




There are two ways to install and run the Java Parser Module:

- Install it within the MCP environment and run it in the MCP Java Processor. This is the recommended approach.
- Install it on an external Windows server that has a recent release of the Java Development Kit (JDK).

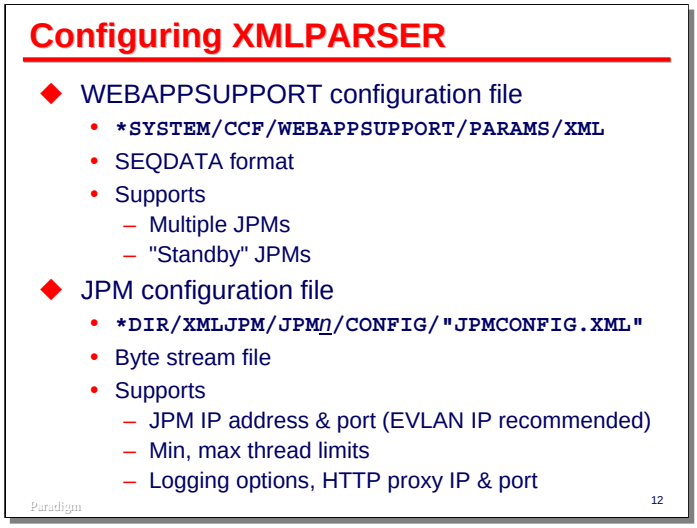
Installation for use by MCP Java is relatively easy. The XMLPARSER is bundled in the standard ClearPath IOE, and its files should be loaded to the **`*SYSTEM/INSTALLS`** directory by Simple Install.

Before using the XMLPARSER, you need to establish its configuration files, which are discussed over the next couple of slides.

You finish the installation in one of two ways:

- Manually start the **`*SYSTEM/CCF/XMLPARSER/WFL/JAVA`** job. This will copy the necessary files from the Installs directory to a permanent directory and initiate a Java task to run the JPM.
- Starting with MCP 14, WEBAPPSUPPORT can now automatically install the JPM the first time the XMLPARSER API is activated.

Installation and configuration of XMLPARSER in an external Windows server is beyond the scope of this presentation, but is described in Section 4 of the WEBAPPSUPPORT reference manual.




There are two configuration files that must be set up prior to running the XMPARSER API.

The first is the WEBAPPSUPPORT configuration file. This is a single file with global options for the API within the MCP environment. It is named

**`*SYSTEM/CCF/WEBAPPSUPPORT/PARAMS/XML`**

and is in SEQDATA format (text in columns 1-72, sequence numbers in 73-80).

The primary purpose of this file is to define the JPMs to which WEBAPPSUPPORT will connect and to provide attributes for each JPM connection. You can define multiple JPMs, and must define at least one. When there are multiple JPMs, WEBAPPSUPPORT will distribute the workload across them. When there are multiple JPMs, some of them can be identified as "standby" JPMs, which can be brought into service to replace an active JPM that has failed, or whose Java Processor has failed.

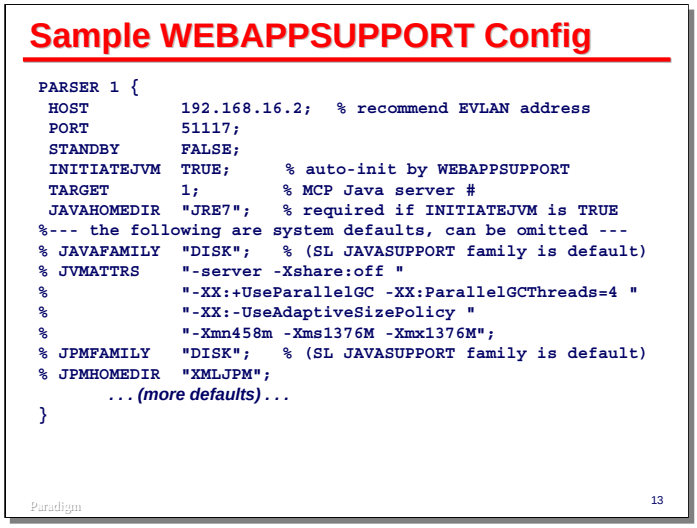
A sample WEBAPPSUPPORT configuration file is shown on the next slide.

The second is the JPM configuration file. There is a separate copy of this file for each JPM defined in the WEBAPPSUPPORT configuration file. These files are named:

**`*DIR/XMLJPM/JPM`*n*`/CONFIG/"JPMCONFIG.XML"`**

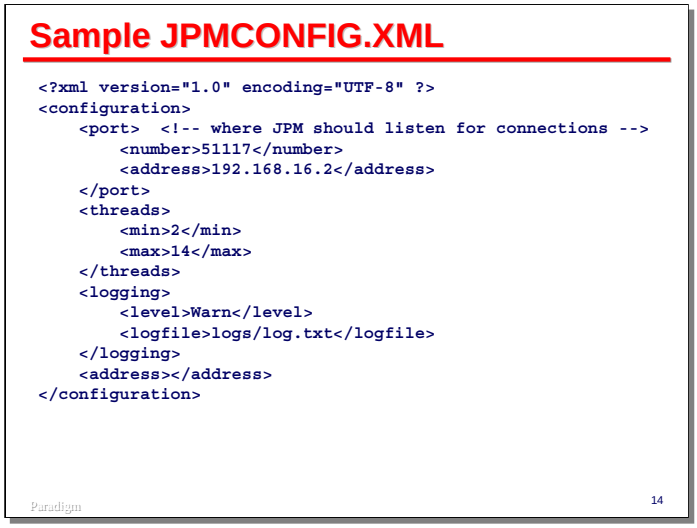
Where *n* corresponds to the parser number defined in the WEBAPPSUPPORT configuration file. This is a byte-stream file, so you will not be able to edit it with CANDE or PWB.

This file defines the IP address and port number on which the JPM will listen for connections from WEBAPPSUPPORT. When running under MCP Java, use of one of the EVLAN addresses is recommended. The file also specifies minimum and maximum numbers of threads the JPM can use, as well as logging and network proxy configuration.



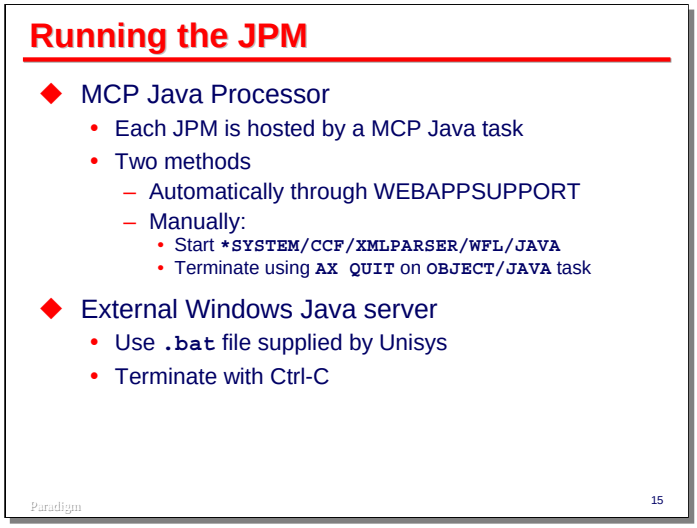

This slide shows a sample WEBAPPSUPPORT configuration file containing attributes for a single JPM.

- Each JPM is assigned a number using the **PARSER** keyword.
- The **HOST** and **PORT** attributes define the network address of the JPM. WEBAPPSUPPORT will attempt to connect over this address and port.
- **STANDBY** indicates whether this is an active or standby JPM.
- **INITIATEJVM** indicates whether WEBAPPSUPPORT is to initiate the JPM automatically as needed (**TRUE**) or the JPM will be initiated manually using the Unisys-supplied WFL job (**FALSE**).
- **TARGET** identifies the number of the Java Processor on which this JPM will run.
- **JAVAHOMEDIR** identifies the version of MCP Java that will be used. It becomes a node in the name of the Java VM task (e.g., ***DIR/JRE7/BIN/JAVA**).
- The remaining attributes for a parser definition have system defaults that work well for most environments. You can usually delete these or comment them out in the configuration file. See the WEBAPPSUPPORT reference manual for details on these.




This slide shows a sample **`JPMCONFIG.XML`** file for one of the JPMs. The primary purpose of this file is to specify the IP address and port on which the JPM will listen for connections from WEBAPPSUPPORT. The address and port should match those specified for the JPM in the WEBAPPSUPPORT configuration file. Use of the EVLAN is recommended wherever possible, as it is more secure than external networks.

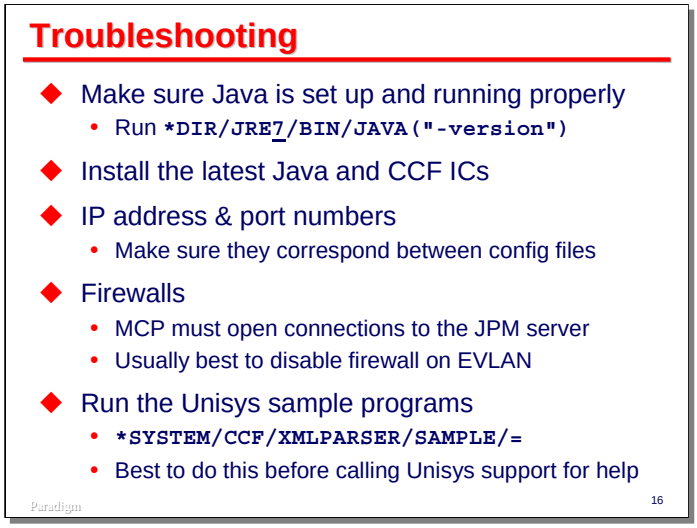
You can also specify limits for the minimum and maximum number of threads the JPM will execute at one time, logging options, and if necessary, the network proxy configuration.




When running the JPM under MCP Java, there are two ways to initiate a JPM:

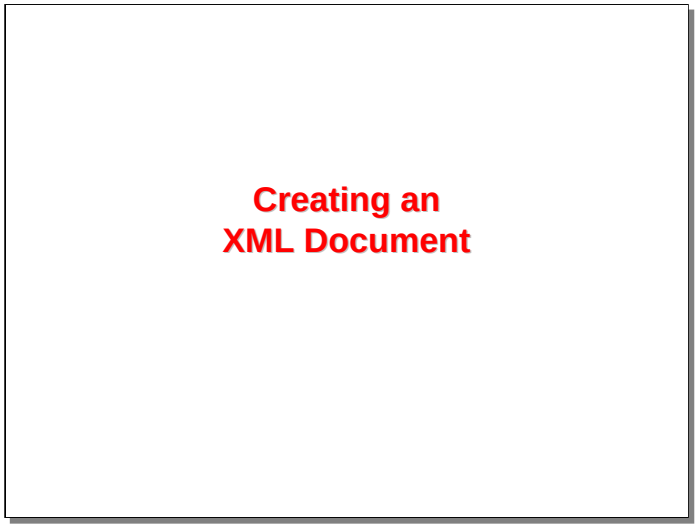
- If the **INITIATEJVM** attribute in the WEBAPPSUPPORT configuration file is **TRUE**, WEBAPPSUPPORT will automatically initiate the JPM when the library initiates. Once all applications disconnect from the library, the library will terminate all JPMs gracefully and then terminate itself.
- If you choose not to have WEBAPPSUPPORT initiate the JPM automatically, you can manually start the Unisys-supplied WFL job. This is the same WFL used to install the JPM. The job will run an MCP Java task to host the JPM. You can terminate this task (and the WFL job) gracefully by sending the task an "**AX QUIT**".

Running the JPM in an external Windows server is beyond the scope of this presentation. Unisys supplies a **.bat** file that can be used to initiate the JPM, which can be terminated using Ctrl-C from the command line.

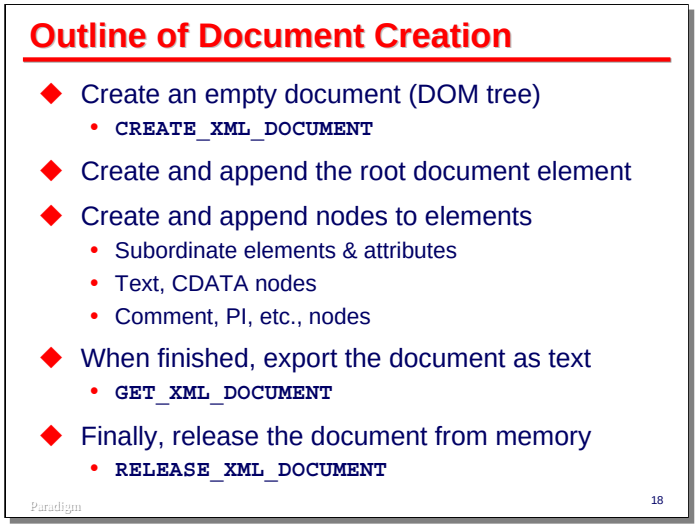



Getting the XMLPARSER to work initially can sometimes be problematic. Here are a few things to watch for if you are having trouble:

- Make sure that MCP Java is set up and running properly. You can check this by running the Java task shown on the slide.
- It may help to install the latest Java and CCF Interim Corrections from the Unisys support site. The base release for MCP 15, in particular, had several issues.
- Make sure the IP addresses and port numbers match between the WEBAPPSUPPORT configuration file and the JPM configuration files.
- Check that firewalls are not inhibiting connections on the addresses and port numbers used by the JPMs. When running on an entry-level Libra server (where the MCP environment runs within the Windows environment) and the Java Processor is running in the local Windows environment, make sure that Windows Firewall is not blocking the IP addresses or port numbers.
- Try running the Unisys sample programs, which are located in the directory shown on the slide. You should probably do this before calling Unisys support for help, since that is probably the first thing they will ask you to do, anyway.

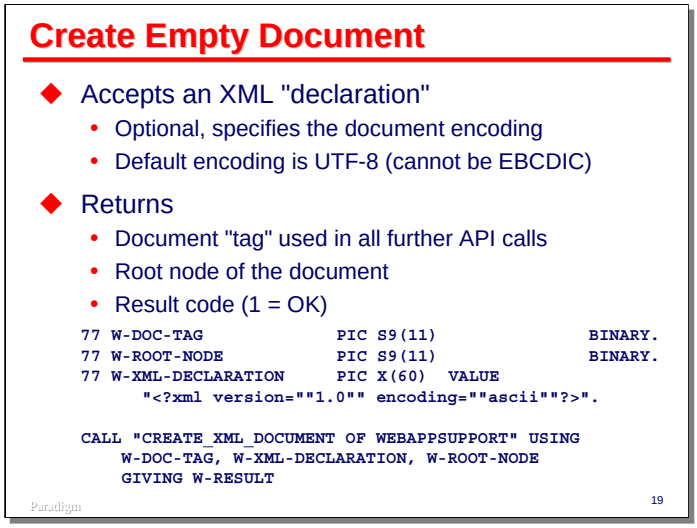



Once the XMLPARSER is installed and configured properly, you can start to use it to manipulate XML documents. The first activity we will explore is creation of an XML document from scratch.




Creation of an XML document generally requires the following steps:

- First, create a new, empty DOM tree using **`CREATE_XML_DOCUMENT`**.
- Second, create an append the root document element. XML documents must consist of a single element at the root of the DOM tree. This element is termed the document root.
- Next, build the tree by appending subordinate elements and nodes to the tree. Elements may need to have attributes appended to them. Text is inserted in the document by creating text nodes and appending them to element nodes. Elements may contain other types of nodes, including other elements.
- Once the DOM tree is built, export the document to text using **`GET_XML_DOCUMENT`**.
- Finally, release the document using **`RELEASE_XML_DOCUMENT`**. This will return the memory used by the document in both WEBAPPSUPPORT and the JPM, and will also free up the table slots used by the document in WEBAPPSUPPORT.




As with most WEBAPPSUPPORT APIs, the XMLPARSER creates objects that you manipulate. These objects are not allocated within your application, however, but within WEBAPPSUPPORT. The API returns to you a "tag" value (not to be confused with a textual XML tag) that identifies the object to WEBAPPSUPPORT. You must save these object tag values and pass them as required to API methods.

When creating an XML document, you may optionally supply a string known as an XML "declaration." This has the form "**`<?xml`** ", optionally followed by attributes of the document as name=value pairs, and terminated by "**`?>`**". Two common attributes are:

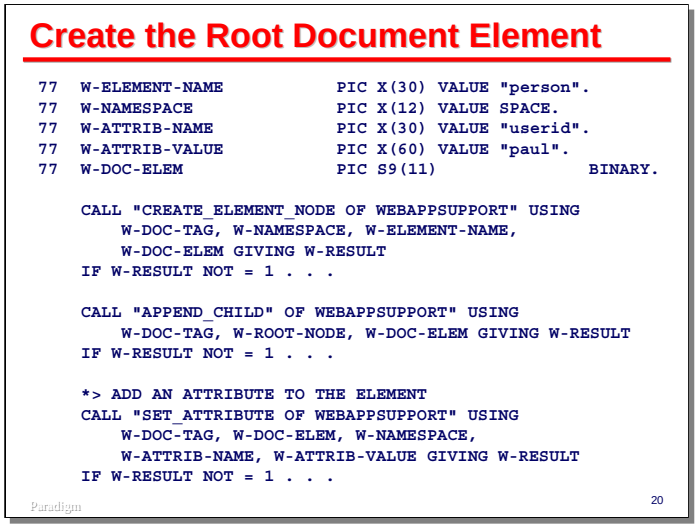
- **`version`** – the choices are "1.0" and "1.1".
- **`encoding`** – this specifies the *document* character set, i.e., the character set used within the JPM. It must be an ASCII-based character set. If not specified, it will be UTF-8. It may not be EBCDIC.

The slide shows a typical example of an XML declaration string. Note that XML is case sensitive and that attribute values must be enclosed in either double or single quotes.

To create a new XML DOM tree, you call the **`CREATE_XML_DOCUMENT`** method, passing the following parameters:

- The first parameter is a data item in which the document object tag will be returned. All tag values are binary integers. If the data item contains a tag value for a previously-allocated document, that document will be released before the new document is created. The document tag must be supplied on most XMLPARSER method calls.
- The second parameter is the data item containing the XML declaration string. The declaration is optional. If it is not to be supplied, the data item should contain spaces; otherwise the declaration text should be left-justified over spaces.
- The third parameter is a data item in which the object tag for the "root node" of the document will be returned. Note that this is not the "root document element" node. It is the node to which the root element must be attached.

Most WEBAPPSUPPORT methods return a binary integer as the procedure result. A value of 1 indicates the call was successful. Negative numbers indicate errors. In some cases a zero is returned to indicate the method took no action or returned no data.




The next step is to create the "root document element" and attach it to the "root node." This requires at least two method calls.

**`CREATE_ELEMENT_NODE`** creates a new, empty element that is not yet attached to the document.

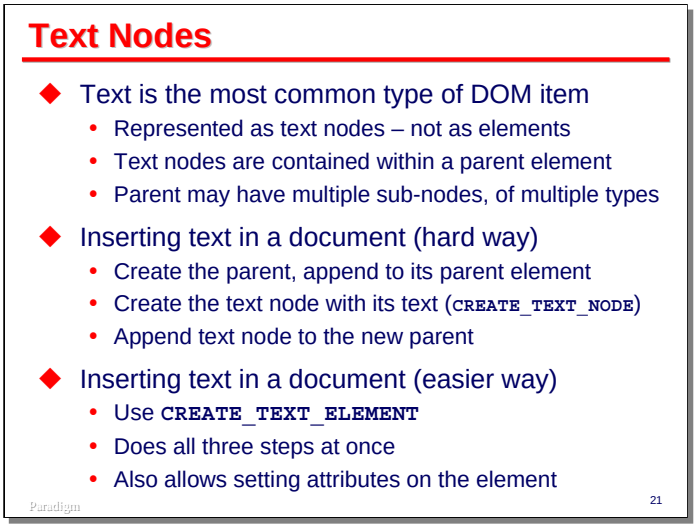
- The first parameter is the document object tag.
- The second parameter is the optional namespace for the tag name. Namespaces are beyond the scope of this presentation. If your document does not use namespaces, this data item should contain spaces.
- The third parameter is the element's tag name.
- The fourth parameter is a data item that will return the new element's object tag value.

**`APPEND_CHILD`** will attach a node as the last child of a specified parent node. Since this is a new document, the only node that exists is the root node, so that must be used as the parent.

- The first parameter is the document tag.
- The second parameter is the tag for the parent node. In this case it is the value for the root node returned from **`CREATE_XML_DOCUMENT`**.
- The third parameter is the object tag for the child node being appended to the parent. In this case it is the object tag for the new element just created. If the node being appended is already the child of some other node, it is removed from that node before being appended to the new parent.

If the new element is to have attributes, they may be attached to the element either before or after it is appended to its parent. Attributes can be created and modified using the **`SET_ATTRIBUTE`** method:

- The first parameter is the document object tag.
- The second parameter is the object tag for the attribute's element.
- The third parameter is a data item containing the namespace for the attribute, or spaces if there is no namespace.
- The fourth parameter is a data item containing the attribute name, left-justified over spaces.
- The fifth parameter is a data item containing the attribute value, also left-justified over spaces.

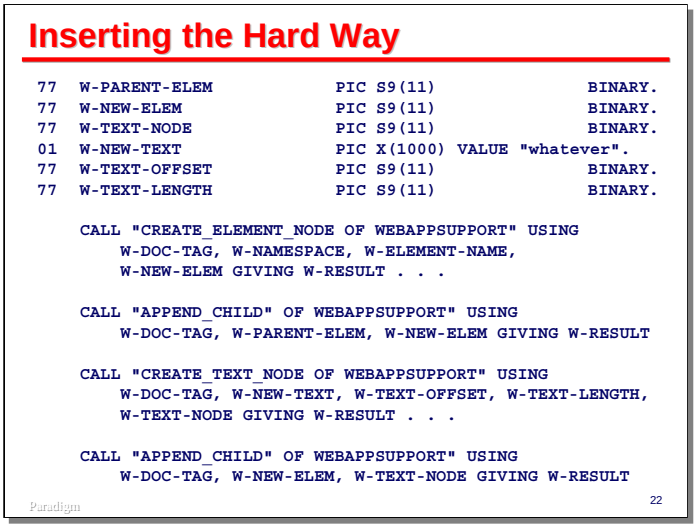



Perhaps the most common type of entity in an XML document is plain text. Text is represented in the DOM tree as text nodes, not as elements. Text nodes are always contained within (i.e., are children of) element nodes. A parent element may have multiple children, which may be any combination of elements, text nodes, and other types of nodes.

There are two ways using XMLPARSER to insert text nodes into a document. The "hard way" is to create an element (or use an existing one), and if necessary, append that element to a parent. Then you create a text node and append that to the element.

Since creating an element with a single subordinate text node is a very common operation, XMPARSER provides a convenience method, **`CREATE_TEXT_ELEMENT`**, that will create the element, append it to a parent element, create the text node, and append that to the new element, all in one call. It also allows you to set multiple attributes for the newly-created element.

Examples of both approaches are shown on the following slides.



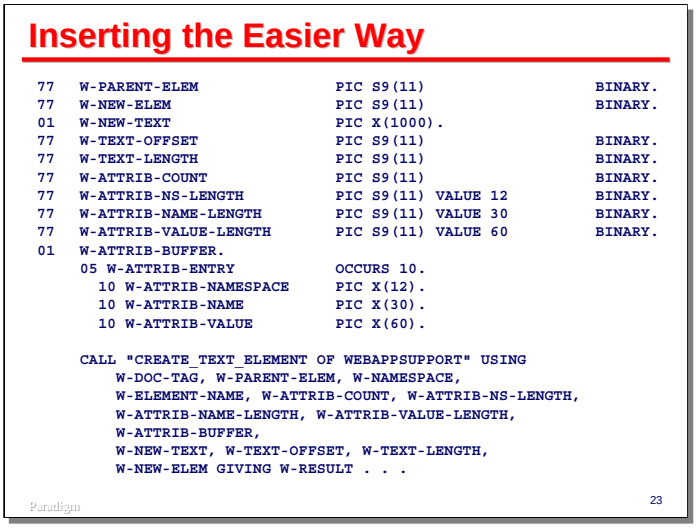

The "hard way" requires at least four method calls, assuming the element containing the text node does not already exist.

The first two steps, using **`CREATE_ELEMENT_NODE`** and **`APPEND_CHILD`**, work the same as for creating the root document element discussed a few slides ago.

**`CREATE_TEXT_NODE`** creates a new text node:

- The first parameter is the document object tag.
- The second parameter is a data item containing the text for the node.
- The third parameter is a zero-relative offset within the string of the second parameter where the text starts.
- The fourth parameter is the length of the text in bytes.

**`APPEND_CHILD`** is then used again to append the new text node to the new element node.

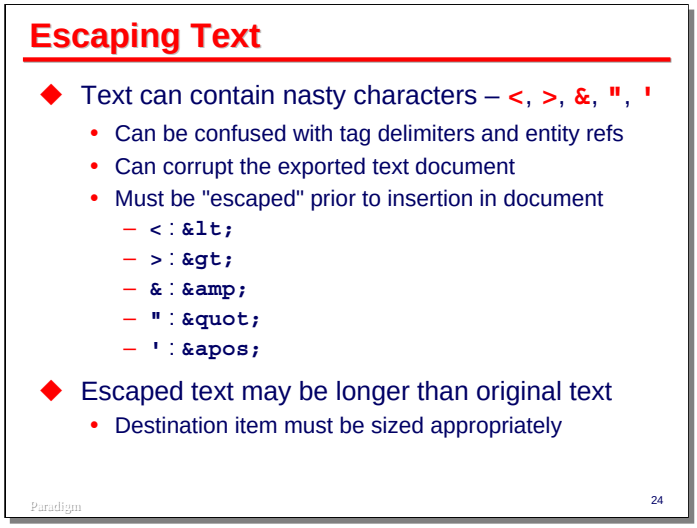



This slide shows an example of creating a text node the "easier" way. This looks like a lot of code, but most of it consists of data declarations that are written only once and can be reused.

In addition to creating the element and text nodes, **CREATE_TEXT_ELEMENT** allows you to specify names and values of multiple attributes for the new element. It does that using a COBOL-like table containing three columns: namespace (which should be blank if namespaces are not being used), name, and value. You need to specify to the method the number of attributes you want to create, and the widths of the namespace, name, and value columns in the table, as described below.

The method takes the following parameters:

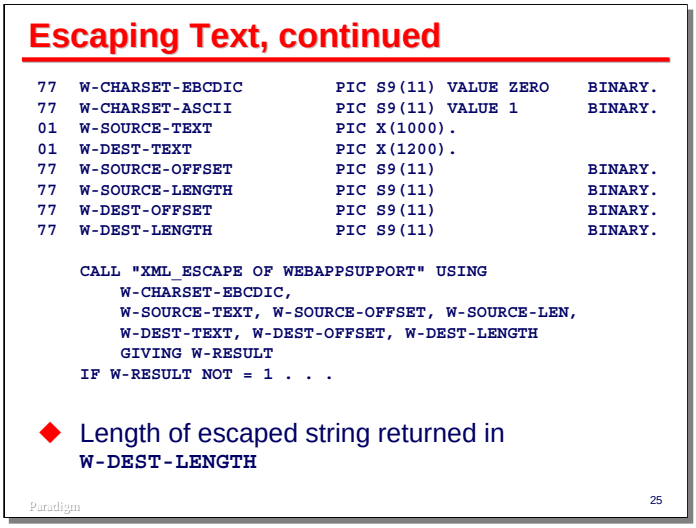
- The first parameter is the document object tag.
- The second parameter is the object tag for the parent element to which the new element should be attached.
- The third parameter is the namespace for the new element's tag name, and should be spaces if namespaces are not being used.
- The fourth parameter is the new element's tag name, left-justified over spaces.
- The fifth parameter is a binary data item indicating the number of attributes to be created. If this value is zero, then the sixth, seventh, eighth, and ninth parameters are not relevant and their contents are ignored by the method.
- The sixth parameter is the width of the attribute table's namespace column.
- The seventh parameter is the width of the attribute table's name column.
- The eighth parameter is the width of the attribute table's value column.
- The ninth parameter is the data item holding the table of attribute data.
- The tenth parameter is the text of the new text node.
- The eleventh parameter is the zero-relative offset into the tenth parameter to the start of the text, as for **CREATE_TEXT_NODE**.
- The twelfth parameter is the length of the node text, as for **CREATE_TEXT_NODE**.
- The thirteenth parameter is a binary integer that will return the object tag for the new *element* node. An object tag for the text node is not returned, but can be obtained as the first child of the new element node.




When creating text nodes, you must be sensitive to the fact that the DOM tree will probably be exported as text at some point. Certain characters in text nodes can be confused with delimiters in the textual representation of an XML document, and if present, may result the textual document being corrupt and unable to be parsed later.

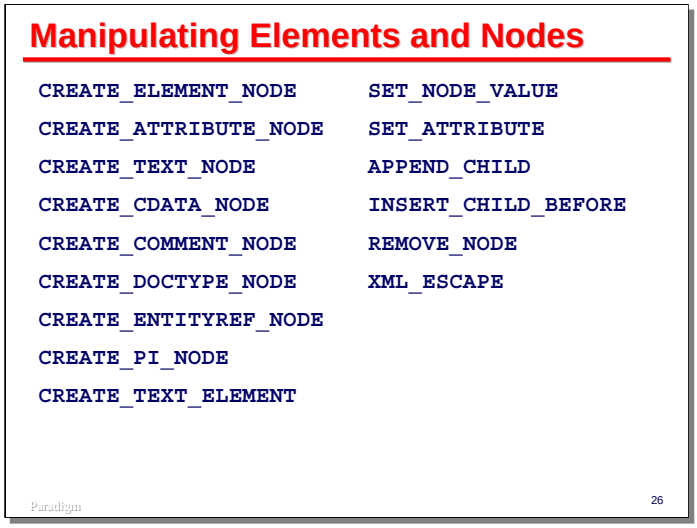
The problematic characters are shown on the slide. If it is possible these may be present in the text, then you must "escape" the text before using it to create a text node. The slide shows the "character entities" used in XML text to represent these characters. These are the same character entities used to escape HTML documents.

Note that if the original text contains any of the problematic characters, its escaped form will be longer than the original. You must take this into account when sizing the destination data item.



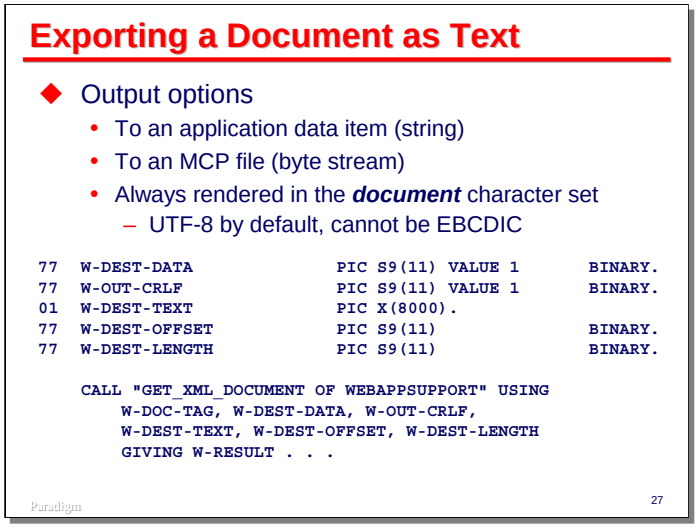

The WEBAPPSUPPORT library provides a utility method that will escape XML text, **`XML_ESCAPE`**. It has the following parameters:

- The first parameter is a binary value indicating the *application* character set. This is the character set in which your application represents text. Value 0 is EBCDIC; value 1 is ASCII.
- The second parameter is a data item containing the source (unescaped) text.
- The third parameter is the zero-relative offset to the start of the unescaped text in the second parameter.
- The fourth parameter is the length of the unescaped text in bytes.
- The fifth parameter is the data item to receive the escaped text. This data item should not overlap the source data item.
- The sixth parameter is the zero-relative offset indicating where in the fifth parameter the escaped text should begin.
- The value of the seventh parameter is ignored on input. On output it returns the length of the escaped text in characters.




This slide shows all of the WEBAPPSUPPORT methods for manipulating XML document elements and nodes. We have already discussed several of these, but a few more bear mention:

- We have already discussed **`CREATE_ELEMENT_NODE`** and **`CREATE_TEXT_NODE`**. The remaining node creation methods are for nodes of other types. These are not as common, and are seldom used in simple XML documents.
- **`SET_NODE_VALUE`** can be use to change the value associated with many node types. It is particularly useful to change the text of a text node.
- **`INSERT_CHILD_BEFORE`** is similar to **`APPEND_CHILD`**, but instead of appending a child node at the end of the list of child nodes for a parent, it inserts the child before a specified, existing child of the parent node.
- **`REMOVE_NODE`** removes a node and all of its child nodes from the document. At the completion of this call, that node and its children no longer exist.

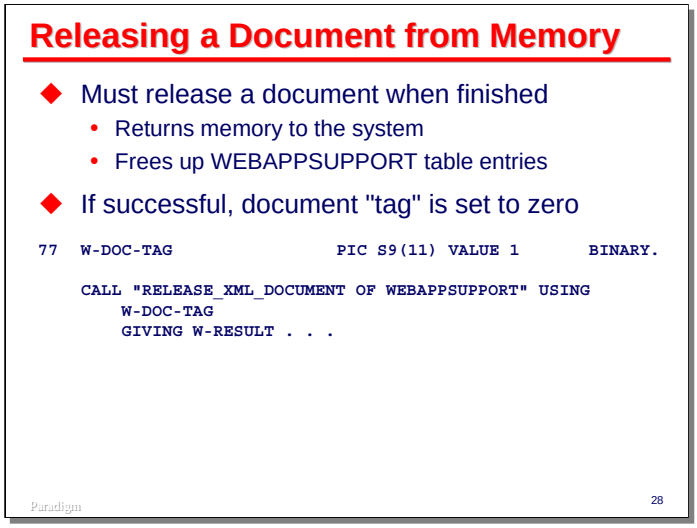



Once you have a document created and all of its nodes in place in the DOM tree, you can export the document in textual form. XMLPARSER allows you to return the text in a data item, or to an MCP byte stream file.

Note that the text is always rendered using the *document* character set (i.e., the one that is internal to the JPM), not the *application* character set used by your program. The document character set is determined from the XML declaration when the document is created or imported from a text file. It must be one of the ASCII-based character sets. If not specified, the character set defaults to UTF-8. The document character set may not be EBCDIC.

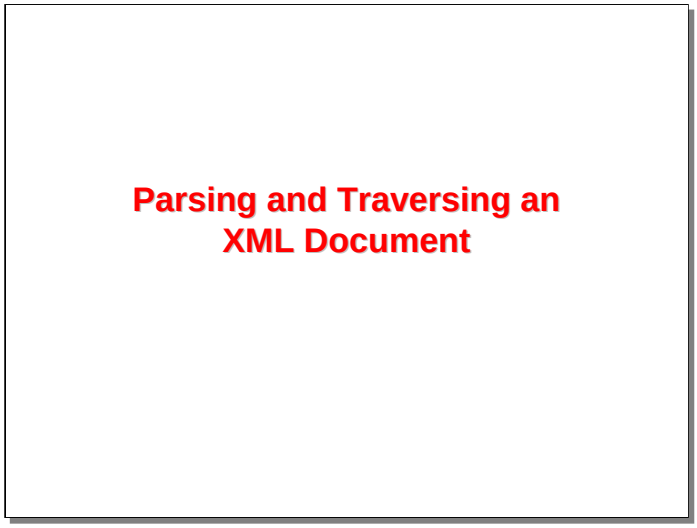
The slide shows an example of rendering a document to a data item using **`GET_XML_DOCUMENT`**:

- The first parameter is the document object tag.
- The second parameter is a binary integer that specifies the destination of the document. Value 1 indicates a data item (the fourth parameter); value 2 indicates an MCP byte-stream file.
- The third parameter is a binary integer that specifies whether the JPM is to format the document with new-lines and indenting. Value 1 indicates that new-lines and indenting are to be formatted; value 2 indicates no new-lines or white space are to be included in the document. A call to **`SET_XML_OPTION`** can specify the number of spaces to indent at each level of the document structure.
- The fourth parameter is a data item that will either receive the rendered text of the document or specify the title of the MCP file to which the document will be rendered.
- The fifth parameter is a zero-relative offset into the fourth parameter that indicates where the rendered text or MCP file title will start.
- The sixth parameter specifies the length of the data in the fourth parameter. If rendering XML text to a data item, on output this parameter will have the length of the rendered text. If rendering to an MCP file, on input this parameter will specify the length of the file title.

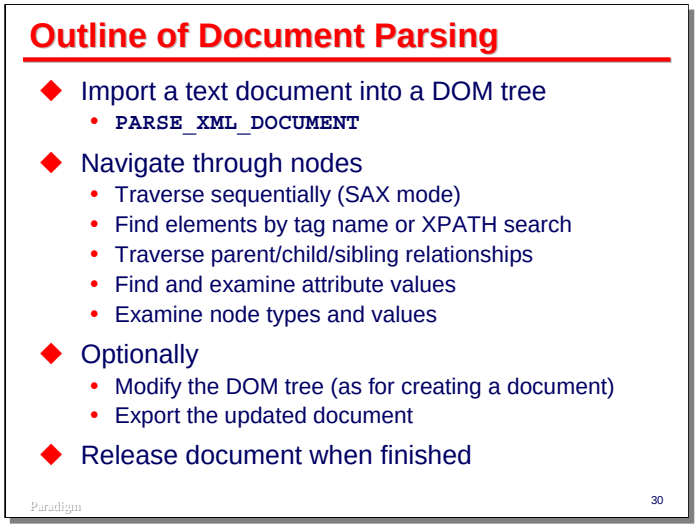



When you have finished working with a document in the XMLPARSER DOM, you must release it. This causes WEBAPPSUPPORT to return memory to the system and free up internal table slots.

You call **`RELEASE_XML_DOCUMENT`** passing the document object tag. If the release is successful, the method will set the tag's data item to zero.

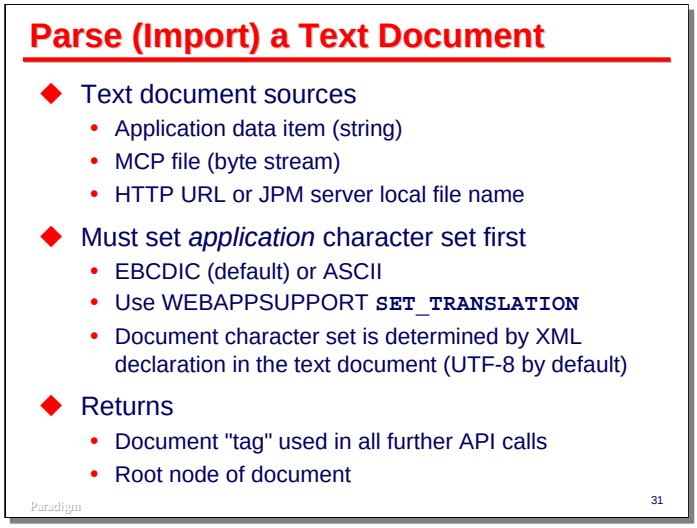



Thus far we have been discussing creation of an XML document and manipulating its elements and nodes. You can also start with an existing XML document in textual form, parse (import) it into a DOM tree, and traverse the nodes of the tree to examine its structure and data content.




Parsing and traversal of an existing XML document generally involves the following steps:

- Import the document into a DOM tree using **`PARSE_XML_DOCUMENT`**.
- Once the document is in the form of a DOM tree, you can navigate through its nodes in a number of ways:
  - The simplest form of traversal is to step through the document nodes sequentially. This is sometimes referred to as "SAX mode."
  - You can search for specific elements either by name or by using an XPATH query. XPATH is beyond the scope of this presentation.
  - You can traverse the parent/child/sibling relationships among the nodes of the tree.
  - Given a node, you can search for attributes by name and examine attribute values.
  - You can also examine nodes for their type and value.
- After importing a document, you can optionally modify the document in the DOM tree and then export it to text when finished.
- As always, when you are finished with the DOM tree, you must release the document to free up memory and WEBAPPSUPPORT table slots.



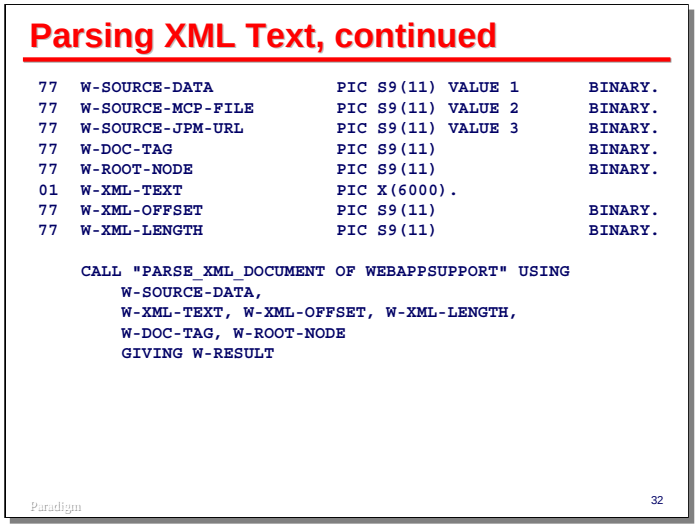

Parsing an XML document involves supplying the text of the document to the JPM so that it can convert it to a DOM tree. The XML text can come from a variety of sources:

- It can be passed to the API as a string in a data item.
- It can be read from an MCP byte-stream file. Record-oriented files are not supported.
- It can be read from a web server given the URL at which the document can be found.
- It can be read from a local file on the Windows server that is hosting the JPM.

Before parsing a document, you should set the *application* character set that will be used by WEBAPPSUPPORT. This is the character set your program uses to interface with the API. The default is EBCDIC, but you can set it to ASCII using the **`SET_TRANSLATION`** method of WEBAPPSUPPORT.

The *document* character set used internally by the JPM is determined from the XML declaration in the text document. If there is no declaration or the encoding is not specified, it defaults to UTF-8.

The result of parsing a document is similar to that of creating a new document: the **`PARSE_XML_DOCUMENT`** returns the document object tag and the root node of the document.

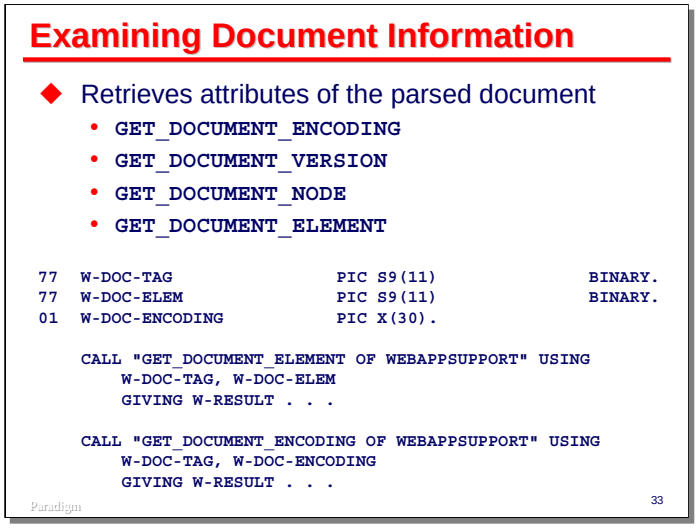



This slide shows an example of parsing a document from a data item.

The call to the **`PARSE_XML_DOCUMENT`** method has the following parameters:

- The first parameter indicates the type of source from which the textual document will be read. Value 1 indicates a data item (the second parameter); value 2 indicates an MCP byte-stream file, the title of which must be in the second parameter; value 3 indicates a URL or file local to the JPM server.
- The second parameter represents the source, either the XML text itself, the title of an MCP file, or a URL or JPM file name.
- The third parameter is the zero-relative offset into the second parameter where the text starts.
- The fourth parameter is the length of the text in bytes.
- The fifth parameter is a binary data item that will receive the parsed document's object tag.
- The sixth parameter is another binary data item that will receive the tag for the document's root node.

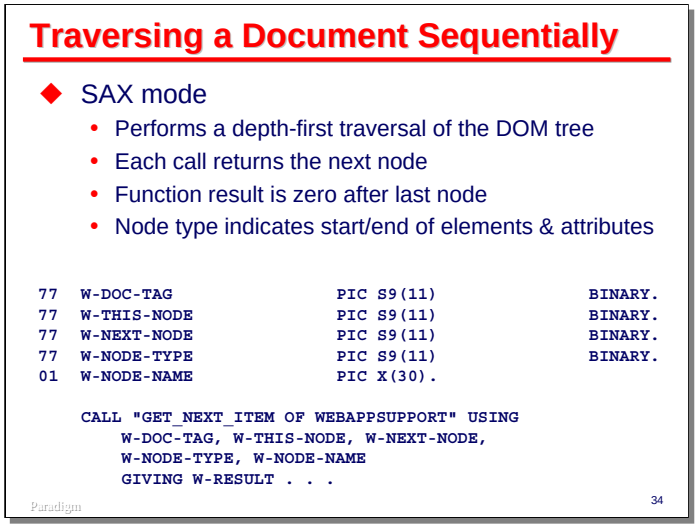
Any errors will be reported in the result value returned by the method.




Once you have the textual XML document parsed and imported as a DOM tree, you can begin to use the API to examine it. There are four methods that return information about the document itself:

- **`GET_DOCUMENT_ENCODING`** returns a string identifying the *document* character set used internally by the JPM. This is determined from the encoding attribute of the XML declaration for the document, and will be UTF-8 if not specified.
- **`GET_DOCUMENT_VERSION`** will return the version of the XML standard under which the document is to be interpreted. This is also an attribute of the XML declaration. If no version was specified, the method returns "1.0".
- **`GET_DOCUMENT_NODE`** returns the root node of the document. This is the same value as is returned by the sixth parameter of **`PARSE_XML_DOCUMENT`**.
- **`GET_DOCUMENT_ELEMENT`** returns the root document element. This is the outermost element of the document, and is a child of the root node.

The slides show a couple of examples of calls to these routines.

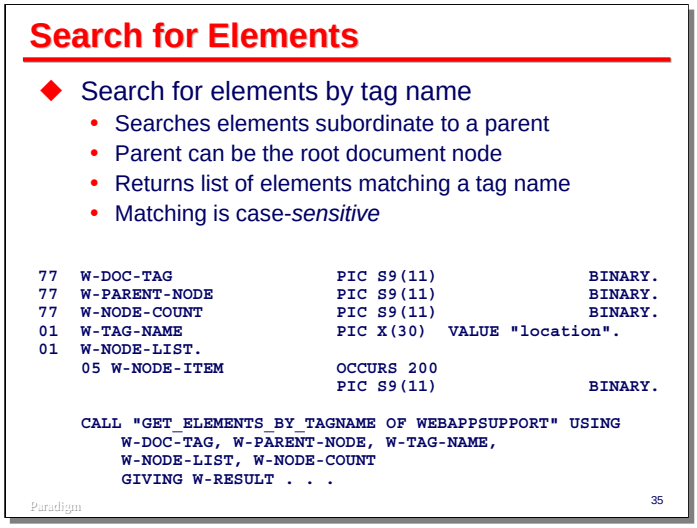



One option for examining an XML document is to traverse its nodes sequentially using **`GET_NEXT_ITEM`**. This performs a depth-first traversal of the DOM tree using so-called "SAX mode" access. You can start with any node in the document. If you want to traverse the entire document, start with the root node. The method returns a zero result when the end of the document is reached.

While the concept is simple, actually using sequential traversal can be complex. Each call returns a node-type value that describes the node just encountered. For element, attribute, and entity-reference nodes, which can have subordinate nodes, a type value is returned twice – once when the node is entered and once when it is exited after traversing all of its children. This allows the application to keep track of the nesting of the nodes within each other.

**`GET_NEXT_ITEM`** has the following parameters:

- The first parameter is the document object tag.
- The second parameter is the tag for the node where you want to start. This is usually the node returned by the prior call on **`GET_NEXT_ITEM`**.
- The third parameter returns the tag for the node immediately following the one specified by the second parameter.
- The fourth parameter returns the node-type value for the node represented by the third parameter. See the documentation in the WEBAPPSUPPORT reference manual for a list of these values.
- The fifth parameter returns a string with the tag name (if any) of the node represented by the third parameter.

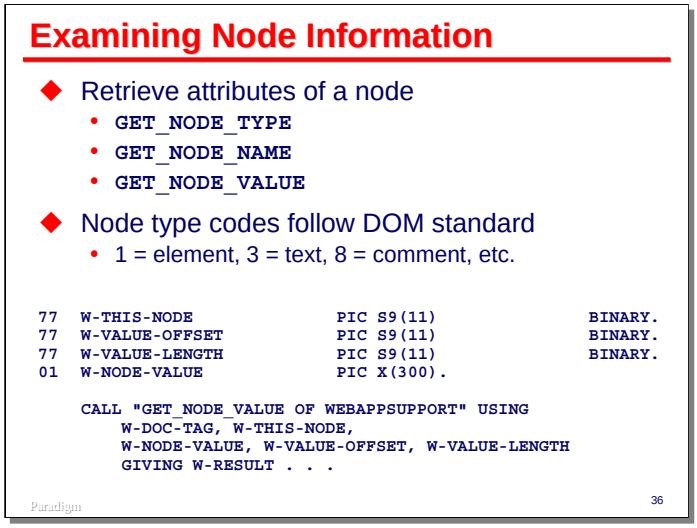



Another way to traverse the document is to search for elements having a specified tag name. Since there are frequently multiple elements in a document with the same tag name, this search returns a list of element nodes.

This type of search takes place recursively for all of the children (and their children, etc.) subordinate to a specified parent node. Thus, the resulting list will contain only nodes that are subordinate to that parent.

The slide shows an example of a call on **`GET_ELEMENTS_BY_TAGNAME`**:

- The first parameter is the document object tag.
- The second parameter is the object tag for the parent node under which you wish to search. If you need to search the whole document, use the document's root node as the parent.
- The third parameter is the tag name to search for. Since XML is a case-sensitive notation, matching of tag names is also case sensitive.
- The fourth parameter is a data area that will receive the list of nodes that were found. The list is an array of binary integers containing the object tag for each such node.
- The fifth parameter is a binary data item that will return the number of nodes in the list.

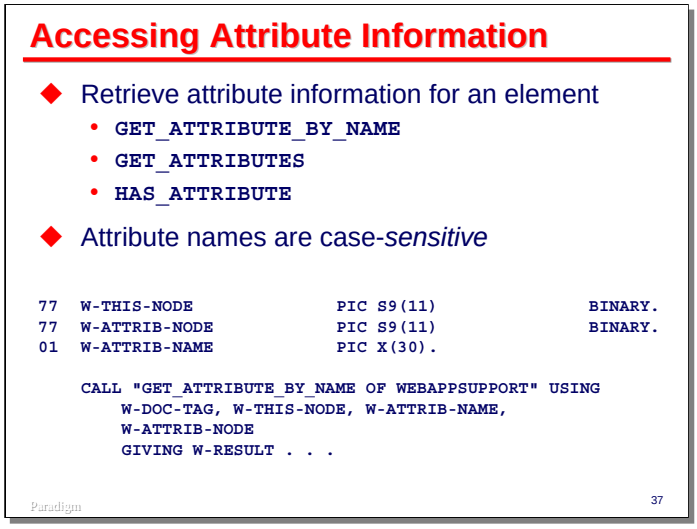



Once you have the object tag value for a node, you can examine information about that node:

- **GET_NODE_TYPE** returns the node-type value for the node. Every node has a type:
  - 1 = element node
  - 2 = attribute node
  - 3 = text node
  - 4 = CDATA section node (CDATA is similar to a text node)
  - 5 = entity-reference node
  - 7 = processing-instruction node
  - 8 = comment node
  - 9 = document node (only for input)
  - 10 = document type (DTD) node
- **GET_NODE_NAME** returns the tag name (if any) for the node
- **GET_NODE_VALUE** return the value (if any) for a node. This is especially useful with text and CDATA nodes.

The slide shows an example of a call on **GET_NODE_VALUE**:

- The first parameter is the document object tag.
- The second parameter is the object tag for the node.
- The third parameter is a data item that will receive the node's value.
- The fourth parameter is the zero-relative offset into the third parameter where the value text will start.
- The fifth parameter is a binary data item that will return the length of the value text.



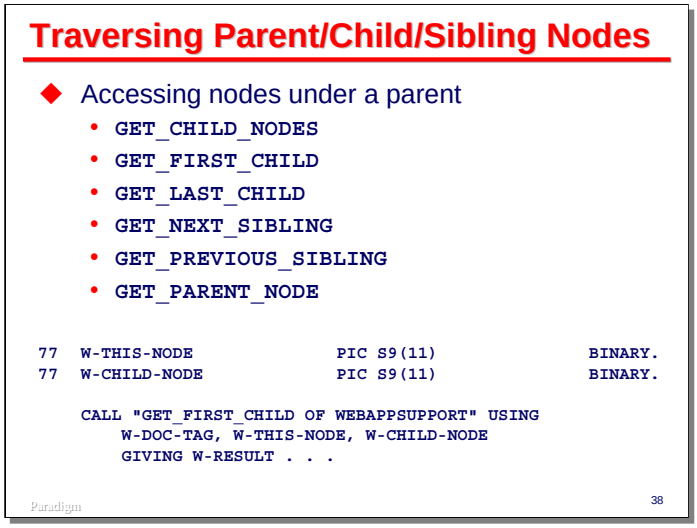

You can also examine the attributes for a node and the values of those attributes:

- **GET_ATTRIBUTE_BY_NAME** searches the attributes for a specified node to find the first one of a given name. Name matching is case sensitive.
- **GET_ATTRIBUTES** returns a COBOL-like table of attribute names and values, similar to the table used with **CREATE_TEXT_ELEMENT**.
- **HAS_ATTRIBUTE** determines whether a specified node has an attribute of a given name.

The slide shows an example of a call on **GET_ATTRIBUTE_BY_NAME**:

- The first parameter is the document object tag.
- The second parameter is the object tag for the node whose attributes will be searched.
- The third parameter is a data item containing the attribute name.
- The fourth parameter is a binary integer that will receive the object tag for the named attribute, if found.

If the attribute is not found, the method returns a zero result.




Since the DOM is structured internally as a tree, the nodes have hierarchical relationships, and the tree can be traversed used parent/child/sibling linkages.

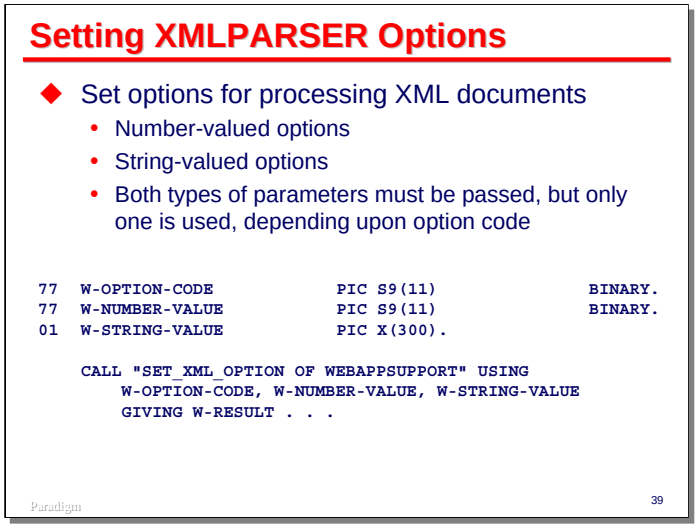
- **GET_CHILD_NODES** returns a list of nodes for all of the children under a specified parent node. The list is similar to that for **GET_ELEMENTS_BY_TAGNAME**.
- **GET_FIRST_CHILD** returns the node object tag for the first child under a specified parent node.
- **GET_LAST_CHILD** returns the node object tag for the last child under a specified parent node.
- **GET_NEXT_SIBLING** returns the node object tag for the next sibling (if any) of a specified node.
- **GET_PREVIOUS_SIBLING** returns the node object tag for the prior sibling (if any) of a specified node.
- **GET_PARENT_NODE** returns the node object tag for a specified node's parent node.

For the **FIRST**, **LAST**, **NEXT**, and **PREVIOUS** calls, if no such node exists, the method returns a zero result.

The slide shows an example of a call on **GET_FIRST_CHILD**:

- The first parameter is the document object tag.
- The second parameter is the object tag for the parent node.
- The third parameter is a binary integer that will return the object tag for the first child, if any.

Each of these methods will return a zero result if no node is returned.

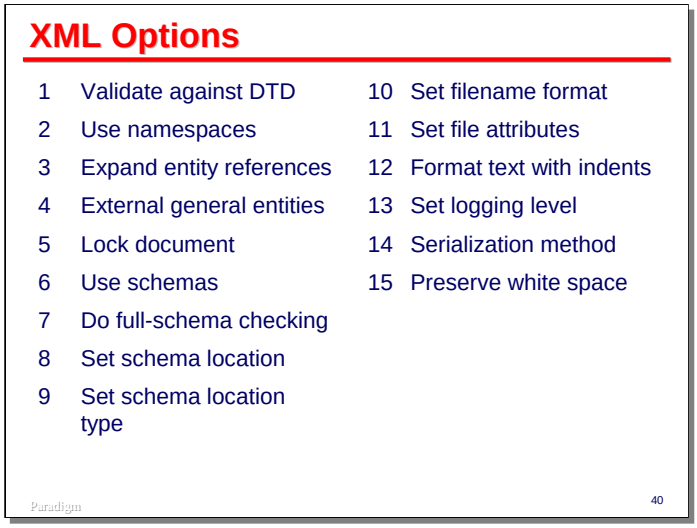



There are a number of global options for the XMLPARSER API that can be changed using the **`SET_XML_OPTION`** method. Some options require a numeric parameter and some require a string parameter. The method has both a numeric and string parameter – which one is used in a particular call depends on the option being set.

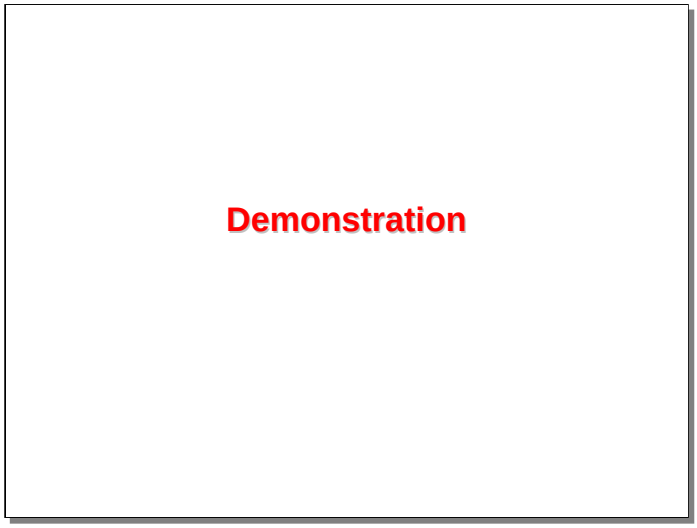
The slide shows a generic example of a call on **`SET_XML_OPTION`**:

- The first parameter is a binary integer containing the option code. The next slide shows the option codes.
- The second parameter is used if the option requires a numeric parameter.
- The third parameter is used if the option requires a string parameter.

See the WEBAPPSUPPORT reference manual for details on which options require which types of parameters.




This slide shows all of the option codes for **`SET_XML_OPTION`** as of MCP 17.




There are two sample COBOL-85 programs in the associated files for this presentation.

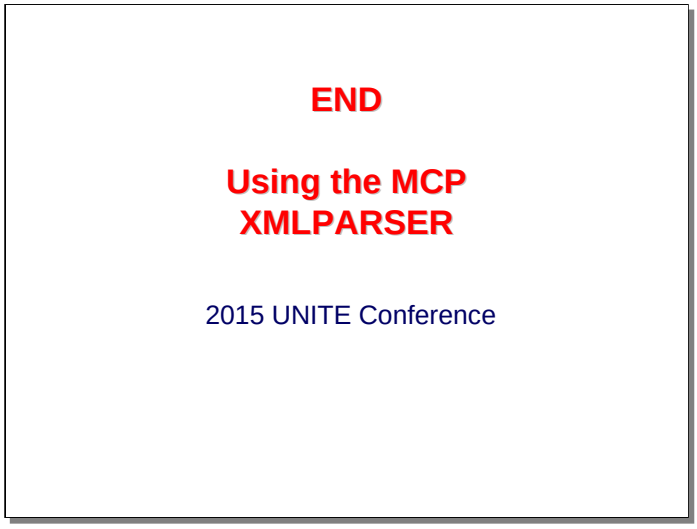
- **`XMLPARSER/CREATEDOC/DEMO`** displays a form that accepts name and address data. It then creates a simple XML document that encapsulates that data, exports the document as text, and displays the XML text.
- **`XMLPARSER/GEONAMES/DEMO`** is an extension to the **`HTTPCLIENT/GEONAMES/DEMO`** example in the MCP 4013 HTTPCLIENT presentation at this conference. The original program used a free web service to search for information about place names. The service returns the result as XML, which the original program simply displayed. The sample for this presentation extends the original by parsing the XML text returned by the web service and formatting the text contained therein. Information about the web service can be found at http://www.geonames.org/.

## References

- *WEBAPPSUPPORT Application Programming Guide* (3826 5286), Sections 4-8

Paradigm

42


END
Using the MCP
XMLPARSER
2015 UNITE Conference