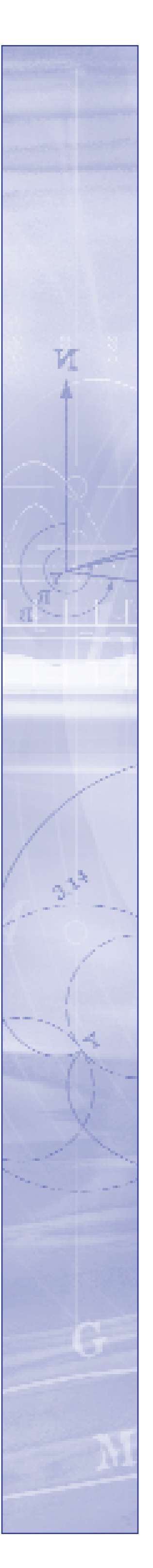


A Guide to avenue.quark 6.0

©2003 Quark Technology Partnership and Quark, Inc. as to the content and arrangement of this material. All rights reserved.

©1999–2003 Quark Technology Partnership and Quark, Inc. as to the technology. All rights reserved. Patents pending.

Information in this document is subject to change without notice and does not represent a commitment on the part of Quark Technology Partnership or its licensee, Quark, Inc.

Quark Products and materials are subject to the copyright and other intellectual property protection of the United States and foreign countries. Unauthorized use or reproduction without Quark's written consent is prohibited.

Quark, QuarkXPress, QuarkXPress Passport, QuarkXTensions, avenue.quark, and XTensions are trademarks of Quark, Inc. and all applicable affiliated companies, Reg. U.S. Pat. & Tm. Off. and in many other countries. The Quark logo is a trademark of Quark, Inc. and all applicable affiliated companies.

Microsoft is a registered trademark of Microsoft Corporation in the United States and/or other countries.

All other trademarks are the properties of their respective owners.

QUARK IS NOT THE MANUFACTURER OF THIRD PARTY SOFTWARE OR OTHER THIRD PARTY HARDWARE (HEREINAFTER "THIRD PARTY PRODUCTS") AND SUCH THIRD PARTY PRODUCTS HAVE NOT BEEN CREATED, REVIEWED, OR TESTED BY QUARK, THE QUARK AFFILIATED COMPANIES OR THEIR LICENSORS. (QUARK AFFILIATED COMPANIES SHALL MEAN ANY PERSON, BRANCH, OR ENTITY CONTROLLING, CONTROLLED BY OR UNDER COMMON CONTROL WITH QUARK OR ITS PARENT OR A MAJORITY OF THE QUARK SHAREHOLDERS, WHETHER NOW EXISTING OR FORMED IN THE FUTURE, TOGETHER WITH ANY PERSON, BRANCH, OR ENTITY WHICH MAY ACQUIRE SUCH STATUS IN THE FUTURE.)

QUARK, THE QUARK AFFILIATED COMPANIES AND/OR THEIR LICENSORS MAKE NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE QUARK PRODUCTS/SERVICES AND/OR THIRD PARTY PRODUCTS/SERVICES, THEIR MERCHANTABILITY, OR THEIR FITNESS FOR A PARTICULAR PURPOSE. QUARK, THE QUARK AFFILIATED COMPANIES AND THEIR LICENSORS DISCLAIM ALL WARRANTIES RELATING TO THE QUARK PRODUCTS/SERVICES AND ANY THIRD PARTY PRODUCTS/SERVICES. ALL OTHER WARRANTIES AND CONDITIONS, WHETHER EXPRESS, IMPLIED OR COLLATERAL, AND WHETHER OR NOT, MADE BY DISTRIBUTORS, RETAILERS, XTENSIONS DEVELOPERS OR OTHER THIRD PARTIES ARE DISCLAIMED BY QUARK, THE QUARK AFFILIATED COMPANIES AND THEIR LICENSORS, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF NON-INFRINGEMENT, COMPATIBILITY, OR THAT THE SOFTWARE IS ERROR-FREE OR THAT ERRORS CAN OR WILL BE CORRECTED. THIRD PARTIES MAY PROVIDE LIMITED WARRANTIES AS TO THEIR OWN PRODUCTS AND/OR SERVICES, AND USERS MUST LOOK TO SAID THIRD PARTIES FOR SUCH WARRANTIES, IF ANY. SOME JURISDICTIONS, STATES OR PROVINCES DO NOT ALLOW LIMITATIONS ON IMPLIED WARRANTIES, SO THE ABOVE LIMITATION MAY NOT APPLY TO PARTICULAR USERS.

IN NO EVENT SHALL QUARK, THE QUARK AFFILIATED COMPANIES, AND/OR THEIR LICENSORS BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL, CONSEQUENTIAL OR PUNITIVE DAMAGES, INCLUDING, BUT NOT LIMITED TO, ANY LOST PROFITS, LOST TIME, LOST SAVINGS, LOST DATA, LOST FEES, OR EXPENSES OF ANY KIND ARISING FROM INSTALLATION OR USE OF THE QUARK PRODUCTS/SERVICES, IN ANY MATTER, HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY. IF, NOTWITHSTANDING THE FOREGOING, QUARK, THE QUARK AFFILIATED COMPANIES AND/OR THEIR LICENSORS ARE FOUND TO HAVE LIABILITY RELATING TO THE QUARK PRODUCTS/SERVICES OR THIRD PARTY PRODUCTS/SERVICES, SUCH LIABILITY SHALL BE LIMITED TO THE AMOUNT PAID BY THE USER TO QUARK FOR THE SOFTWARE/SERVICES AT ISSUE (EXCLUDING THIRD PARTY PRODUCTS/SERVICES), IF ANY, OR THE LOWEST AMOUNT UNDER APPLICABLE LAW, WHICHEVER IS LESS. THESE LIMITATIONS WILL APPLY EVEN IF QUARK, THE QUARK AFFILIATED COMPANIES, THEIR LICENSORS AND/OR THEIR AGENTS HAVE BEEN ADVISED OF SUCH POSSIBLE DAMAGES. SOME JURISDICTIONS, STATES OR PROVINCES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THIS LIMITATION OR EXCLUSION MAY NOT APPLY. ALL OTHER LIMITATIONS PROVIDED UNDER APPLICABLE LAW, INCLUDING STATUTES OF LIMITATION, SHALL CONTINUE TO APPLY.

IN THE EVENT ANY OF THESE PROVISIONS ARE OR BECOME UNENFORCEABLE UNDER APPLICABLE LAW, SUCH PROVISION SHALL BE MODIFIED OR LIMITED IN ITS EFFECT TO THE EXTENT NECESSARY TO CAUSE IT TO BE ENFORCEABLE

USE OF THE QUARK PRODUCTS IS SUBJECT TO THE TERMS OF THE END USER LICENSE AGREEMENT OR OTHER APPLICABLE AGREEMENTS FOR SUCH PRODUCT/SERVICE. IN THE EVENT OF A CONFLICT BETWEEN SUCH AGREEMENTS AND THESE PROVISIONS, THE RELEVANT AGREEMENTS SHALL CONTROL.

A Guide to avenue.quark 6.0

Chapter 1: Installing and Customizing avenue.quark

Minimum System Requirements 6

Installation Instructions 6

Customizing avenue.quark 7

Chapter 2: Avenue.quark Basics

Introduction to XML 10

Understanding XML 12

Working with XML 14

Working with DTDs 19

Industry-Standard DTDs 42

Chapter 3: XML Workspace Palette

XML Workspace Palette 45

Chapter 4: Menus and Dialog Boxes

Preference Settings 57

Edit Menu 64

Utilities Menu 71

Choose Rule/Position Dialog Box 71

Chapter 5: Tagging Rule Sets

Understanding Rule-Based Tagging 75

Working with Tagging Rule Sets 78

Chapter 6: Tagging Content

Creating, Opening, and Saving XML Documents 82

Working with XML Templates 87

Working with XML Document Content 90

Tagging Text 93

Tagging Pictures 96

Manually Entering New Content 96

Previewing Tagged Text 97

Appendices

Appendix A: XML Quick Reference 98
Appendix B: DTD Quick Reference 101
Appendix C: Understanding Encodings 105
Appendix D: Sample avenue.quark Scenario 107

Chapter 1: Installing and Customizing avenue.quark

Once you install avenue.quark™ software, you can customize avenue.quark for your workflow. You can control the color of tagged and untagged content (**Utilities** & **Show Tagged Content**), set the color of the XML markers that display when you choose **View** & **Show Invisibles**, turn dynamic content updating on and off, and control certain elements of tagging.

MINIMUM SYSTEM REQUIREMENTS

- QuarkXPress™ or QuarkXPress Passport™ 6.0 software or later

INSTALLATION INSTRUCTIONS

Avenue.quark software is automatically installed by the QuarkXPress or QuarkXPress Passport Installer. If you need to reinstall the QuarkXTensions™ software, avenue.quark, follow these steps:

FOR MAC OS

1. Quit QuarkXPress or QuarkXPress Passport.
2. On the QuarkXPress CD-ROM, locate the “avenue.quark” file.
3. Copy the “avenue.quark” file into the “XTension” folder within your QuarkXPress or QuarkXPress Passport application folder.
4. Launch QuarkXPress or QuarkXPress Passport to access the features of avenue.quark.

FOR WINDOWS

1 Exit QuarkXPress or QuarkXPress Passport.

2 On the QuarkXPress CD-ROM, locate the “avenue.quark.xnt” file.

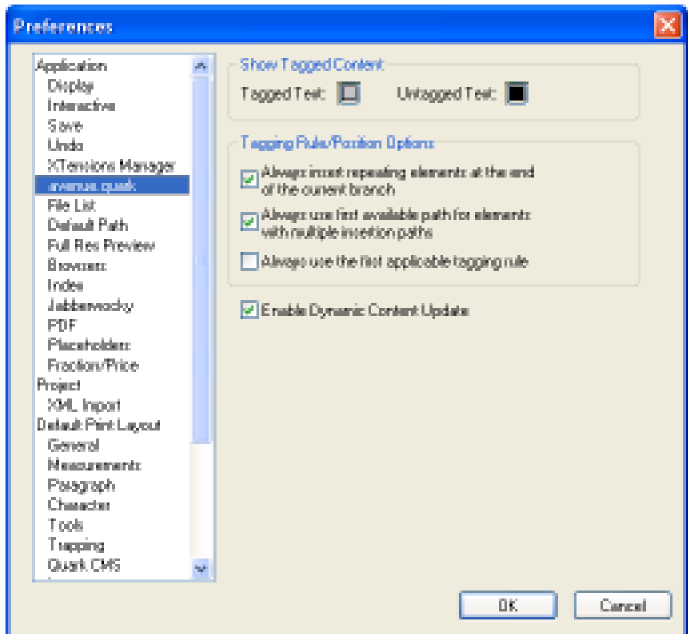
3 Copy the “avenue.quark.xnt” file into the “XTension” folder within your QuarkXPress or QuarkXPress Passport application folder.

4 Launch QuarkXPress or QuarkXPress Passport to access the features of avenue.quark.

CUSTOMIZING AVENUE.QUARK

Avenue.quark uses several default settings to control how tagged content displays, specify the color of marker text, and turn dynamic content updating on and off. These settings are saved with the application and are never saved with projects. To modify avenue.quark preferences:

1 Choose **QuarkXPress** & **Preferences** & **avenue.quark** on Mac OS or **Edit** & **Preferences** & **avenue.quark** on Windows to display the **avenue.quark** pane.



Use the avenue.quark pane to specify how tagged text displays in a project.

2 To specify the color that is used to display tagged content when you choose **Utilities** & **Show Tagged Content**, click the **Tagged Text** button and then choose a color in the resulting dialog box.

3 To specify the color that is used to display untagged content when you choose **Utilities** & **Show Tagged Content**, click the **Untagged Text** button and then choose a color in the resulting dialog box.

4 Use the **Tagging Rule/Position Options** to control certain elements of tagging:

- The **Always insert repeating elements at the end of the current branch** check box controls the placement of new repeating elements (elements marked with a + or * in the DTD). When this box is checked, avenue.quark always puts new repeating elements at the end of the active branch. When this check box is unchecked, avenue.quark displays the **Choose Rule/Position** dialog box and lets you manually choose the position of a new repeating element.
- The **Always use first available path for elements with multiple insertion paths** check box controls the placement of new elements that could be inserted in a number of places according to the DTD. For example, say a tagging rule calls for the creation of a <paragraph> element. If the DTD states that a new <paragraph> element may be created either at the end of the current branch or as a child of a new <sidebar> element, which kind of <paragraph> element should avenue.quark generate? If this check box is checked, avenue.quark creates the first <paragraph> element it finds in the DTD tree (a <paragraph> element at the root level of the current branch). If this check box is unchecked, avenue.quark displays the **Choose Rule/Position** dialog box.
- The **Always use the first applicable tagging rule** check box applies to tagging rule conflicts. When this box is checked, avenue.quark always chooses the first of a series of applicable rules in tagging rule conflicts. When this box is unchecked, avenue.quark displays the **Choose Rule/Position** dialog box so you can either manually choose the element type to be applied to the selected text or click **Choose Automatically.**

5 To specify that the content of elements in active XML documents should be continuously updated to reflect the content of the QuarkXPress items they're linked to, check **Enable Dynamic Content Update**. You might want to uncheck this button if QuarkXPress seems to be running very slowly when editing large XML documents; you can manually update the content when this box is unchecked by clicking the **Synchronize Content** button in the **XML Workspace** palette.

The dynamic content updating feature works for elements, but not for attributes.

6 To specify the color of the markers that display when you choose **View** & **Show Invisibles**, display the **Preferences** dialog box **Placeholders** pane (**QuarkXPress** & **Preferences** on Mac OS or **Edit** & **Preferences** on Windows), click the **Color** button in the **Placeholder Descriptor** area, and then choose a color in the resulting dialog box. Enter or choose a shade percentage in the **Shade** field. Click **OK**, then click **OK** again to close the **Preferences** dialog box.

Placeholder Descriptor

Color:

Shade: 100%

Use the Placeholder Descriptor area of the Preferences dialog box Placeholders pane to specify the color of text markers.

7 Click **OK**.

Chapter 2: Avenue.quark Basics

Many people want to use the Web to publish information they create in QuarkXPress format. A very efficient way is to separate the content of QuarkXPress projects from the projects themselves, and store that content in a structured format such as XML. Then you can re-use the content not only on the Web, but in other formats as well — print, CD-ROM, you name it. Avenue.quark software makes it easy for you to extract your QuarkXPress content and store it in XML format.

INTRODUCTION TO XML

Avenue.quark lets you extract the content of QuarkXPress projects and store that content in XML format. You can then easily re-use the content in a variety of ways, including on the Web. This section briefly explains the process and definitions; more detailed descriptions follow in subsequent sections.

WHAT IS CONTENT?

Content is the information that makes your projects valuable. For example, the content of a magazine may include articles, photographs, interviews, and diagrams.

Content can also be defined by what it is *not*. For example, headers, footers, and "Continued on page x" notes are generally *not* considered to be part of a magazine's content. Rather, they're part of the magazine's presentation — aspects of the magazine that are desirable only when the magazine is being presented in printed format. Presentation may change depending on the medium information is published in, but content generally stays the same.

Avenue.quark lets you separate content from presentation by extracting that content from your QuarkXPress projects and storing it in XML format. Then you can re-use that content with different presentations — in print, on the Web, on CD-ROM, and so forth.

WHAT IS XML?

XML stands for Extensible Markup Language. XML is a way for you to specify the structure of content and label the pieces of that content in a meaningful way.

LABELING CONTENT

Why do we need to label content? Because although we can pick up a magazine and know that a particular line of text is a headline, such distinctions aren't so

easy for a computer. XML lets you label ("tag") information in a way that computers can understand; once a computer understands that a particular line of text is a headline, it can automatically format that line as a headline.

To label a piece of content in XML, you insert an opening XML tag before the content and a closing XML tag after the content, like this:

```
<headline>Internet Grows by 400%</headline>
```

As you can see, an opening tag consists of an element name between a < and a >. A closing tag is the same, with a / after the <. Here, we've tagged the text "Internet Grows by 400%" as a headline by putting it between opening and closing <headline> tags.

IDENTIFYING STRUCTURE

We know that a news story generally consists of a headline, a byline, body text, and some photos or diagrams with captions. However, computers don't know such things until you tell them.

XML lets you describe the structure of your layouts with DTDs (document type definitions). A DTD specifies that the information in a layout will use a particular set of tags and follow a particular set of structural rules. For example, a DTD for a news story might specify that:

- Each story must have exactly one <headline>.
- Each story may or may not have a <byline>.
- Each story must have at least one <paragraph>.
- Each story may have zero or more <illustration> elements.
- Each illustration must be immediately followed by exactly one <caption>.

By consistently adhering to the rules of a DTD, an organization can ensure that its documents are structured predictably and consistently. This makes it much easier for organizations to move content from one medium to another — for example, from print to the Web, or vice versa.

Avenue.quark *requires* the use of DTDs. For information on creating and adapting DTDs, see "Working With DTDs" and "Industry-Standard DTDs" in this chapter.

A NEUTRAL FORMAT

XML is a "neutral" format in that it contains no information about formatting. This means it can be used with a wide variety of applications, which can apply different kinds of formatting when the content is presented through different kinds of media. For a more detailed discussion of XML, see "Understanding XML."

WHAT CAN I DO WITH CONTENT STORED IN XML FORMAT?

Once you've extracted the content from a QuarkXPress layout, you can use that content in a variety of ways. For example, you can dynamically translate XML-tagged content into HTML format and serve it on the Web. This method of converting QuarkXPress content to HTML is superior to simple HTML export because it lets you easily format, reformat, and reorganize the content.

UNDERSTANDING XML

Now that you have a general idea of what avenue.quark is and how it works, let's take a look at the details, beginning with a look at XML.

XML (Extensible Markup Language) is a way of specifying the structure of documents and labeling specific pieces of content with tags. XML's structural controls let you make sure that all the necessary parts of that document are present and occur in the proper order. Labeling content makes it easy for other applications to use or display that content.

Before we consider how XML accomplishes all of this, let's talk about why it's necessary.

THE PROBLEMS XML SOLVES

XML was derived from an older and more complicated markup language, SGML (Standard Generalized Markup Language). XML was created to solve a variety of related problems, some of which were originally solved by SGML, others of which are unique.

ASSIGNING STRUCTURE AND LABELS TO INFORMATION

XML is sometimes referred to as a meta-language because it lets you define customized markup languages for specific uses. You do this by creating a DTD (document type definition). A DTD specifies what kind of information may go into a document, how the parts of the document should be tagged (labeled), the order in which the parts should occur, and how many of each part are allowed. A document is considered valid according to a given DTD only if it follows that DTD's rules.

DTDs let you enforce the structure of document. If you have a document's DTD, you know what kind of information to expect when you display that document. DTDs also make the information in XML documents easy for computers to process; if a computer can understand a DTD, it can understand the information in any XML document that adheres to that DTD. For example, using a document's DTD, a computer program might let you search through every occurrence of a particular type of information (such as company name) in that document, or produce an HTML page that lists all occurrences of that type of information (for example, a list of company names).

Specialized DTDs have already been developed for chemistry, mathematics, technical documentation, and even fictional works. Potential applications include workflow control, software specification, and just about any other field of endeavor that involves the exchange of structured information.

Unlike SGML, XML lets you create *well-formed* documents — that is, documents that follow the rules of XML but do not follow a particular DTD. However, it's difficult to maintain consistency among documents if you do not have a standard, so avenue.quark requires you to use DTDs.

MAKING SENSE OF HTML

HTML has proved to be a powerful and versatile format for displaying information on the World Wide Web. However, it has two major shortcomings: it describes only the formatting of data, not its meaning, and you cannot create new HTML tags.

XML solves both of these problems. If you use XML to label data in an XML document, you can then base the HTML formatting on those labels. For example, say you have an XML document that includes a list of companies and some information about each of those companies. To transform this list into an HTML Web page in which every company name is bold, you can use an XML-to-HTML converter, and instruct the converter to format every line that's tagged as a `<companyName>` as bold text. This means you no longer have to go through and format each company name and address manually. This can save Web site creators enormous amounts of time.

INFORMATION EXCHANGE

Because computer applications have been developed by many people and organizations for many different uses, they store information in many different formats. For example, two companies may store their customer information in two completely different formats, even though the customer information stored by the companies (name, address, phone number, and so forth) is basically identical.

XML solves this kind of problem by providing a standardized, nonproprietary format for the transfer of information between applications. XML was developed, refined, and approved by a group of professionals from different industries working together as part of the World Wide Web Consortium (the W3C). The specification is available to anyone who wants to use it (see www.w3.org), and many organizations and industries already do.

If two companies use software that can convert their records to XML with an agreed-upon DTD, they can exchange those records with no risk of data loss due to incompatible formats. For more information on DTDs and information exchange, see "Industry-Standard DTDs" in this chapter.

For a more in-depth discussion of XML, see "Working with XML" in this chapter.

For a detailed explanation of the XML 1.0 specification, see *XML: The Annotated Specification*, by Bob DuCharme or *XML: A Primer*, by Simon St. Laurent.

WORKING WITH XML

An XML document contains structured data that has been broken down into elements, each of which is described with XML tags.

XML ELEMENTS AND XML TAGS

An XML element contains a piece of information, such as a company name, a headline, or a part number. You create an element by putting a piece of information between two XML tags: an opening tag containing the element's name between a less-than (<) symbol and a greater-than symbol (>), and a closing tag that is the same except for the inclusion of a slash (/) before the element name. For example, a tagged "name" element might look like this:

```
<name>Gertrude</name>
```

It's important to understand the difference between an XML element and an XML tag. An XML tag is simply the label that is attached to a piece of information; an XML element includes both the piece of information and the tags that surround it.

XML tags let you describe and add structure to the data they surround. For instance, the following introductory paragraph is tagged with an <introduction> tag:

```
<introduction>
  Frank Lloyd Wright was one of America's finest and most celebrated architects
  Here is the story of his life.
</introduction>
```

Within the <introduction> element, you can tag other sub-elements to add more structure to your document:

```
<introduction>
  <name>Frank Lloyd Wright</name> was one of America s finest and most
  celebrated <job>architects</job>. Here is the story of his life.
</introduction>
```

Syntax is important for XML tags. Unlike HTML tags, they are case-sensitive; a <Name> tag is different from a <name> tag, which is different from a <NAME> tag. Each XML tag name must begin with a letter or an underscore (_); subsequent characters in the name may be letters, underscores, numbers, hyphens, and periods, but not spaces or tabs. For example, the XML tag name <_.dir> is a correctly formed tag name, but the names <_ dir> and <.dir> are not. The <_ dir> tag name is incorrect because it contains white space (a tab or space) after the underscore. The <.dir> tag name is incorrect because it begins with a period instead of an underscore or letter.

It's useful to know the difference between "elements" and "element types." An *element type* can be thought of as a specific tag name that can be applied to data; an *element* is a piece of data and the tags that surround it. For example, a document containing a list of names and addresses might have only two element types, <name> and <address>, but hundreds of elements that use those tags.

Avenue.quark supports XML namespaces. Element names that include namespace prefixes (such as <HTML:H1>) are handled in the same way as element names without prefixes.

XML ATTRIBUTES

Let's say you're working with elements tagged as <textbook>, and you want to be able to specify additional information about each <textbook> element you create. For example, say you want to designate a specific <textbook> element not just as a textbook, but with the corresponding course number and author name and whether the book is required or optional.

There are several ways you could do this. One way involves creating additional element types, like this:

```
<textbook>
<course> ENGL 3500 </course>
<category> Required </category>
<author> William Styron </author>
Lie Down in Darkness
</textbook>
```

Another, perhaps more streamlined, way to do this is by using an XML feature called attributes. Attributes are designed to provide information about an element. They are included within an element's start tag, so there is never any doubt about which element they are related to.

An attribute consists of an attribute name, followed by an equals sign, followed by an attribute value between quotation marks. For example, the following

single element uses three attributes to provide the same information as the example above:

```
<textbook course="ENGL 3500" category="required" author="William Styron" > Lie
Down in Darkness </textbook>
```

Attributes are useful for several reasons. For example, they make it easy to search a document and generate a list of all the <textbook> elements that contain the value “required” in the category attribute. They can also be useful in conjunction with empty elements; see the next section for details.

EMPTY ELEMENTS

Empty elements include a start tag and an end tag, and do not surround any data, like this:

```
<IDnumber></IDnumber>
```

Since empty elements have no content between their tags, the starting and closing tags are often combined, like this:

```
<IDnumber/>
```

You can use attributes along with empty elements to refer to URLs or externally-stored files. For example, the following empty element could be used (with an appropriate XML interpreter) to display a picture of an author:

```
<authorPicture URL="www.quark.com/picture"/>
```

Adding an attribute named “URL” to an element does *not* guarantee that the URL will be accessed when the XML file is processed. The application that processes the file must know what to do with the URL attribute.

COMMENTS

Just as in HTML, you can include comments in an XML file. Comments are bracketed by <!— and —>, and are essentially ignored by XML processors. So, for example, to insert a comment about the status of an <address> element, you could do the following:

```
<address>
  <!-- Waiting to get this address from Accounting. -->
</address>
```

In this manual, comments always display in red for easy identification.

PROCESSING INSTRUCTIONS

In HTML, comments are often used to contain special commands for browsers and other HTML processors. In an effort to restrict XML comments to being

just that — comments — the authors of the XML specification have included a method for inserting customized commands in XML files and DTDs. Such customized commands, called processing instructions (or “PIs”), are enclosed between a `<?` and a `?>`. They begin with an application name, followed by a space and any information that might be of interest to the named application. Processing instructions can be used anywhere that comments can display.

When adding comments and processing instructions using the XML Workspace palette, remember that avenue.quark adds the opening and closing tags automatically. Adding these tags manually can lead to problems.

According to the XML 1.0 specification, it is illegal to include a closing comment tag (`-->`) in a comment. It is also illegal to include a closing processing instruction tag (`?>`) in a processing instruction. If you do either of these things, avenue.quark will not be able to reopen the resulting XML document.

XML DECLARATION

Each XML document should begin with an XML declaration. Like a processing instruction, an XML declaration is enclosed between a `<?` and a `?>`. Here’s an example of an XML declaration:

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

The `version` attribute declares that this document adheres to the rules of XML 1.0. The `standalone` attribute indicates that all markup declarations needed to process this XML document are included in the document.

ENTITY REFERENCES

An entity reference is a word that serves as shorthand for a character, string, or file. For example, by using the `<` entity reference to represent a less than character (<) in the content of an XML document, you can avoid confusing the XML parser (which would otherwise erroneously read the “<” character as the beginning of a tag). For more information on entity references, see “Entity References” in the “Working with DTDs” section of this chapter.

WELL-FORMED XML

For an XML document to be well-formed, it should begin with an XML declaration and have a root element that contains all of its other elements (`<article>` in the example below). Well-formed XML also requires every element in the document to have a corresponding end tag. The following is an example of a well-formed XML document:

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

<article>
```

```
  <newsflash>
    <title>Forney Museum to Close</title>
    <author>Linda Spano</author>
    <content>
      The Forney Transportation Museum closes its doors next week.
    </content>
  </newsflash>
</article>
```

VALID XML

A well-formed XML document can be limited in its usefulness unless it is also valid. An XML document is considered to be valid when it adheres to the specifications of a specific DTD. For more information about DTDs and validating XML documents, see "Working with DTDs" in this chapter.

XML PROCESSORS

An XML processor is a program that reads an XML file and does something with it. There are various kinds of XML processors. An XML processor might convert an XML file into an HTML Web page, a PDF file, or a PostScript file. It might read the XML file's content out loud, or convert the content to Braille. An XML processor could even be used to copy structured XML content into a database.

XML PARSERS

An XML parser recognizes the rules of XML and checks to see if an XML document is well-formed. However, an XML parser does *not* necessarily check to see if an XML document is valid according to its DTD; this requires a validating XML parser (see below).

VALIDATING XML PARSERS

Validating XML parsers compare an XML document to a DTD and verify whether the document conforms to the DTD's rules. A good validating parser will also provide constructive feedback about any problems it finds in the XML file. For more information about XML parsers, see "Working with DTDs" in this chapter.

For a quick reference to XML features and conventions, see Appendix A, "XML Quick Reference," in Chapter 7, "Appendices."

WORKING WITH DTDS

A DTD (document type definition) specifies which elements an XML file may contain and how those elements must be structured. XML documents don't necessarily have to have a corresponding DTD; as long as an XML file follows basic XML syntax, it's

considered to be well-formed and can be read by an XML-savvy application. However, an XML file can only be considered valid if it adheres to a particular DTD.

DTDs are important because they provide a reliable, well-documented structure for XML documents. Without DTDs, two organizations that work together might decide to structure and tag their XML documents in entirely different ways; thus, their data stores would remain incompatible even after they both have made the transition to XML. However, if both organizations have the same DTD — perhaps a DTD they developed together, or a DTD that has become standard in their industry — they can easily and predictably exchange information.

EXTERNAL AND INTERNAL DTDS

There are two kinds of DTDs: external DTDs and internal DTDs. Technically, a DTD consists of the list of markup declarations (element declarations, attribute declarations, entities, notations, processing instructions, and comments) that is referenced by a `DOCTYPE` declaration. What this document refers to as "external DTDs" and "internal DTDs" are not technically complete DTDs; however, it is convenient and fairly common to refer to them as such.

EXTERNAL DTDS

An external DTD (or external subset) is a file containing a list of markup declarations. External DTDs are easy to share between XML documents and organizations. To use an external DTD in an XML file, refer to it at the beginning of the XML file, like this:

```
<?xml version="1.0" standalone="no"?>
<!— The following line specifies a root element (<myDocument>) and points to the
    URL of an external DTD file named
    "mydocument.dtd" —>
<!DOCTYPE myDocument SYSTEM "http://www.quark.com/mydocument.dtd">
<!-- Document begins here -->
<myDocument>
  When laws are outlawed, only outlaws will follow the rules.
</myDocument>
```

INTERNAL DTDS

An internal DTD (or *internal subset*) is actually included in the XML file it describes. To use an internal DTD in an XML file, you simply add it to the beginning of the XML file, like this:

```
<?xml version="1.0" standalone="yes"?>
<!— The following line specifies a root element (<myDocument>) and signifies the
    beginning of the DTD -->
<!DOCTYPE myDocument [
```

```
  <!-- Internal DTD begins here -->
  <!ELEMENT myDocument ANY>
<!-- End of DTD -->
]>
<!-- Document begins here -->
<myDocument>
  When laws are outlawed, only outlaws will follow the rules.
</myDocument>
```

If a document uses an external DTD (or any other sort of external entity), the `standalone` attribute in the first line must be set to `no`. For more information, see “Using entity references” in this section.

COMBINING INTERNAL AND EXTERNAL DTDS

In a given XML document, you can specify an external DTD, then add to or override that DTD with an internal DTD. Here’s how such an XML document might look.

```
<?xml version="1.0" standalone="no"?>
<!— The following line specifies a root element (<myDocument>), points to the
    URL of an external DTD file named "mydocument.dtd", and then signifies the
    beginning of the internal DTD -->
<!DOCTYPE myDocument SYSTEM "http://www.quark.com/mydocument.dtd" [
  <!— Internal DTD goes here; it may add new element types in addition to the
      element types defined in the external DTD -->
  <!ELEMENT myLocalDTDelement ANY>
<!-- End of DTD -->
]>
<!-- Document begins here -->
<myDocument>
  <myLocalDTDelement>
    When laws are outlawed, only outlaws will follow the rules.
  </myLocalDTDelement>
</myDocument>
```

PLANNING A DTD

You probably don’t want to just sit down and start writing a DTD; it requires considerable planning if you want to do it right. Before you begin the process of creating your own DTD, you may want to consider using an industry-standard

DTD. For more information on this option, see “Industry-Standard DTDs” in this chapter.

A good way to begin is to figure out what exactly you want your DTD to do. First, decide which elements you want to use. If you want to use elements such as <address>, think about whether you want to subdivide those elements into subelements such as <streetAddress>, <unitNumber>, <city>, <state>, and <ZIPcode>. (Give such subdivisions serious consideration if there’s any chance that you may one day transfer the contents of your XML files into a database.)

So much for the easy part. Next, you need to figure out the relationships between all these elements. A DTD can specify which elements are allowed, what order they must be in, and which (and how many) sub-elements they may contain. It can specify which other elements can contain a given element, and it can specify whether a given element must contain data or not.

Elliotte Rusty Harold, in *XML: Extensible Markup Language*, recommends using a table to help you figure out the relationships between the various elements in your DTD. The table should have the following columns (the data in the columns is provided as an example only):

ELEMENT NAME	MUST CONTAIN	MAY CONTAIN	MUST BE CONTAINED BY
<course>	<title>,<author>, <category>	<publisher> <new price>, <used price>	<FallBookList>
<title>			<course>

Each row in the table should represent an element that you want to use in your DTD.

CREATING AN INTERNAL DTD

To better understand how a DTD is constructed, you can create a very simple internal DTD. Even if this DTD does not serve your publishing needs, the act of creating one will reinforce the concepts in the following sections. This internal DTD lists all books for fall semester courses. To create this DTD:

1. Open a text editor such as SimpleText (Mac OS) or WordPad (Windows).
2. In the new text document, enter the following:

```
<!ELEMENT FallBookList (course, title+, author+, category, newprice, usedprice)>
```

   This establishes the root element of the DTD. Think of the root element as the main element; all other elements will exist *within* the root element. This line also states that elements contained in the root element can be of any type.

3. Now that you have the root element defined, you’ll need to define further elements. Enter the following:

```
<!ELEMENT course (#PCDATA)>
```

This defines the element `course`, and by not adding a symbol after the element name, you are indicating that there can be only one instance of this element (per book).

4 Now you can add the title and author by entering the following:

```
<!ELEMENT title+ (#PCDATA)>
<!ELEMENT author+ (#PCDATA)>
```

Although a book only has one title, one course may have several books, and one book may have several authors or editors, so adding the + symbol after `title` and `author` allows a course to contain one or more titles, and for a book to have one or more authors.

5 At this point, you can use an element such as `category` to indicate whether the book is required or optional. Enter the following:

```
<!ELEMENT category (#PCDATA)
```

Since the book can only be required or optional, you only need one instance of this element. Not adding a symbol after the element name indicates that this element can only occur once per book.

6 Now you're ready to add the prices. A book only has one price, but you may want to create two elements, one for the price of a new book and one for the price of a used book. Enter the following:

```
<!ELEMENT newprice (#PCDATA)
<!ELEMENT usedprice (#PCDATA)
```

7 Save the text document as a plain text (ASCII) file, and name it "BookList.dtd." You have just created a DTD that can be used as an internal DTD in conjunction with avenue.quark.

This exercise introduced you to the basics of creating a DTD; details and advanced concepts are explained below.

UNDERSTANDING A DTD

Like an XML file, a DTD consists of plain text. An XML file may use no DTD, an external DTD, an internal DTD, or both an external and an internal DTD.

Regardless of which type of DTD an XML document uses, it must refer to or include that DTD in its prologue (opening section), just after the XML declaration and before the body of the XML document. The DTD section begins with `<!DOCTYPE rootname [` and ends with `]>`. Here, for example, is a complete XML document containing a complete DTD (in bold type):

```
<?xml version="1.0" standalone="yes"?>
<!-- DTD begins here -->
```

```
<!DOCTYPE message [
 <!ELEMENT message ANY>
]>
<!-- Document begins here -->
<message>
 When laws are outlawed, only outlaws will follow the rules.
</message>
```

Let's break that down a bit:

- The root name (the word after `<!DOCTYPE`) specifies the root element type of the XML file.
- The square brackets indicate and enclose the internal DTD.
- `<!ELEMENT message ANY>` defines an element type named "message." `!ELEMENT` means "The following is an element type declaration." Next comes the name of the element (`message`) and information about what can be contained by this element (in this case, `ANY`, meaning both text and additional elements).

As you can see, each element type definition specifies both the element's name and the kind of data that element may contain. If you wanted to change the element type definition for `<message>` so that it could contain text and only text (that is, no other elements), you could do so by changing the `ANY` keyword to `(#PCDATA)`, like this:

```
<?xml version="1.0" standalone="yes"?>
<!-- DTD begins here -->
<!DOCTYPE message [
 <!ELEMENT message (#PCDATA)>
]>
<!-- Document begins here -->
<message>
 When laws are outlawed, only outlaws will follow the rules.
</message>
```

However, you probably wouldn't want to do this, since that would mean your document's root element could contain *only* parsed character data; you wouldn't be able to add more elements to subdivide the information.

"PCDATA" stands for "parsed character data": that is, text that may include entity references, comments, and processing instructions.

Let's take a look at a more complex DTD. The following DTD defines a structure for a directory of branch offices:

```
<!-- Root element is <branchOfficeDirectory> -->
<!ELEMENT branchOfficeDirectory ANY>
<!ELEMENT streetAddress (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT postalCode (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT fax (#PCDATA)>
<!ELEMENT eMail (#PCDATA)>
```

Note that we've inserted a comment indicating that <branchOfficeDirectory> is the root element of the DTD. We did this because a DTD can't explicitly designate a root element; specifying the root element is technically the job of the !DOCTYPE line in an XML document. But it's a good idea to specify root elements with a comment so users of the DTD can see what they are.

Some DTDs may contain more than one element that can serve as a root element. For example, you can write a DTD that contains definitions for both white paper documents and FAQ documents, then use that DTD to create both kinds of document simply by specifying <whitePaper> or <FAQ> as the root element of each XML file.

The remaining lines in the DTD declare elements for each office's street address, city, state, postal code, country, phone number, fax number, and e-mail address.

CONTROLLING TAG SELECTION AND ORDER

The above DTD might work just fine for you, but it doesn't really take advantage of XML's features. For example, it doesn't specify any means of indicating which address elements go with which offices, and it doesn't specify any particular order for the information. So you could create a document that lists all the cities, streets, phone numbers, and so forth in random order, and it would still be valid according to this DTD.

To give the DTD a meaningful structure, you need a way to tie all the component elements for each listing together and put them in a particular order. One way to do this is by creating a *container element* to contain the relevant information for one office (we'll call it <branchOffice>), and then specifying which subelements must make up that container element and the order in which they must fall. We can do all of this by adding one line to the DTD (in bold type):

```
<!-- Root element is <branchOfficeDirectory> -->
<!ELEMENT branchOfficeDirectory ANY>
<!ELEMENT streetAddress (#PCDATA)>
```

```
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT postalCode (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT fax (#PCDATA)>
<!ELEMENT eMail (#PCDATA)>
<!ELEMENT branchOffice (streetAddress, city,state, postalCode, country,
  phone, fax, eMail)>
```

What this new element says is, “If the document contains an element named <branchOffice>, that element must contain exactly one of each of the following elements, in this order, and nothing else.”

What if some of your branch offices have more than one line in their street address? You can allow one or more occurrences of any element in a list of subelements by adding + to the end of the element’s name. For example, to allow one or more <streetAddress> elements in our <branchOffice> element, we could do the following:

```
<!ELEMENT branchOffice (streetAddress+, city,state, postalCode, country,
  phone, fax, eMail)>
```

What if some of your branch offices don’t have fax machines? What if some of them have more than one? To specify zero or more occurrences of an element, add * to the end of the element name, like this:

```
<!ELEMENT branchOffice (streetAddress+, city,state, postalCode, country,
  phone, fax*, eMail)>
```

What if some of your branch offices are in a country that does not use postal codes? To specify that zero or one occurrences of a given element may occur, add a question mark to the end of the element’s name, like this:

```
<!ELEMENT branchOffice (streetAddress+, city,state, postalCode?, country,
  phone, fax*, eMail)>
```

What is called a “state” in the United States may have a different name elsewhere. Canada, for example, is divided into provinces. If you have offices in both the United States and Canada, you may want to provide the option of using a <state> element *or* a <province> element. You can do this by putting the two options between a pair of parentheses, separated by a | character, like this:

```
<!ELEMENT branchOffice (streetAddress+, city, (state|province), postalCode?,
  country, phone, fax*, eMail)>
```

Last, you can make sure that a <branchOfficeDirectory> consists of nothing but <branchOffice> listings by changing the definition of <branchOfficeDirectory> from ANY to (branchOffice*). Here's the final product:

```
<!-- Root element is <branchOfficeDirectory> -->
<!ELEMENT branchOfficeDirectory (branchOffice*)>
<!ELEMENT streetAddress (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT province (#PCDATA)>
<!ELEMENT postalCode (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT fax (#PCDATA)>
<!ELEMENT eMail (#PCDATA)>
<!ELEMENT branchOffice (streetAddress+, city, (state|province), postalCode?,
  country, phone, fax*, eMail)>
```

To review:

SYMBOL	MEANING
None	Exactly one
+	One or more
*	Zero or more
?	Zero or one

The special symbols can be used in conjunction with the parentheses to create complex element type declarations such as the following DTD, designed to list contact information on a day-by-day basis:

```
<!-- Root element is <contactSchedule> -->
<!ELEMENT contactSchedule (contactInfo*)>
<!ELEMENT businessPhone (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT businessFax (#PCDATA)>
<!ELEMENT homePhone (#PCDATA)>
<!ELEMENT pager (#PCDATA)>
<!ELEMENT homeFax (#PCDATA)>
<!ELEMENT awayLocation (#PCDATA)>
<!ELEMENT awayPhone (#PCDATA)>
```

```
<!ELEMENT awayFax (#PCDATA)>
<!ELEMENT eMail (#PCDATA)>
<!ELEMENT Message (#PCDATA)>
<!ELEMENT contactInfo (date, ((businessPhone, businessFax*, eMail) |
((homePhone | awayPhone | pager)+, (homeFax* | awayFax*), eMail)),
Message?)>
```

On any given day, the subject of this list might be in the office, at home, or away on a business trip. Thus, each <contactInfo> element may include one of the following lists of information, with subelements in the order given:

- The date, a business phone number, zero or more business fax numbers, the e-mail address, and zero or one messages.
- The date, one or more home phone numbers or pager numbers, zero or more home fax numbers, the e-mail address, and zero or one messages.
- The date, one or more on-the-road phone numbers or pager numbers, zero or more on-the-road fax numbers, the e-mail address, and zero or one messages.

ALLOWING EMPTY TAGS

If you want to write your XML documents so they are easily translated into HTML format, you might want to include tags such as <BR> and <HR> in your XML file, with an eye toward translating them verbatim into the HTML file.

You can't really do this in XML, because every element must have a closing tag. However, you can create what are called empty tags and let an XML-to-HTML converter translate them into the proper output tags. For example, to allow the creation of <HR> tags, you would include the following line in the DTD:

```
<!ELEMENT HR EMPTY>
```

To use this tag, you could insert a line like the following into your XML file:

```
<HR/>
```

You can't include it as <HR>, because every XML tag must either have a closing tag or end with a forward slash, but an XML-to-HTML converter should convert the <HR/> to an <HR>.

Empty tags are often used to contain images; the URL of the image data is stored in one of the empty tag's attributes. For more information about attributes, see "Defining attributes" in this section.

USING CHARACTER REFERENCES

A character reference is a way of representing Unicode characters in parsed character data. The syntax for character references is as follows:

```
&#UnicodeValueOfCharacter;
```

For example, to insert the euro monetary sign before the number 500 in an <amount> element, you could do the following (character reference in bold):

```
<amount>&#x20AC;500</amount>
```

It is the job of the XML processor to substitute the appropriate Unicode characters for character entity references at output.

USING ENTITY REFERENCES

An entity reference is a bit of text that represents something else, such as a character, a string of text, an externally-stored XML file, or a binary file (such as a picture or sound file). There are five kinds of entity references:

- *Parsed internal entity* represent frequently used text strings.
- *Parsed external entity* references are used to refer to externally-stored text files containing parsable data.
- *Unparsed external entity* references are often used to refer to binary files such as pictures, spreadsheets, and sounds.
- *Internal parameter entity* references represent frequently used markup declarations within DTDs (always parsed).
- *External parameter entity* references refer to externally-stored text files containing parsable data from within DTDs (always parsed).

These entity reference types are described in detail below.

What's the difference between an entity and an entity reference? An entity reference is the shorthand you insert into an XML document to represent an entity. An entity is the content that replaces the entity reference when the XML is processed.

PARSED INTERNAL ENTITY REFERENCES

A parsed internal entity reference is basically shorthand for a string of characters that you plan to re-use often within a given XML document. The format for declaring a parsed internal entity reference in a DTD is as follows:

```
<!ENTITY entityName "replacement text">
```

For example, say you're building an XML document that contains a list of employees and some information about each of them. Each employee's record needs to contain the phrase "Years with the company:", followed by a number. Rather than entering the phrase over and over again manually, you can create a parsed internal entity reference for the phrase as part of the document's DTD, as follows:

```
<!-- Root element is <employeeListing> -->
```

```
<!DOCTYPE employeeListing [
<!-- DTD begins here -->
  <!ENTITY yrs "Years with the company:">
  <!ELEMENT employeeListing (employee*)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT IDnumber (#PCDATA)>
  <!ELEMENT yearsWithCompany (#PCDATA)>
  <!ELEMENT employee (name, IDnumber, yearsWithCompany)>
]>
```

To use the `yrs` parsed internal entity reference in the XML document, you might do the following:

```
<employee listing>
<employee>
  <name>Alexis Barnswallow</name>
  <IDnumber>H867KL671BR</IDnumber>
  <yearsWithCompany>&yrs; 12</yearsWithCompany>
</employee>
</employee listing>
```

When this `<employee>` element is processed, the XML processor will expand the parsed internal entity reference, resulting in the following XML:

```
<employee>
  <name>Alexis Barnswallow</name>
  <IDnumber>H867KL671BR</IDnumber>
  <yearsWithCompany>Years with the company: 12</yearsWithCompany>
</employee>
```

There are five predefined parsed internal entity references available in XML. Unlike all other parsed internal entity references, these are part of the XML specification and do not need to be declared.

CHARACTER	ENTITY REFERENCE
<	`<`
>	`>`
&	`&`
"	`"`
'	`'`

For example, say you need to use a greater than sign (>) in your XML document's content. As you know, the greater than sign indicates the closing of a tag in XML. To avoid confusing the XML processor, you can substitute `>` for

the greater than sign wherever it occurs. For example, to express “the whole > the sum of its parts” in an XML file, you could do the following:

```
<platitude>
  the whole &gt; the sum of its parts
</platitude>
```

There are three restrictions to using parsed internal entity references:

- You must declare a parsed internal entity reference before you use it (with the exception of the five listed above).
- You can use a parsed internal entity reference in element content and in attribute values.
- You can use a parsed internal entity reference inside another parsed internal entity reference, but only if this does not lead to a circular reference. For example, it would be illegal to use parsed internal entity reference A in parsed internal entity reference B if parsed internal entity reference B is used in parsed internal entity reference A.

PARSED EXTERNAL ENTITY REFERENCES

A parsed external entity reference lets you include content stored in an externally-located text file. Parsed external entity references should be declared in the DTD in one of the following ways:

```
<!ENTITY entityName SYSTEM "URL of file to be referenced">
<!ENTITY entityName PUBLIC "name of file to be referenced"
                           "URL of file to be referenced">
```

The first example lets you use the URL of a particular file. The second example lets you use the *name* of a resource, which may in turn point to a URL; the URL that follows is a “backup” URL, to be used only if the name cannot be resolved.

Parsed external entity references can be used to share content between XML files. For example, here’s a complete sample XML document in which the content is stored in a text file named “myfile.txt” on Quark’s Web site:

```
<?xml version="1.0" standalone="no"?>
<!-- Root element is <myRoot> -->
<!DOCTYPE myRoot [
  <!-- DTD begins here -->
  <!ELEMENT myRoot ANY>
  <!ENTITY xmlContent SYSTEM "http://www.quark.com/myfile.txt">
]>
<!-- Document begins here -->
<myRoot>
```

```
  &xmlContent;
</myRoot>
```

This is handy because it lets you also use the content in “myfile.txt” in other XML files.

If a document uses external entity references, you should set the “standalone” attribute in the XML declaration to “no.”

UNPARSED EXTERNAL ENTITY REFERENCES

What if you want to refer to a picture, spreadsheet, sound file, HTML file, or other non-XML file in an XML document? You can’t use a parsed external entity reference because the XML processor will try to parse your binary file, and that will lead to errors.

To get around this problem, you can supply a notation at the end of the external entity reference. A notation simply tells the XML processor not to parse the target file, and indicates what kind of file it is. The format for declaring a notation in a DTD is as follows:

```
<!NOTATION notationName SYSTEM "ApplicationName">
```

For example, to make a connection between JPEG files and Adobe Photoshop, you could add a notation such as this one to the DTD:

```
<!NOTATION jpeg SYSTEM "Adobe Photoshop">
```

To utilize a notation in an external entity reference declaration, use the following syntax:

```
<!ENTITY entityName SYSTEM "URL" NDATA notationName>
```

For example, to create an entity named “myPicture” that points to a URL containing a JPEG file, you could use the following tag:

```
<!ENTITY myPicture SYSTEM "http://www.quark.com/picture.jpg" NDATA jpeg>
```

You can also use the `PUBLIC` syntax with notations, specifying first a public notation name and then a backup notation URL:

```
<!ENTITY myPicture PUBLIC "-//Quark//Fictional JPEG Name"
          "http://www.quark.com/xml/picture.jpg" NDATA jpeg>
```

Unparsed entity references are not the only way to refer to external files in XML files without specifying that they must be parsed. You can also store the URL of such a file as plain element or attribute content. The first example below references the URL of a picture file as element content, and the second example references the same URL as attribute content:

```
<myPicture>http://www.quark.com.picture.jpg</myPicture>
<myPicture URL="http://www.quark.com.picture.jpg"/>
```

Whether you choose to use unparsed entities, elements, or attributes to refer to non-XML files is up to you. Any of these methods will work equally well, as long as the application that processes the XML knows that the URLs are URLs.

INTERNAL PARAMETER ENTITY REFERENCES

If you want to create an entity reference that is used only within a particular DTD, you must create a parameter entity reference. An internal parameter entity reference is very similar to a parsed internal entity reference, except it begins with a % instead of a &, both in its declaration and when you use it:

```
<!ENTITY % entityName "entity definition">
%entityName;
```

You can use internal parameter entity references in a DTD's external subset in the same way that you use parsed internal entities in an XML document. For example, here we use an internal parameter entity reference to create a shorthand way of referring to a content model that describes a person's name:

```
<!ENTITY % name (firstName, lastName) >
<!ELEMENT employerName %name;>
<!ELEMENT employeeName %name;>
<!ELEMENT customerName %name;>
```

This is useful because it makes it easy for you to change the definition of all types of names at one time. So, for example, if you decided you wanted to also store middle names for all employers, employees, and customers, you could just change the internal parameter entity declaration above to:

```
<!ENTITY % name (firstName, middleName, lastName) >
```

Note that this kind of internal parameter entity reference can be used only in a DTD's external subset.

EXTERNAL PARAMETER ENTITY REFERENCES

An external parameter entity reference is very similar to a parsed external entity reference, except it begins with a `%` instead of a `&`, both in its declaration and when you use it. For example, the following two lines (from an XML document's internal subset) first create an entity reference pointing to an external DTD called "standardHeader.dtd" and then include that external DTD in the XML file:

```
<!ENTITY % standardHeader SYSTEM "standardHeader.dtd">
%standardheader;
```

For more information about this usage, see "Using public DTDs" in this section.

Parameter entity references can be used only within a DTD.

Internal and external parameter entity references can be used together. For example, you can use internal parameter entity references in the internal subset to refer to entities that are defined in the external subset. This is useful because it lets you change the definition of an entity without having to change the internal subset of XML files that use the entity. For example, you could include the following declaration in a text file named “entitiesFile.txt”:

```
<!ENTITY % nameEntity "<!ELEMENT name (firstName, lastName)>">
```

Then, in the internal subset of XML documents, include the following:

```
<!— Include the file containing the above entity —>>
<!ENTITY % entitiesFile SYSTEM nameEntities.txt >
%entitiesFile;
<!— Now call the entity defined in the external file —>
%nameEntity;
```

This would enable you to change the definition of the `nameEntity` entity reference in any number of XML documents by changing it in the “entities-File.txt” file.

DEFINING ATTRIBUTES

In addition to containing content, elements can also have attributes (see “Understanding XML” in this chapter). There is some disagreement about the role of attributes, but for the purpose of this discussion, we’ll assume that an attribute should contain information about an element that is important to the XML processor, but is not part of the content of the XML file itself.

For example, say you’re using XML to maintain a list of books for display on a Web site. The list can be displayed in two ways: as a full list, or as a list of all the books that have been added to the list in the past 10 days. In order to make this work, the XML document needs to indicate the date on which each book is entered.

You could add a `<dateEntered>` subtag to the definition of the `<book>` tag, but the date on which a given book is entered into your system isn’t really information about the book itself, so you might choose instead to create an attribute named `dateEntered`.

The syntax for attribute declarations is as follows:

```
<!ATTLIST elementName attributeName AttributeType DefaultValue>
```

So, to give the `<book>` element a `dateEntered` attribute with a default value of `01/01/2000`, you would add the following line to the DTD:

```
<!ATTLIST book dateEntered CDATA "01/01/2000">
```

Then to use this attribute in a `<book>` element, you would use an attribute-value pair, like this:

```
<book dateEntered="11/11/1998">
 Description of book goes here
</book>
```

This attribute would give the XML processor the information necessary to display books based on their entry date.

REQUIRED, IMPLIED, AND FIXED ATTRIBUTES

Each attribute may be required, implied, or fixed. A *required attribute default* specifies that the element must contain this attribute. For example, the following attribute declaration specifies that each `<book>` element must have a `dateEntered` attribute:

```
<!ATTLIST book dateEntered CDATA #REQUIRED>
```

An *implied attribute default* indicates that the element may or may not contain this attribute, at the XML author's discretion. For example, the following attribute declaration specifies that each `<book>` may or may not contain a `dateEntered` attribute:

```
<!ATTLIST book dateEntered CDATA #IMPLIED>
```

A *fixed attribute value* indicates that the attribute must contain an exact value for each element. For example, the following attribute declaration specifies that every `<book>` must have a `dateEntered` value equal to `11/11/1998`:

```
<!ATTLIST book dateEntered CDATA #FIXED "11/11/1998">
```

In this example, the XML processor will assume that every `<book>` element has a `dateEntered` attribute set to `11/11/1998`, even if the attribute is omitted.

If an attribute declaration has a default value, but does not specify `#REQUIRED`, `#IMPLIED`, or `#FIXED`, the XML processor will assume the default value for the attribute whenever the attribute is omitted.

ATTRIBUTE TYPES

The `CDATA` keyword in our sample attribute declaration indicates that we want this attribute to contain character data. However, `CDATA` is only one option for attribute type. The full list follows.

- A `CDATA` attribute may contain character data and entity references.
- An `ENTITY` attribute must contain the name of an unparsed entity declared in the DTD. (For more information about entities, see "Using entity references" in this section.) For example, you could use an `ENTITY` attribute to contain the URL of a graphic:

```
<!-- In the DTD -->
<!ENTITY defaultCover SYSTEM "noCover.jpg" NDATA jpg>
<!ENTITY myCover SYSTEM "myBookCover.jpg" NDATA jpg>
```

```
-
<!ATTLIST book cover ENTITY defaultCover>
<!-- In the XML body -->
<book cover="myCover">
  Description of book goes here
</book>
```

- An ENTITIES attribute must contain the names of one or more of the unparsed entities declared in the DTD. The list of entities must be separated by spaces. For example:

```
<!-- In the DTD -->
<!ENTITY myCover SYSTEM "myBookCover.jpg" NDATA jpg>
<!ENTITY myAuthor SYSTEM "myBookAuthor.jpg" NDATA jpg>
-
<!ATTLIST book graphics ENTITIES #IMPLIED>
<!-- In the XML body -->
<book graphics="myCover myAuthor">
  Description of book goes here
</book>
```

- An *enumerated list attribute* may contain a single name from a list of names enclosed in parentheses. (“Name” here means the values in the list must all follow the same naming conventions as XML elements.)

  To use an enumerated list attribute, add the list of names to the attribute declaration in place of a keyword. For example, the following line specifies that the saleStatus attribute of the <book> element may contain *only* On_Sale or Regular_Price:

```
<!ATTLIST book saleStatus (On_Sale | Regular_Price) #IMPLIED>
```

- An *ID attribute* specifies that each element must have a unique value for that attribute. For example:

```
<!-- In the DTD -->
<!ATTLIST book bookNumber ID #REQUIRED>
<!-- In the XML body -->
<book bookNumber="B068157">
  Description of book goes here
</book>
```

An ID attribute must have a declared default of #IMPLIED or #REQUIRED. No element may have more than one ID attribute.

- An IDREF attribute must “point to” another element in the XML document that has an ID attribute value. For example:

```
<!-- In the DTD -->
<!ATTLIST book bookNumber ID #REQUIRED>
<!ATTLIST book cataloguedIn IDREF #REQUIRED>
<!-- In the XML body -->
<book bookNumber="B000321">
 A catalog of books about herbs.
</book>
<book bookNumber="B000123" cataloguedIn="B000321">
 A book about herbs.
</book>
```

- An IDREFS attribute’s value consists of a series of IDREF attributes, separated by spaces.
- A NMTOKEN attribute’s value may contain a variety of characters, including letters, numbers, underscores, periods, and so forth, but may not contain spaces. For example, the following example would be illegal, because the phrase “My Local Name” contains spaces:

```
<!-- In the DTD -->
<!ATTLIST book localName NMTOKEN #IMPLIED>
<!-- In the XML body -->
<book localname="My Local Name">
 Description of book goes here
</book>
```

- A NMTOKENS attribute is the same as a NMTOKEN attribute, except it specifies that the attribute value may be a list of NMTOKENs separated by spaces.

```
<!-- In the DTD -->
<!ATTLIST XMLbook subjectType NMTOKENS #IMPLIED (xml xsl other)>
<!-- In the XML body -->
<XMLbook subjectType="xml">
 Description of book goes here
</XMLbook>
```

- A NOTATION attribute must contain one or more XML notation names from the DTD. This lets you create notations that specify image-viewing and movie-playing applications, and then use an element attribute to make sure the appropriate application is used for a given piece of content. For example:

```
<!-- In the DTD -->
```

```
<!NOTATION jpg SYSTEM "PictureViewer">
<!NOTATION mov SYSTEM "MoviePlayer">
<!ELEMENT multimediaElement EMPTY>
  <!ATTLIST multimediaElement file ENTITY #REQUIRED>
  <!ATTLIST multimediaElement type NOTATION #REQUIRED>
<!-- In the XML body -->
<multimediaElement file="MyImage.jpg" type="jpg" />
<multimediaElement file="MyMovie.mov" type="mov" />
```

This information lets the application that processes the XML know that it can use the PictureViewer application to open the image file and the MoviePlayer application to open the movie file.

No element may have more than one `NOTATION` attribute.

- An enumerated NOTATION attribute must contain a list of one or more XML notation names from the DTD, in parentheses. You might, for example, create notations that specify multiple image-viewing and movie-playing applications, then use an element attribute to make sure the appropriate set of applications is used for a given piece of content. Here's how it might look:

```
<!-- In the DTD -->
<!NOTATION picViewer SYSTEM "PictureViewer">
<!NOTATION photoshop SYSTEM "Photoshop.exe">
<!NOTATION movPlyrMac SYSTEM "MoviePlayer">
<!NOTATION movPlyrWin SYSTEM "Movieplayer.exe">
<!ELEMENT image EMPTY>
  <!ATTLIST image file ENTITY #REQUIRED>
  <!ATTLIST image imageApp NOTATION (picViewer | photoshop)
#REQUIRED>
<!ELEMENT movie EMPTY>
  <!ATTLIST movie file ENTITY #REQUIRED>
  <!ATTLIST movie movieApp NOTATION (movPlyrMac | movPlyrWin)
          #REQUIRED>
<!-- In the XML body -->
<image file="MyImage.jpg" imageApp="picViewer"/>
<movie file="MyMovie.mov" movieApp="movPlyrMac"/>
```

Here, instead of creating one element for both images and movies, we create two separate elements, <image> and <movie>. For each of these elements, the DTD specifies two applications that might be used to view the file. The determination of which application to use is made in each individual <element> tag in the XML body.

THE XML:LANG ATTRIBUTE

The xml:lang attribute lets you specify which language is used in an element. This attribute should contain one of the following:

- A two-letter language code defined by ISO 639, optionally followed by a hyphen and a subtype (typically a country code)
- An IANA-registered language number, prefixed with “i-” or “I-”
- A user-defined language code, prefixed with “x-” or “X-”

Note that these attributes are not predefined — you must declare them before you use them.

To indicate the language you want, simply assign that language's code. For example, the following DTD specifies an xml:lang element, and the element in the XML body specifies the English language using ISO 639:

```
<!-- In the DTD -->
<!ELEMENT Paragraph (#PCDATA)>
<!ATTLIST Paragraph xml:lang NMTOKEN #REQUIRED>
<!-- In the XML body -->
<Paragraph xml:lang="en">
Paragraph data goes here.
</Paragraph>
```

You can specify language subtypes by adding a hyphenated extension to the language name. For example, the following element specifies International English (used in the United Kingdom), as opposed to U.S. English:

```
<!-- In the XML body -->
<Paragraph xml:lang="en-GB">
Paragraph data goes here.
</Paragraph>
```

THE XML:SPACE ATTRIBUTE

The xml:space attribute lets you indicate to the application that processed the XML that it should leave all white space for an element and its children as is (unless one of the element's children resets the tag). For example, the following DTD specifies an xml:space attribute, and the element in the XML body sets that attribute to preserve for that element and its children:

```
<!-- In the DTD -->
<!ELEMENT Paragraph (#PCDATA)>
<!ATTLIST Paragraph xml:space (default | preserve) "default">
<!-- In the XML body -->
<Paragraph xml:space="preserve">
Paragraph data goes here.
   All
  white
   space
 preserved.
</Paragraph>
```

IGNORE AND INCLUDE

You can use the <![IGNORE[]]> tag to tell the XML parser to ignore a stretch of text in an external DTD. Take for example the following:

```
<-- This element declaration is parsed as usual: -->
<!ELEMENT studentStep (#PCDATA)>
<![IGNORE[
  <-- This element declaration is ignored by the XML parser: -->
  <!ELEMENT instructorNote (#PCDATA)>
]]>
```

You can tell the XML parser to parse the text within the tags by simply changing the `IGNORE` to an `INCLUDE`, as follows:

```
<-- This element declaration is parsed as usual: -->
<!ELEMENT studentStep (#PCDATA)>
<![INCLUDE[
  <-- This element declaration is now also parsed as usual: -->
  <!ELEMENT instructorNote (#PCDATA)>
]]>
```

USING PUBLIC DTDS

As we mentioned earlier, you can refer to an external DTD in an XML document's `DOCTYPE` declaration, like this:

```
<?xml version="1.0" standalone="no">
<!-- DTD begins here -->
<!DOCTYPE myDocument SYSTEM "mydocument.dtd">
<!-- Document begins here -->
<myDocument>
…
```

If you are using a DTD that has been approved by a body such as the International Standards Organization (ISO), you can use a `PUBLIC` entity reference that specifies the name of a publically available copy of the DTD. When you do this, you must also supply the URL of a `SYSTEM` DTD file, so there's something to fall back on if the `PUBLIC` copy of the DTD is unavailable.

```
<?xml version="1.0" standalone="no">
<!-- DTD begins here -->
<!-- First URL below is PUBLIC DTD, second is backup SYSTEM DTD -->
<!DOCTYPE stdDoc PUBLIC "-//Quark//DTD stdDoc 1.0//EN"
                        "http://www.quark.com/xml/stdDoc.dtd">
<!-- Document begins here -->
<stdDoc>
…
```

COMBINING DTDS TO CREATE COMPOSITE DTDS

Sometimes you may create separate DTDs to define different parts of a document. For example, your organization may use one DTD for all of its XML files' header and footer information, but different DTDs for the body of documents produced in different parts of the company. You can accommodate such situations by creating a single new DTD that includes the various DTDs you need and specifies an order for their root elements, like this:

```
<!ENTITY % standardHeader SYSTEM "standardHeader.dtd">
<!ENTITY % QARept SYSTEM "QARept.dtd">
<!ENTITY % standardFooter SYSTEM "standardFooter.dtd">
%standardHeader;
%QARept;
%standardFooter;
<!-- Root element is <QAReptDoc> -->
<!ELEMENT QAReptDoc (standardHeader, QARept, standardFooter)>
```

For documents created with this DTD, <QAReptDoc> would be the root element, and <standardHeader>, <QARept>, and <standardFooter> would be its immediate subelements. A document that uses this DTD might look something like this:

```
<?xml version="1.0" standalone="no"?>
<!— The following line specifies a root element (<QAReptDoc>) and points to the
    URL of an external DTD file named
    "QAReptDoc.dtd" —>
<!DOCTYPE QAReptDoc SYSTEM  QAReptDoc.dtd">
<!-- Document begins here -->
<QAReptDoc>
  <standardHeader>
    <!— Standard header content goes here -->
  </standardHeader>
  <QARept>
    <!-- QA report content goes here -->
  </QARept>
  <standardFooter>
    <!— Standard footer content goes here -->
  </standardFooter>
</QAReptDoc>
```

MAKING LOCAL MODIFICATIONS TO IMPORTED DTDS

Some workflows may involve DTDs that are almost identical for a group of users, but which require small adjustments to work in any particular department or group. This is easy to arrange; just include the DTD in the DOCTYPE declaration, then add any necessary markup declarations to the internal subset. You cannot redefine an element that is already defined in the external DTD, but you can redefine entities and default values for attributes.

VALIDATING AN XML FILE AGAINST A DTD

If you're writing your XML documents with a word processor, you can read through the corresponding DTD and make sure that you follow the rules. But you won't really know for sure whether you did until you validate the XML document against the DTD using a program called a validating parser. The validating parser reads the DTD and then checks your XML file to make sure it adheres to the DTD's rules. A good validating parser should also tell you what problems it finds (if any).

Remember that if you want to check an XML document for adherence to a particular DTD, you need a *validating* XML parser, not just a plain XML parser. There are many XML parsers that will tell you if an XML file is well-formed, but considerably fewer that will tell you if an XML file is valid.

For a quick reference to DTD features and conventions, see Appendix B, "DTD Quick Reference," in Chapter 7, "Appendices."

INDUSTRY-STANDARD DTDS

Should you develop a new DTD, custom-designed to fit the needs of your organization? Or should you use an industry-standard DTD that will save you development time and let you easily exchange information with other organizations in your industry?

There are advantages to both approaches. If you create your own DTD from scratch, you have total control over the structure of that DTD and the process of updating it. However, you're also looking at a significant investment of time and effort, and you must be very careful to consider the needs of everyone who will be using that DTD. If you use an industry-standard DTD, you don't have to go through the DTD development process, but you have to follow the DTD's conventions and adhere to the structure it defines.

PROS AND CONS OF USING INDUSTRY-STANDARD DTDS

If you plan to exchange information with other organizations, an industry-standard DTD might be a good idea. Using an industry-standard DTD can help ensure that information exchange goes smoothly, and that the information you tag can be re-used in other contexts. Indeed, this is one of the reasons XML was

developed: to help standardize the formats in which information is stored and exchanged.

Using an industry-standard DTD can present its own challenges, because two organizations may have very different needs, even if the data they work with is essentially the same. Industry-standard DTDs can be modified for use within an organization, but that partially defeats their purpose, which is to ensure that information is stored in a consistent format between organizations.

CAN I USE AN INDUSTRY-STANDARD DTD?

Whether you can use an industry-standard DTD depends on a number of factors.

DOES AN INDUSTRY-STANDARD DTD EXIST FOR YOUR INDUSTRY?

To find out the answer to this question, you can look for industry-standard DTDs on the World Wide Web. Two good places to look are www.schema.net and www.xml.org.

IF AN INDUSTRY-STANDARD DTD EXISTS, DOES IT MEET YOUR NEEDS?

If you locate an industry-standard DTD that seems like it might work for your organization, *review it carefully before you begin using it*. The use of an inappropriate DTD can lead to serious problems. For example, an inappropriate DTD can:

- Force authors to use unfamiliar and inappropriate structures.
- Require authors to shoehorn data into inappropriate elements.
- Restrict authors' ability to index and search information.
- Provide too much leeway, leading to improper markup.
- Make conversion to other formats more difficult.
- Force applications that process data to be overly complex.
- Incur huge expenses if the DTD ultimately turns out to be unusable, forcing you to convert all of your data to a different format.

Think carefully about this question; if the DTD you choose does not meet your needs, the cumulative effect of any shortcomings will probably grow with time.

IF NO INDUSTRY-STANDARD DTD EXISTS FOR YOUR INDUSTRY, IS ONE IN DEVELOPMENT?

If you can't find an industry-standard DTD that fits your organizational needs, you might want to find out if anyone else in your industry is developing one. If so, your organization may have a chance to bring its particular expertise to bear in the development of the DTD. Participation in the development of an industry-standard DTD can help you avoid problems that can result from adopting a DTD developed by someone who doesn't understand your needs.

EXTENDING INDUSTRY-STANDARD DTDS

Some organizations choose to use an industry-standard DTD, but modify that DTD to make it suit their particular needs. For example, to make the ISO-standard “book” SGML DTD work for them, the University of California Press made a series of adjustments to it, adding elements that let them store information such as chapter subtitles and chapter-specific bylines. The ISO (International Standards Organization) provides guidelines for modifying its DTDs, so even if you make such modifications, your new DTD is still somewhat standardized.

What if you need to exchange data with other organizations that use the original, unmodified DTD? Some organizations choose to create utilities that can convert documents that adhere to their modified DTD into documents that adhere to the original form of the DTD. This kind of solution gives you many of the advantages of having a customized DTD, yet still allows you to exchange data with other organizations in the industry.

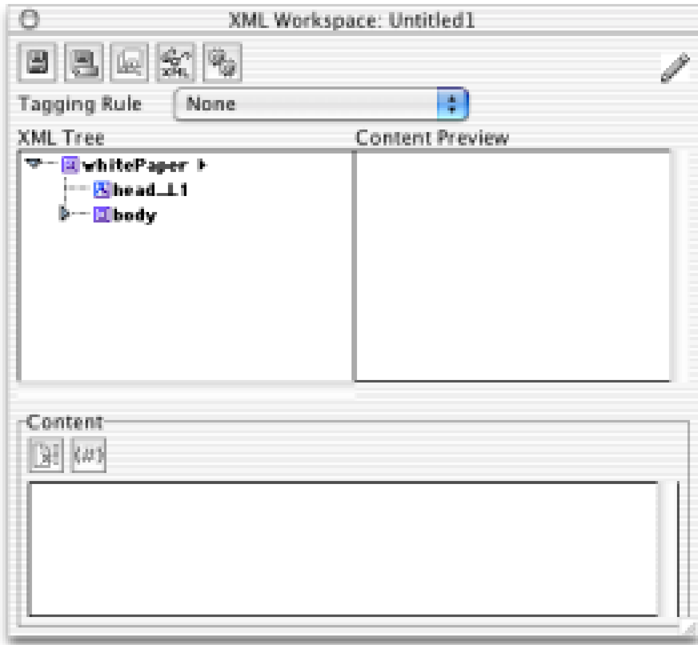
Chapter 3: XML Workspace Palette

Palettes are movable interface windows that provide you with the commands and features you need to complete specific tasks. Palettes are especially convenient because they can be placed anywhere on your screen, allowing you to customize your workspace. Click and drag the bar at the top of a palette to reposition it. You can resize most palettes by dragging the resize box in the lower right corner of the palette.

XML WORKSPACE PALETTE

Avenue.quark provides you with the ***XML Workspace*** *palette, which displays an XML document as an easy-to-understand hierarchical "tree." You can use the XML Workspace palette to create, edit, view, and save XML documents from within QuarkXPress.*

The ***XML Workspace*** *palette (**File** & **New** & **XML**) lets you view and edit the content of an XML document.*



XML Workspace palette

SAVE (BUTTON)

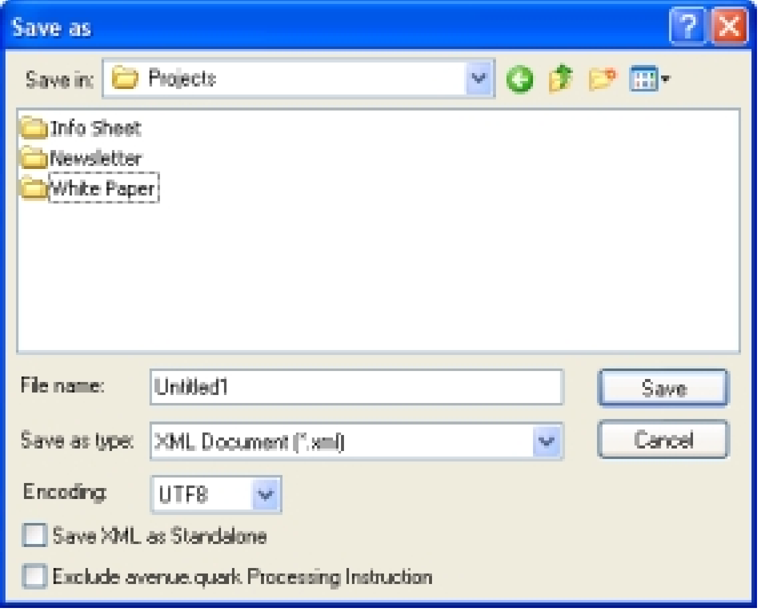
Clicking the **Save** button saves the active XML document. If the active XML document has not yet been saved, clicking the **Save** button displays the **Save As** dialog box, which lets you specify the name, location, type, and character encoding for the document.

SAVE AS (BUTTON)

The **Save As** button displays the **Save As** dialog box, which lets you specify the name, location, type, and character encoding for the active XML document.

SAVE AS (DIALOG BOX)

The **Save As** dialog box lets you specify the name, location, type, and character encoding for the active XML document.



Save As dialog box

The **Save current XML as** field lets you enter a name for the active XML document.

The **Save as Type** pop-up menu offers two options:

- Choosing **XML Document** specifies that the active layout should be saved as an XML document.
- Choosing **avenue.quark Template** specifies that the active XML document should be saved as a template that can be used as a basis for new XML documents.

The **Encoding** pop-up menu lets you choose a character encoding method for the XML document. The available options are **UTF-8**, and **UTF-16 (Unicode). UTF-8** is selected by default.

Check **Save XML as Standalone** if you want to include the entire DTD in the XML file. Uncheck **Save XML as Standalone** to include only the name of the external DTD file.

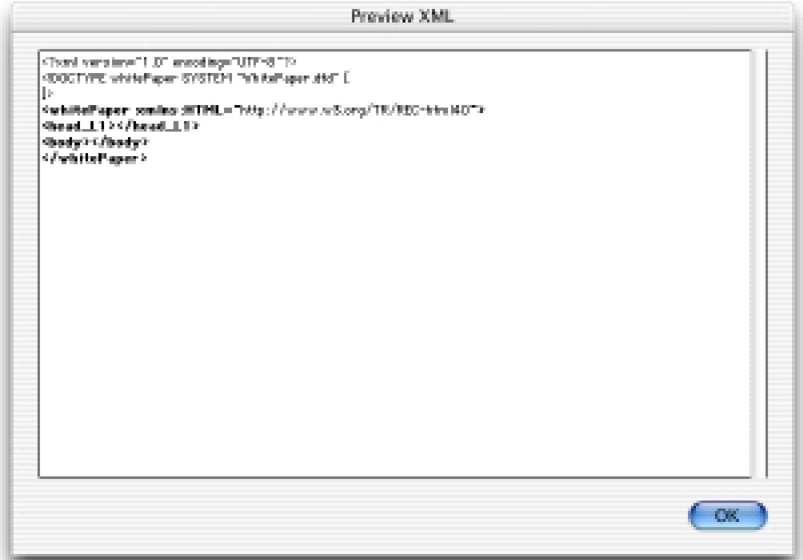
Check **Exclude avenue.quark Processing Instructions** if you do not want avenue.quark to create processing instructions. Uncheck **Exclude avenue.quark Processing Instructions** to let avenue.quark create processing instructions. Processing instructions are customized commands inserted in the XML document that tell other applications how to process particular types of information.

REVERT TO SAVED (BUTTON)

The **Revert to Saved** button lets you discard changes and restore the active XML document to the most recently saved version.

PREVIEW XML (BUTTON)

The **Preview XML** button displays the **Preview XML** dialog box, which lets you view the active XML document as XML text. You can copy and paste from this window, but you cannot edit the text.



Preview XML dialog box

Although upper-ASCII characters (characters above ASCII 127) display unaltered in the **Preview XML** dialog box, such characters are converted to the appropriate codes at export, depending on the encoding method you choose in the **Save As** dialog box.

SYNCHRONIZE CONTENT (BUTTON)

The **Synchronize Content** button synchronizes the content of elements in the active XML document with the corresponding content in the active QuarkXPress layout or layouts.

PENCIL (ICON)

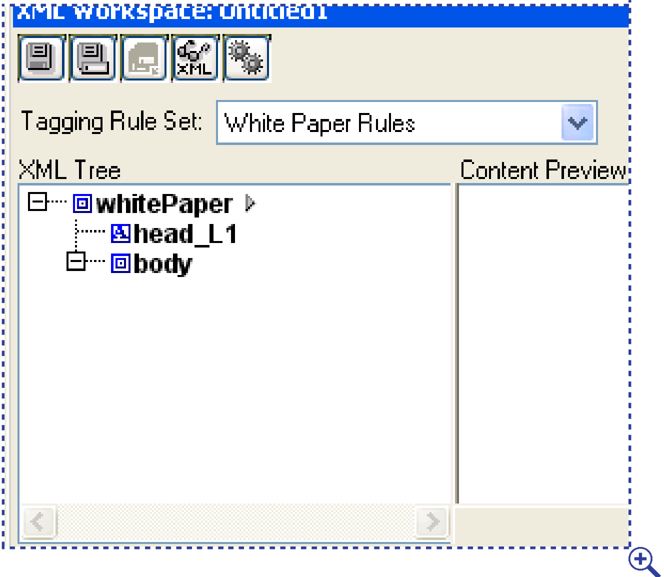
A pencil icon displays in the upper-right corner of the **XML Workspace** palette, indicating the active XML document.

TAGGING RULE SET (POP-UP MENU)

The **Tagging Rule Set** pop-up menu lets you choose a tagging rule set for use with the active XML document. This pop-up menu displays only those tagging rule sets that are associated with the active XML document.

XML TREE (LIST)

The **XML Tree** list displays a hierarchical tree representation of the XML document. You can display and hide the contents of container elements and attributes by clicking the and > disclosure triangles (Mac OS) or the and disclosure boxes (Windows). You can scroll through the **XML Tree** list using the horizontal and vertical bars or using the arrow keys.



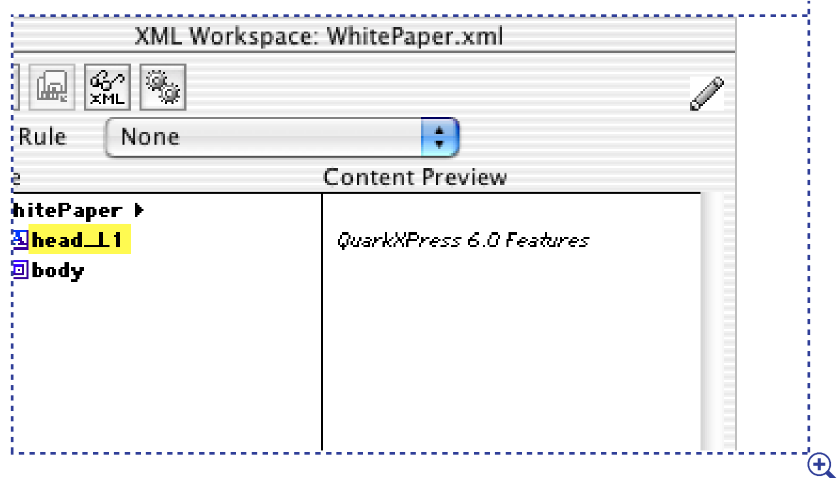
XML Tree list

Different types of items in the **XML Tree** list have different icons:

- ANY elements
- Attributes
- Comments
- Container elements
- EMPTY elements
- Mixed content elements
- PCDATA elements
- Processing instructions

CONTENT PREVIEW (LIST)

The **Content Preview** list displays the content of the element, attribute, or comment selected in the left side of the list. Content that is too large to display fully is indicated with ellipsis points (...).



Content Preview list

An attribute's content is handled differently according to its type (which is specified in the DTD). The content of different attribute types displays as follows:

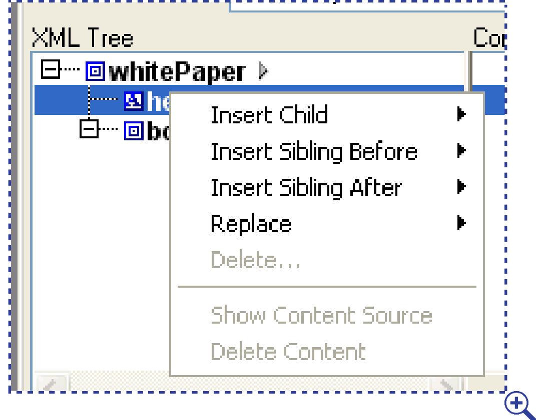
- If an attribute has a default value, the default value displays.
- If an attribute has a fixed value, that value displays.
- If an attribute is required and does not have a default or fixed value, avenue.quark will insert an underline (_) when the XML document is saved; this keeps the XML file valid. You can change this value later by opening the XML file in a text editor.
- Implied (optional) attributes are editable and display their default values in the **Content** area. If no default value is specified, nothing displays. If you do not change the implied attribute's value, the implied attribute is not included in the XML file when the XML document is saved.
- Fixed attributes display with a lock icon y.

If an element, comment, PCDATA block, or processing instruction is linked to content in a QuarkXPress layout, and that content has changed since the last time the XML document was closed, the right side of the **XML Tree** list displays a icon.

Both the right side of the **XML Tree** list and the **Content** field display the content of the selected element, comment, or processing instruction. You can edit the content in the **Content** field.

XML TREE (CONTEXT MENU)

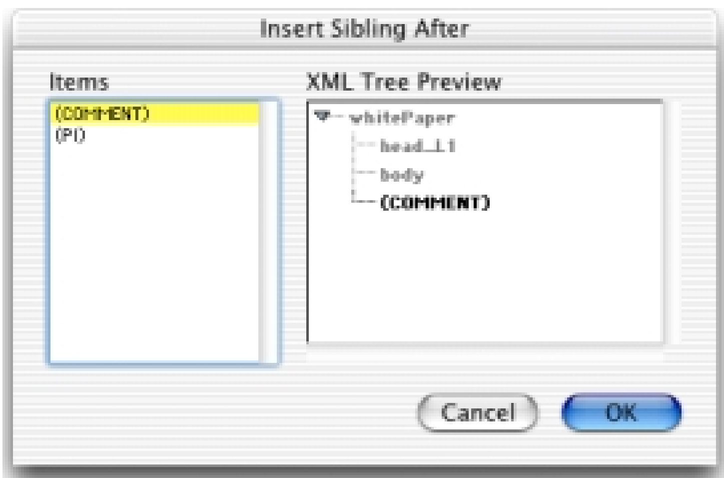
Control+clicking (Mac OS) or right-clicking (Windows) an element displays a context menu with the following options (note that not all the following options are available for all elements):



XML Tree context menu

- Choosing **Insert Child** lets you insert an element, comment, processing instruction, or PCDATA block as an immediate child of the selected element.
- Choosing **Insert Sibling Before** lets you insert an element, comment, processing instruction, or PCDATA block before the selected item.
- Choosing **Insert Sibling After** lets you insert an element, comment, processing instruction, or PCDATA block after the selected item.
- Choosing **Replace** lets you replace the selected element with a different element.
- Choosing **Delete** displays the **Delete** dialog box, which lets you delete the selected item. In this dialog box, any required elements that will also be deleted display in red struck-through text. This action cannot be undone.
- Choosing **Show Content Source** scrolls to and highlights the source of the selected XML element (if the content originated in the active QuarkXPress layout).
- Choosing **Delete Content** lets you delete the content of the selected item. This option is available when the selected item has content or after a link is broken. Note that choosing this item does not delete content from the active QuarkXPress layout — only from the selected element in the active XML document.

When you choose **Insert Child**, **Insert Sibling Before**, **Insert Sibling After**, or **Replace**, a submenu displays, containing a list of the items that may legally be inserted or substituted. This submenu also contains an **Insert with Preview** menu item, which displays a preview dialog box.



A preview dialog box.

- The **Items** list displays a list of items that may be inserted or substituted.
- The **XML Tree Preview** list shows the selected item and any of its mandatory children (in black), in the context of the XML tree (in gray); you can think of it as a preview of how the document will look after the change.

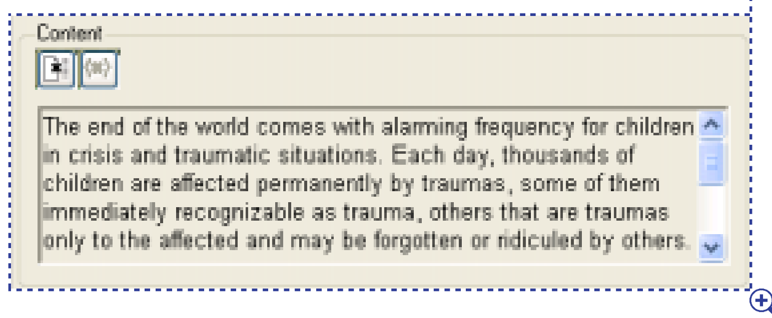
Click **OK** to complete the insertion or replacement, or **Cancel** to stop it.

Some elements have mandatory children. If you insert such an element, its mandatory children must also be inserted. Clicking an element's > disclosure triangle (Mac OS) or the ⊞ disclosure box (Windows) displays any child elements that must also be inserted along with that element. Clicking this icon does *not* display any optional children an element might have.

If an inserted element requires one of a list of non-optional elements (for example, (a | b | c)), avenue.quark uses the first element in the list (here, a). If the element requires one of a list of elements, and one or more of those elements is optional (for example, (a | b | c?)), avenue.quark leaves the element empty.

CONTENT (AREA)

The **Content** area lets you add to, edit, and delete the content of the element, attribute, or comment selected in the **XML Tree** list.



Content area

Both the right side of the **XML Tree** list and the **Content** field display the content of the selected element, attribute, or comment.

BREAK DYNAMIC LINK (BUTTON)

If you drag content from a QuarkXPress layout to an attribute or element, a link is formed between that content and the element in the XML document. This means you can edit the content of the XML document only by editing the content in the QuarkXPress layout.

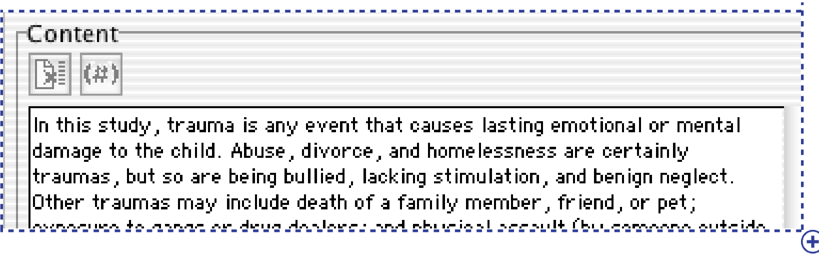
If you want to sever the link between an element in the active XML file and the QuarkXPress layout from which it came, select the element name in the **XML Tree** list and then click the **Break Dynamic Link** button .

GENERATE ID (BUTTON)

For ID attributes, the **Generate ID** button is available in the **Content** area. This button replaces the current content of the selected attribute with an automatically generated value that is unique within the active XML file.

CONTENT (FIELD)

The **Content** field lets you edit the content of elements and attributes that are not linked to QuarkXPress content. The **Content** field supports cutting, copying, and pasting of most types of attributes.

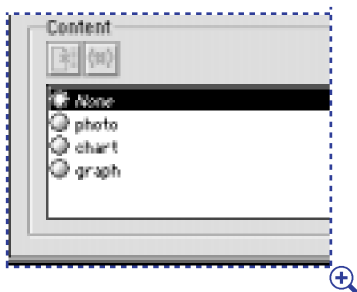


Content field

The **Content** field works differently for some types of attributes, as indicated below.

ENUMERATED ATTRIBUTES

An enumerated attribute may contain one of a series of values. For enumerated attributes, the **Content** field displays a list of valid values. You can select only one option at a time. If no default value is specified in the DTD, the top option is initially selected.



Content field for enumerated attribute

ID ATTRIBUTES

An ID attribute may contain only a value that meets these criteria:

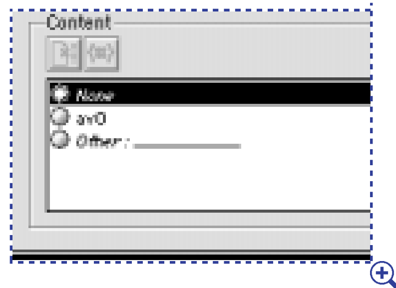
- The value must begin with a letter, underscore, or colon, with remaining characters consisting of letters, digits, underscores, hyphens, and colons, but no spaces.
- The value must be unique within the XML document.

For ID attributes, the **Generate ID** button is available in the **Content** area. This button replaces the current contents of the selected attribute with an automatically-generated value that is unique within the active XML file.

IDREF ATTRIBUTES

An IDREF attribute may contain only an ID attribute value that is used either elsewhere in the active XML document or in an XML document referred to by the active XML document.

For IDREF attributes, the **Content** field displays a list of the ID attributes used in the active XML document, as well as the **Other** option. Clicking **Other** displays a dialog box that lets you enter an IDREF value. You can select only one IDREF value at a time. By default, the **None** option is selected.



Content field for an IDREF attribute

IDREFS ATTRIBUTES

An attribute may contain one or more ID attribute values that are used either in the active XML document or in an XML document referred to by the active XML document.

For IDREFS attributes, the **Content** field displays a list of the ID attributes used in the active XML document, as well as the **Other** option. Clicking **Other** displays a dialog box that lets you enter an IDREF value. You can enter as many IDREF values as you like, separated by spaces.

NMTOKEN ATTRIBUTES

A NMTOKEN attribute may contain only a value that begins with a letter, underscore, or colon, with remaining characters consisting of letters, digits, underscores, hyphens, and colons, but no spaces.

For NMTOKEN attributes, the **Content** field will accept only one NMTOKEN value.

NMTOKENS ATTRIBUTES

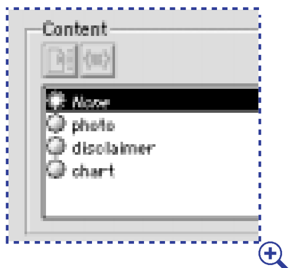
A NMTOKENS attribute may contain a series of values that begin with a letter, underscore, or colon, with remaining characters consisting of letters, digits, underscores, hyphens, and colons, but no spaces.

For NMTOKENS attributes, the **Content** field will accept a series of NMTOKEN values, separated by single spaces.

ENTITY ATTRIBUTES

An ENTITY attribute may contain only the name of an entity defined within the active XML document. For ENTITY attributes, the **Content** field displays a list of the entities defined in the active XML document. You can select only one entity name at a time.

If there is a default entity value, it is selected. If not, the first entity value is selected by default.



Content field for an ENTITY attribute

ENTITIES ATTRIBUTES

An ENTITIES attribute may contain only a list of entities defined within the active XML document. For ENTITIES attributes, the **Content** field displays a list of the entities defined in the active XML document. You can select as many of these entity values as you like.

Chapter 4: Menus and Dialog Boxes

Menus group an application's primary functions and make them readily available, while dialog boxes offer easy access to a variety of controls. Menus and dialog boxes let you "feel" your way through an application and learn it by intuition.

The menus and dialog boxes in the avenue.quark interface make it easy for you to tag the content of QuarkXPress layouts, save that content in XML format, and identify the tagged content in the QuarkXPress layout it came from.

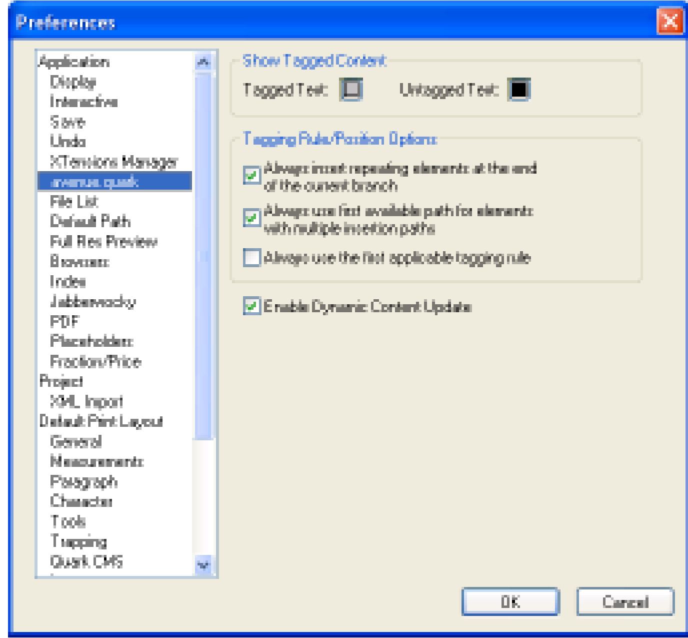
PREFERENCE SETTINGS

The **QuarkXPress** menu (Mac OS) or **Edit** menu (Windows) lets you display the **Preferences** dialog box, where you can specify avenue.quark preferences.

AVENUE.QUARK PREFERENCES (PANE)

QuarkXPress & Preferences & avenue.quark pane (Mac OS),
Edit & Preferences & avenue.quark pane (Windows)

The **avenue.quark** pane lets you specify how tagged content displays on-screen, control the color of marker text, and turn dynamic content updating on and off.



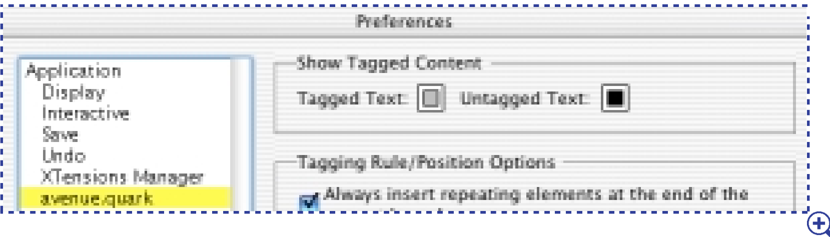
avenue.quark pane of the Preferences dialog box

SHOW TAGGED CONTENT (AREA)

QuarkXPress& Preferences & avenue.quark pane (Mac OS),

Edit & Preferences & avenue.quark pane (Windows)

The **Show Tagged Content** area lets you specify the colors used to display tagged and untagged content when **Show Tagged Content** is selected in the **Utilities** menu. To change the display color for tagged content, click the **Tagged Text** button. To change the display color for untagged content, click the **Untagged Text** button.



Show Tagged Content area

ENABLE DYNAMIC CONTENT UPDATE (CHECK BOX)

QuarkXPress & Preferences & avenue.quark pane (Mac OS),
Edit & Preferences & avenue.quark pane (Windows)

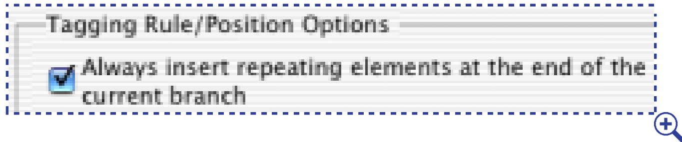
The **Enable Dynamic Content Update** button lets you specify whether the content of elements in active XML documents should be continuously updated to reflect the content of the QuarkXPress items they're linked to. You might want to uncheck this button if QuarkXPress seems to be running very slowly when editing large XML documents; when this box is unchecked, you can manually update the content by clicking the **Synchronize Content** button in the **XML Workspace** palette.

Enable Dynamic Content Update check box

ALWAYS INSERT REPEATING ELEMENTS AT THE END OF THE CURRENT BRANCH (CHECK BOX)

QuarkXPress & Preferences & avenue.quark pane (Mac OS),
Edit & Preferences & avenue.quark pane (Windows)

The **Always insert repeating elements at the end of the current branch** check box controls the placement of new repeating elements (elements marked with a + or * in the DTD). When this box is checked, avenue.quark always puts new repeating elements at the end of the active branch. When this check box is unchecked, avenue.quark displays the **Choose Rule/Position** dialog box and lets you manually choose the position of a new repeating element.



Always insert repeating elements at the end of the current branch check box

ALWAYS USE FIRST AVAILABLE PATH FOR ELEMENTS WITH MULTIPLE INSERTION POINTS (CHECK BOX)

QuarkXPress & Preferences & avenue.quark pane (Mac OS),
Edit & Preferences & avenue.quark pane (Windows)

The **Always use first available path for elements with multiple insertion paths** check box controls the placement of new elements that could be inserted in a number of places according to the DTD. For example, say a tagging rule calls for the creation of a <paragraph> element. If the DTD states that a new <paragraph> element may be created either at the end of the current branch or

as a child of a new <sidebar> element, which kind of <paragraph> element should avenue.quark generate? If this check box is checked, avenue.quark creates the first <paragraph> element it finds in the DTD tree (a <paragraph> element at the root level of the current branch). If this check box is unchecked, avenue.quark displays the **Choose Rule/Position** dialog box.

Always use first available path for elements with multiple insertion points check box

ALWAYS USE THE FIRST APPLICABLE TAGGING RULE (CHECK BOX)

QuarkXPress& Preferences & avenue.quark pane (Mac OS),
Edit & Preferences & avenue.quark pane (Windows)

The **Always use the first applicable tagging rule** check box applies to tagging rule conflicts. When this box is checked, avenue.quark always chooses the first of a series of applicable rules in tagging rule conflicts. When this box is unchecked, avenue.quark displays the **Choose Rule/Position** dialog box and lets you manually choose the element type to be applied to the selected text or click **Choose Automatically.**

Always use the first applicable tagging rule check box

PLACEHOLDER PREFERENCES (PANE)

QuarkXPress& Preferences & Placeholders (Mac OS), Edit & Preferences & Placeholders pane (Windows)

The **Placeholders** pane lets you set preferences for tag brackets. Use the **Placeholder Descriptor** area to specify the tag bracket color settings.

PLACEHOLDER DESCRIPTOR (AREA)

QuarkXPress& Preferences & Placeholders pane (Mac OS),
Edit & Preferences & Placeholders pane (Windows)

The **Placeholder Descriptor** area specifies the color and shade of tag brackets.

Placeholder Descriptor area

COLOR (BUTTON)

QuarkXPress & Preferences & Placeholders pane (Mac OS),
Edit & Preferences & Placeholders pane (Windows)

The **Color** button lets you specify the color of the brackets that surround tagged text when you choose **View** & **Show Invisibles**. To change the color of the brackets, click the button.

SHADE (POP-UP MENU AND FIELD)

QuarkXPress & Preferences & Placeholders pane & Placeholder Descriptor area (Mac OS), Edit & Preferences & Placeholders pane & Placeholder Descriptor area (Windows)

The **Shade** pop-up menu and field lets you specify the percentage of color that is applied to the brackets that surround tagged text when you choose **View** & **Show Invisibles**. To change the shade of the brackets, enter a new percentage in the field, or choose a percentage from the pop-up menu.

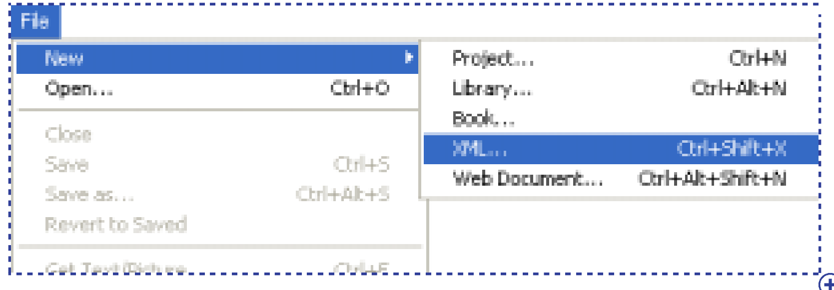
FILE MENU

*When avenue.quark is installed, the QuarkXPress **File** menu lets you create a new XML document.*

XML (COMMAND)

File & New

The **XML** command (C+Shift+X on Mac OS, Ctrl+Shift+X on Windows) displays the **New XML** dialog box, which lets you select a DTD, root element, and tagging rule set for a new XML document.

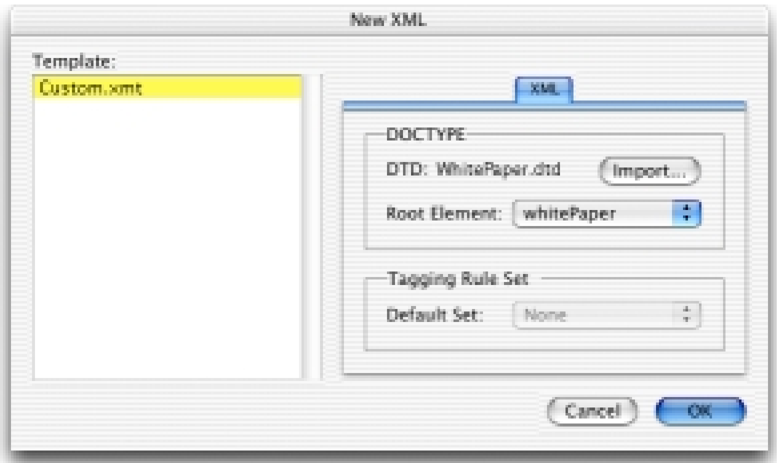


XML command

NEW XML (DIALOG BOX)

File & New & XML

The **New XML** dialog box lets you specify a template, DOCTYPE, and default tagging set for a new XML document.

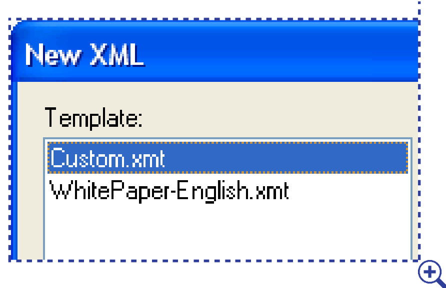


New XML dialog box

TEMPLATE (LIST)

File & New & XML

The **Template** list lets you select the XML template you would like the new XML document to be based on. The list menu displays the names of all XML templates in the "Templates" folder in the QuarkXPress folder. To create XML templates, see "Working with XML Templates" in Chapter 6, "Tagging Content."



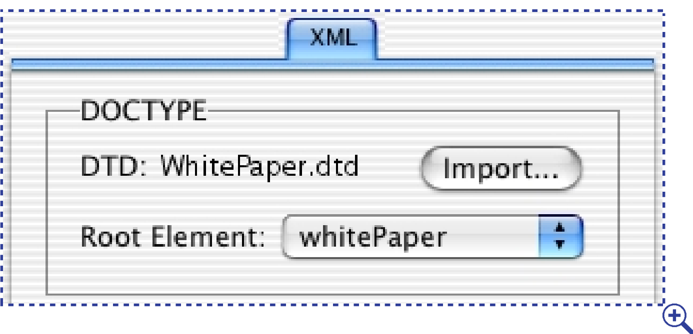
Template list

To create a new XML document that is not based on a template, select **Custom** in the **Template** list.

DOCTYPE (AREA)

File & New & XML

The **DOCTYPE** area lets you select a DTD and root element for a new XML document. If you select **Custom** in the **Template** list, you can click **Import** to import a DTD and then choose a root element from the **Root Element** pop-up menu. If you select a template in the **Template** list, the template's DTD and root element display, but cannot be changed.



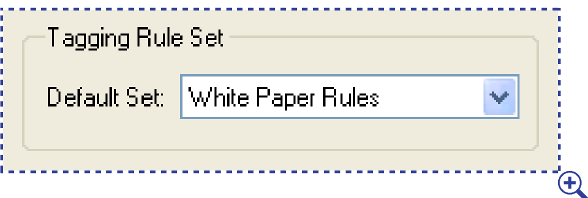
DOCTYPE area

- Clicking the **Import** button displays a dialog box that lets you import a new DTD.
- The **Root Element** pop-up menu lets you choose a root element from the list of element types in the selected DTD. The root element you choose determines the configuration of the XML tree displayed in the **XML Workspace** palette.

TAGGING RULE SET (AREA)

File & New & XML

The **Tagging Rule Set** area lets you choose a default tagging rule set for the new XML document.



Tagging Rule Set area

The **Default Set** pop-up menu displays a list of tagging rule sets associated with the root element/DTD pair selected in the **DOCTYPE** area, or **None** if no tagging rule sets are associated with that pair.

The tagging rule set chosen in the **Default Set** pop-up menu is the default selection displayed in the **Tagging Rule Set** pop-up menu in the **XML Workspace** palette. The tagging rule set can be changed at any time in the **XML Workspace** palette.

EDIT MENU

*The **Edit** menu lets you create tagging rule sets and categories.*

Tagging Rules command

TAGGING RULES (COMMAND)

Edit & Tagging Rules

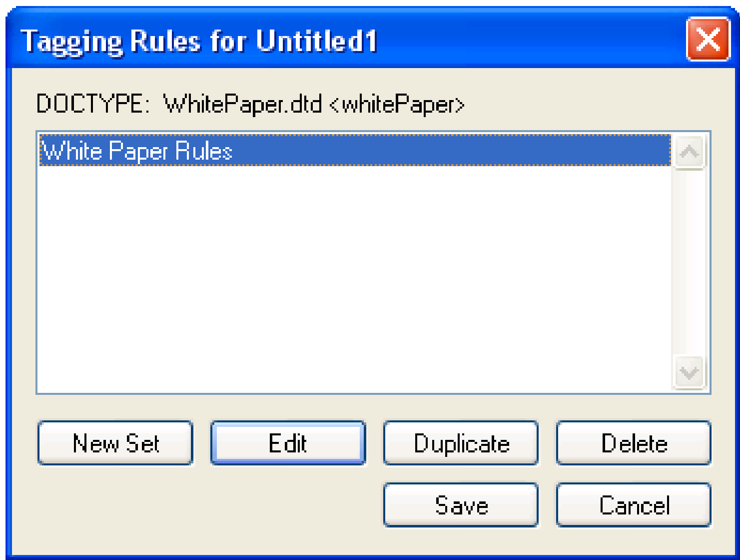
The **Tagging Rules** command displays the **Tagging Rules** dialog box, which lets you create, edit, duplicate, and delete tagging rule sets for the active XML document.

TAGGING RULES (DIALOG BOX)

Edit & Tagging Rules

The **Tagging Rules** dialog box lets you create, edit, duplicate, and delete tagging rule sets for the DOCTYPE of the active XML document. The tagging rule sets

you create in this dialog box are available in the **Tagging Rule Set** pop-up menu in the document's **XML Workspace** palette.



Tagging Rules dialog box

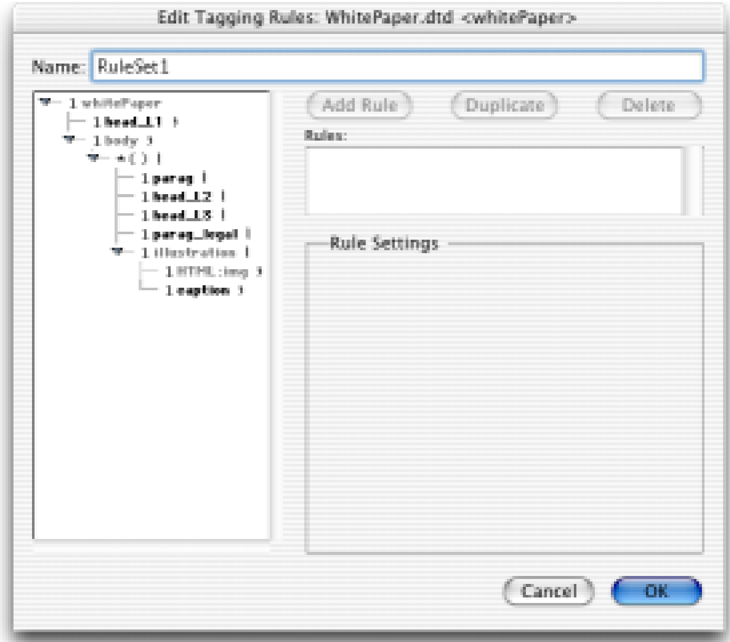
- The **DOCTYPE** field displays the DOCTYPE used by the active document, including the DTD and the <root element>.
- The **Tagging Rule Set** list displays a list of tagging rule sets for the XML document and lets you choose a tagging rule set to edit.
- The **New Set** button displays the **Edit Tagging Rules** dialog box, which lets you create a new tagging rule set for the XML document.
- The **Edit** button displays the **Edit Tagging Rules** dialog box, which lets you edit the tagging rule set selected in the **Tagging Rule Set** list.
- The **Duplicate** button creates a copy of the tagging rule set selected in the **Tagging Rule Set** list. For more information about the **Edit Tagging Rules** dialog box, see "Edit Tagging Rules (dialog box)" in this section.
- The **Delete** button removes the tagging rule set selected in the **Tagging Rule Set** list.
- The **Save** button saves changes made to any tagging rule set in the **Tagging Rules** dialog box. When you click **Save**, avenue.quark closes the dialog box.
- The **Cancel** button closes the **Tagging Rules** dialog box and discards any changes made since opening it.

Tagging rule sets are saved as part of avenue.quark XML documents; they are *not* stored in a preferences file.

EDIT TAGGING RULES (DIALOG BOX)

Edit & Tagging Rules & New Set, Edit, or Duplicate

Clicking **New Set**, **Edit**, or **Duplicate** in the **Tagging Rules** dialog box displays the **Edit Tagging Rules** dialog box, which lets you create and edit tagging rule sets.



Edit Tagging Rules dialog box

- The **Name** field lets you name a new tagging rule set or rename an existing tagging rule set.
- Below the **Name** field are the **DTD Tree** list, the **Rules** list, and the **Rule Setting** area. The **DTD Tree** list (on the left side of the dialog box) displays a tree view of the DTD and its element types — including optional element types — and lets you select an element type for which to define tagging rules. The right side of the dialog box lets you specify tagging rules for the selected element type.

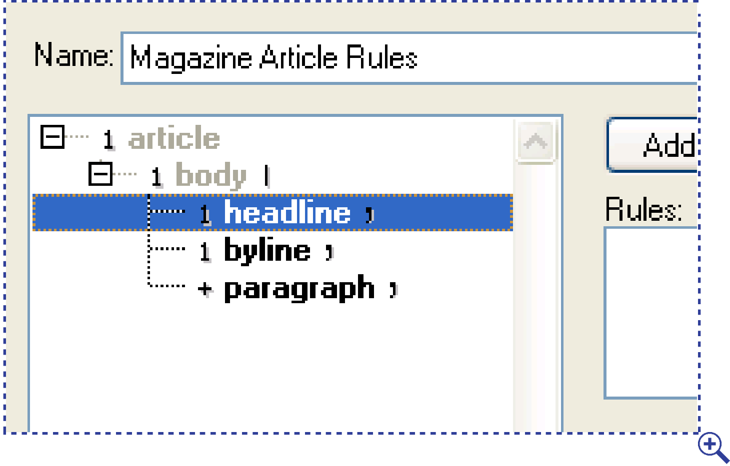
You can use the Tab key to move from the selected element in the DTD list through the steps in the dialog box required to define a tagging rule.

DTD TREE (LIST)

Edit & Tagging Rules & New Set, Edit, or Duplicate

The **DTD Tree** list displays the selected DTD. You can scroll through the list using the horizontal and vertical bars or the arrow keys. You can display or hide

the contents of container elements by clicking the and > disclosure triangles (Mac OS) or the ⊞ and ⊟ disclosure boxes (Windows).



DTD Tree list

A symbol at the beginning of each element's name indicates how many of that element may be contained by the parent element type:

- 1 indicates one and only one
- indicates zero or one
- indicates one or more
- indicates zero or more

A symbol at the end of an element's name indicates the relationship this element can have with other elements:

- indicates that this element must be followed by the next element
- | indicates that either this element or the next element may be used here

---

If an element's name is bold and black, you can create tagging rules for that element type. If an element's name is bold, black, and italic, the tagging rule set already contains at least one rule for that element type. If an element's name is unavailable, no tagging rules may be created for that element type.

ADD RULE, DUPLICATE, AND DELETE (BUTTONS)

Edit & Tagging Rules & New Set, Edit, or Duplicate

The **Add Rule**, **Duplicate**, and **Delete** buttons let you add, duplicate, and delete tagging rules in the **Rules** list.

Add Rule, Duplicate, and Delete buttons

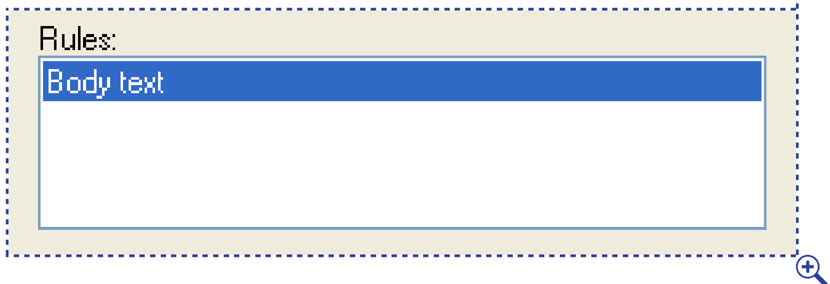
- Clicking the **Add Rule** button lets you add a tagging rule to the **Rules** list
- Clicking the **Duplicate** button lets you duplicate the tagging rule selected in the **Rules** list
- Clicking the **Delete** button lets you delete the tagging rule selected in the **Rules** list

RULES (LIST)

Edit & Tagging Rules & New Set, Edit, or Duplicate

The **Rules** list displays the tagging rules for the selected element type. The rule names in this list reflect the selections made in the **Rule Setting** area for each rule.

To edit a tagging rule, select its name in the list and change its definition in the **Rule Settings** area.

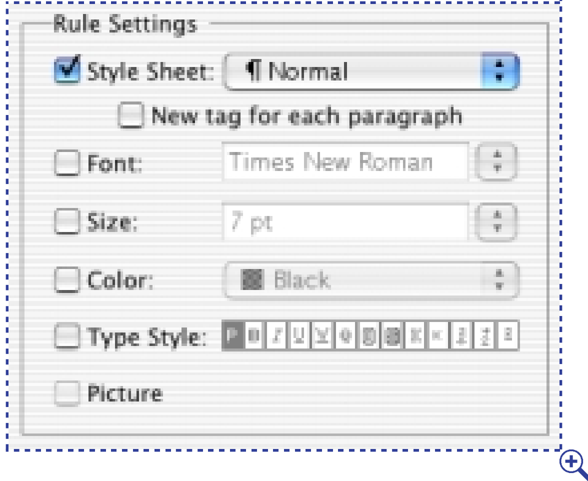


Rules list

RULE SETTINGS (AREA)

Edit & Tagging Rules & New Set, Edit, or Duplicate

The **Rule Settings** area lets you configure the tagging rule selected in the **Rules** list. The options available in this area are based on the active QuarkXPress layout.



Rule Settings area

- The **Style Sheet** check box and pop-up menu let you specify that text that uses a particular paragraph or character style sheet should be tagged with the selected element type. Checking **New tag for each paragraph** specifies that avenue.quark should tag each paragraph styled with the indicated style sheet as a separate element (if the DTD permits it). Unchecking **New tag for each paragraph** specifies that avenue.quark should tag a range of paragraphs styled with the indicated style sheet as a single element.
- The **Font** check box and pop-up menu let you specify that only text that uses a particular font should be tagged with the selected element type.
- The **Size** check box and pop-up menu let you specify that only text of a particular size should be tagged with the selected element type.
- The **Color** check box and pop-up menu let you specify that only text that uses a particular color should be tagged with the selected element type.
- The **Type Style** check box and button list let you specify that only text that uses a particular type style should be tagged with the selected element type.
- The **Picture** check box lets you tag pictures for XML export. You can import tagged pictures back into QuarkXPress using Placeholders or XML Import QuarkXTensions software.

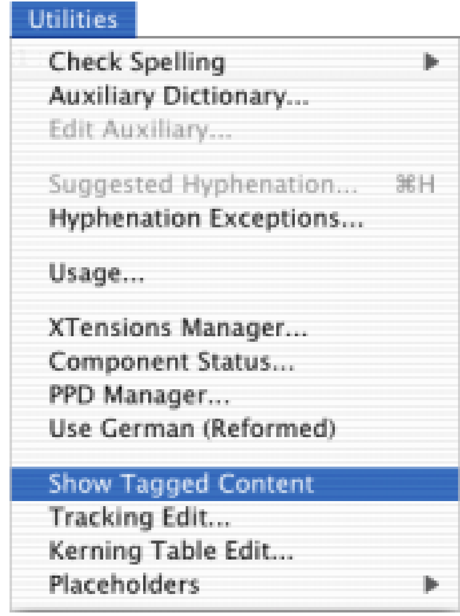
UTILITIES MENU

*The **Utilities** menu lets you show tagged content in the active QuarkXPress layout.*

SHOW TAGGED CONTENT, HIDE TAGGED CONTENT (COMMAND)

Utilities menu

The **Show Tagged Content** command displays tagged content in the active QuarkXPress layout with a particular color. You can specify this color in the **avenue.quark** pane of the **Preferences** dialog box (**QuarkXPress** & **Preferences** & **avenue.quark** on Mac OS or **Edit** & **Preferences** & **avenue.quark** on Windows).

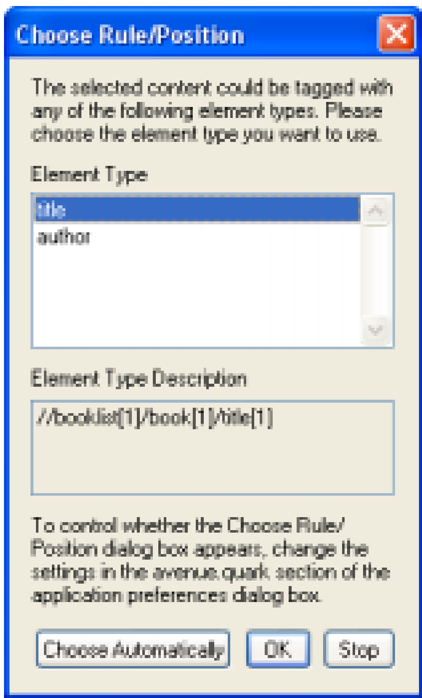


Show Tagged Content command

CHOOSE RULE/POSITION DIALOG BOX

The **Choose Rule/Position** dialog box displays in two different situations:

- When two different tagging rules could be applied to a given range of text.
- When a new element could be inserted at a number of different places in the active branch of the XML tree.



The Choose Rule/Position dialog box lets you determine which rule to use when text matches more than one tagging rule. It also lets you choose a position for a tagged element when more than one position is available.

The controls in this dialog box work as follows:

- The **Element Type** field lists the element types that can be applied to the selected text. The options in this list are sorted as described in "Rule weighting."
- The **Element Type Description** field displays the path of the element that will be created if the element type selected in the **Element Type** field is used. A number in brackets indicates the position of a child relative to its parent; for example, "**/whitePaper/body[2]** "represents a new <body> element that will be the second child element of <whitePaper>. A dot at the end of a path indicates that if that path is selected, the text will be appended to the end of the indicated element.
- The **Choose Automatically** button tags the selected text using the avenue.quark automatic tagging algorithm. This algorithm works as if the first two check boxes in the **Tagging Rule/Position Options** area (**QuarkXPress** & **Preferences** & **avenue.quark** pane on Mac OS or **Edit** & **Preferences** & **avenue.quark** pane on Windows) were checked and automatically selects the top match in the **Element Type** field. After you click this button, rule-based tagging then continues with these settings until the end of the selected text is reached.

- The **Stop** button stops the tagging process without tagging the selected text. All content tagged before this button is clicked remains in the **XML Workspace** palette.
- The **OK** button tags the selected text with the element type listed in the **Element Type** field, at the position indicated by the **Element Type Description** field. Tagging then resumes.

RULE PRIORITIES

When the selected text can be tagged using more than one tagging rule, avenue.quark displays the **Choose Rule/Position dialog box** (unless **Always use the first applicable tagging rule** is checked in the **avenue.quark** pane of the **Preferences** dialog box). The rules that can be applied are then sorted according to their weight and displayed in the **Element Type** field.

Tagging rules are prioritized as follows, with items nearer the top of the list having a higher priority than items nearer the bottom:

- A rule that targets a character style sheet, a color, and a type style
- A rule that targets a character style sheet, plus either a color or a type style
- A rule that targets a character style sheet only
- A rule that targets a color and a type style only
- A rule that targets either a color or a type style only
- A rule that targets a paragraph style sheet, a color, and a type style
- A rule that targets a paragraph style sheet plus either a color or a type style
- A rule that targets a paragraph style sheet only.

For example, say you have a paragraph style named “Body-P” and a character style sheet named “Italic-C,” and you’re applying the “Italic-C” style sheet to emphasized words in your “Body-P” paragraphs. You’d like to tag all your paragraphs as <bodyText> elements, with the emphasized words in those paragraphs tagged as <emphasized> elements.

When avenue.quark encounters an “Italic-C” word in a “Body-P” paragraph, two rules can be applied: the rule that says text styled as “Body-P” should be tagged as <bodyText>, and the rule that says text styled as “Italic-C” should be tagged as <emphasized>. The second rule is given a higher priority, according to the list above, and thus is listed first (and chosen automatically if you click **Choose Automatically**). As a result, your XML looks something like this:

```
<bodyText>I want the <emphasized>other</emphasized> banana.</bodyText>
```

This example assumes that the DTD allows the <bodyText> element to contain both PCDATA and <emphasized> elements.

The rule weighting algorithm is designed to make the **Choose Automatically** button in the **Choose Rule/Position** dialog box work correctly in most tagging situations. If this button does not give you the results you want, though, you can still choose a rule manually in the **Element Type** field.

The following table shows rule weighting in a tabular form:

RULE WEIGHT	PARAGRAPH STYLE SHEET	CHARACTER STYLE SHEET	COLOR	TYPE STYLE
Highest		•	•	•
		•	(•	OR •)
		•		
			•	•
			(•	OR •)
	•		•	•
	•		(•	OR •)
Lowest	•			

Chapter 5: Tagging Rule Sets

One of the most difficult parts of working with XML is getting content from its original format into XML format. A QuarkXPress layout may be organized with style sheets and other conventions, but how do you translate that kind of organization into XML?

Avenue.quark helps you automate this process. Given a QuarkXPress layout and a DTD, avenue.quark lets you create a "tagging rule set," which can automatically map combinations of QuarkXPress style sheets, colors, and type styles to element types in a DTD.

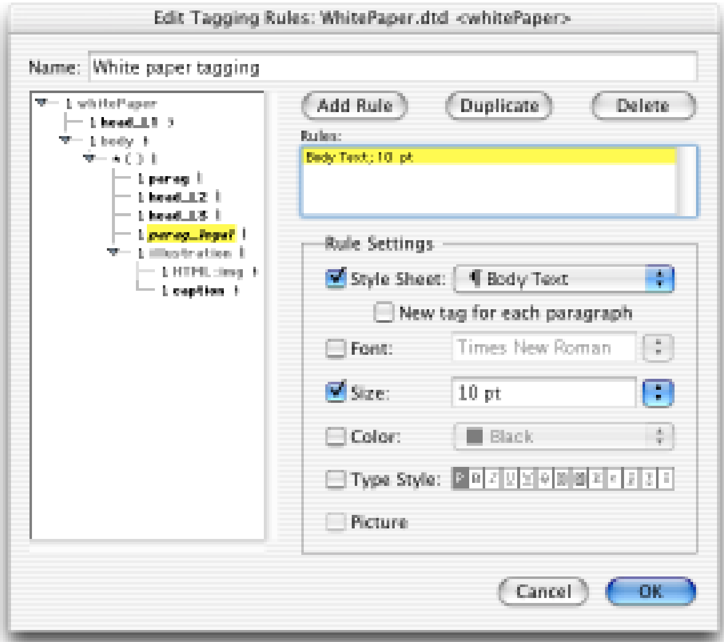
UNDERSTANDING RULE-BASED TAGGING

A tagging rule set lets you associate QuarkXPress style sheets, and text styles with elements in a DTD. You can use a tagging rule set to automate part of the process of tagging a QuarkXPress layout. To use tagging rule sets in rule-based tagging, see Chapter 6, "Tagging Content."

WHAT IS A TAGGING RULE SET?

A tagging rule set is a named set of tagging rules that are all based on a single DTD. Each tagging rule specifies which style sheets, colors, and text styles should be mapped to their corresponding elements. Tagging rules let you specify that when you use rule-based tagging, content that meets a specific set of criteria should be tagged with a particular element name. For example, you could set up a tagging rule indicating that each paragraph that uses the "Headline" paragraph style sheet should be tagged as a `<headline>` element.

You could add a rule to specify that italicized text in paragraphs that use the "01 Title" style sheet should be tagged with `<emphasis>` tags, like this:

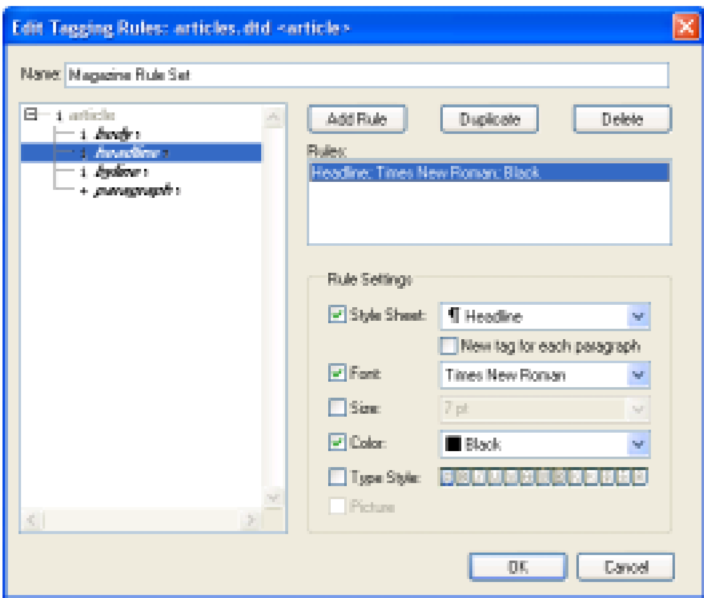


Tagging rule sets let you nest elements within other elements

Given the two tagging rules specified above, a paragraph that uses the “Title” paragraph style sheet and contains italic text might be tagged as follows:

<title>What the Maid <emphasis>Really </emphasis>Saw</title>

In order for the selected element type to be used, *all* the criteria in the **Rule Setting** area must be met. For example, the following tagging rule indicates that only text that uses the “Author” paragraph style sheet *and* is red *and* is bold should be tagged with the <author> element type:



All tagging rule criteria must be met for a tag to be used.

If there is more than one kind of formatting you want mapped to a particular element type, you can create additional rules for that element type. For example, say you have two different paragraph style sheets for names; one style sheet for the first name in a list, and another style sheet for the other names in the list. (This is commonly done for spacing reasons.) You could simply create two tagging rules for the `<name>` element type, one that maps the "First Name" style sheet to `<name>` and one that maps the "Remaining Names" style sheet to `<name>`. Avenue.quark would then tag paragraphs that met *either* rule's criteria as `<name>` elements.

In many workflows, only administrative personnel should create tagging rule sets.

HOW RULE-BASED TEXT TAGGING WORKS

When you use rule-based tagging on a box full of text, avenue.quark goes through that text from beginning to end and tries to tag the text to match the DTD. At any given point in this process, avenue.quark is looking ahead to see if it can find text that matches a rule that fits the DTD. Text that cannot be tagged according to any tagging rule is ignored.

A recursive DTD is a DTD that allows an element to contain itself. For example, if a DTD allows an <A> element to contain an <A> element (directly or indirectly), then that DTD is recursive.

If you use rule-based tagging with a recursive DTD, you can create an "endless loop" situation. To avoid such problems, create a subset of the DTD that is not recursive, and use that subset DTD when tagging layouts with avenue.quark. (Make sure, however, that layouts created with the subset DTD are still valid according to the full DTD.)

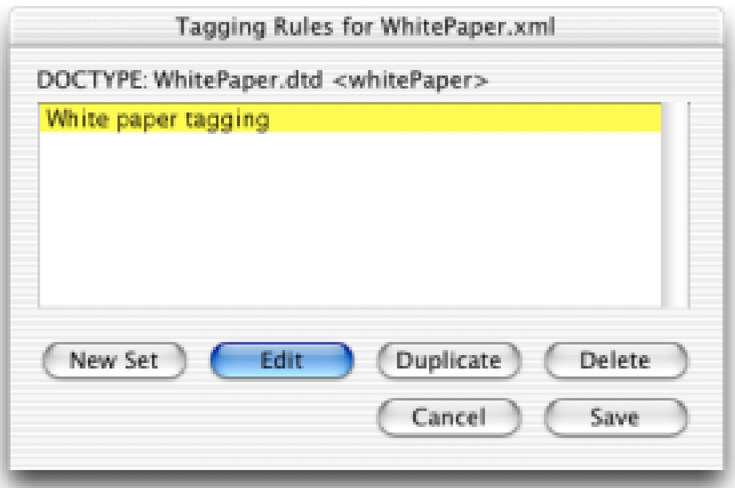
WORKING WITH TAGGING RULE SETS

A tagging rule set lets you associate QuarkXPress style sheets, colors, and text styles with elements in a DTD. You can use a tagging rule set to automate part of the process of tagging a QuarkXPress layout. To use tagging rule sets in rule-based tagging, see Chapter 6, "Tagging Content."

CREATING A TAGGING RULE SET

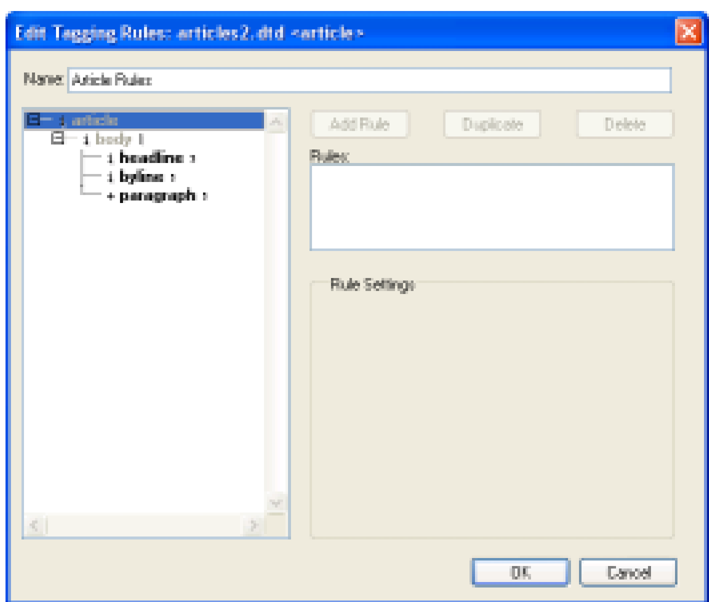
A tagging rule set lets you specify how text should be tagged when you use rule-based tagging. To create a tagging rule set:

1. Create or open the XML document for which you want to create a tagging rule set.
2. Create or open a QuarkXPress project that contains all the style sheets and colors you want to use in the tagging rule set.
3. Choose **Edit** & **Tagging Rules**. The **Tagging Rules** dialog box displays.



Create a new tagging rule set from the Tagging Rules dialog box.

4 Click the **New Set** button to create a new tagging rule set. The **Edit Tagging Rules** dialog box displays, and the DOCTYPE's root element and file name are listed in the title bar.



The Edit Tagging Rules dialog box lets you create and edit a tagging rule set.

5 Enter a name for the tagging rule set in the **Name** field.

6 Select a bold element type in the list on the left. (If an element type's name is unavailable, that means the DTD does not allow it to be associated with rules.) To expand a container element and display all the elements it contains, click the > disclosure triangle (Mac OS) or the ⊞ disclosure box (Windows) next to that element. To view more of the DTD, scroll through the list.

7 To begin adding a new rule to the tagging rule set, click **Add Rule.** A blank rule is added to the **Rules** list.

8 To configure the tagging rule to automatically tag text that uses a particular style sheet, click **Style Sheet** and then choose a style sheet name from the **Style Sheet** pop-up menu. If you want a consecutive series of paragraphs that use the indicated paragraph style sheet to be inserted into separate elements, check **New tag for each paragraph**; if you want a consecutive series of paragraphs that use the indicated style sheet to be inserted into a single element, leave this box unchecked. Style sheets displayed in italics are not present in the active QuarkXPress layout.

In order for the **New tag for each paragraph** option to work, the DTD must support multiple sequential occurrences of the selected element.

9 To configure the tagging rule to automatically tag text that uses a particular font, click **Font** and then choose a font from the **Font** pop-up menu.

10 To configure the tagging rule to automatically tag text that uses a particular font size, click **Size** and then choose a font size from the **Size** pop-up menu.

11 To configure the tagging rule to automatically tag text that uses a particular color, click **Color** and then choose a color name from the **Color** pop-up menu. Color names displayed in italics are not present in the active QuarkXPress layout.

Tagging rule sets contain only the *names* of style sheets and colors. If you change the name of a style sheet or color in the project, you must update the tagging rule set as well.

---

12 To configure the tagging rule to automatically tag text that uses a particular combination of type styles, click **Type Style** and then click the icons to indicate which type styles should be tagged. A type style icon with a black background indicates that text must use this type style to be tagged; a type style icon with a white background indicates that text with this type style will *not* be tagged; and a type style icon with a gray background indicates that this type style will not be taken into account during rule-based tagging.

Remember that text is not tagged until you perform rule-based tagging on it. To perform rule-based tagging, see Chapter 6, "Tagging Content."

13 To add a new rule for the selected element type, click **Add Rule** and then repeat steps 8 through 10. To base a new rule on an existing rule, select the existing rule in the **Rules** list; click **Duplicate** to create a copy of that rule; and then reconfigure the duplicate rule.

14 To delete a rule for the selected element type, select the rule in the **Rules** list and then click **Delete.**

15 To save your changes to the tagging rule set, click **OK.**

16 Click **Save** to close the **Tagging Rules** dialog box.

Element types for which rules have been created display italicized in the DTD list.

If an element type occurs more than once in the DTD tree, creating a rule for one occurrence applies that rule to all occurrences.

To create a tagging rule set that includes rules for style sheets from several projects, create a new project, append all the style sheets from their various projects (**File** & **Append** & **Style Sheets** tab), and then create your tagging rules.

EDITING, DUPLICATING, AND DELETING TAGGING RULE SETS

The **Tagging Rules** dialog box (**Edit** menu) lets you edit, duplicate, and delete tagging rule sets. Select a tagging rule set in the list and click one of these buttons:

- Clicking **Edit** opens the tagging rule set so you can modify it.
- Clicking **Duplicate** creates a copy of the tagging rule set that you can rename and modify.
- Clicking **Delete** removes the tagging rule set from the list.

Chapter 6: Tagging Content

XML lets you tag, or label, the content of layouts. Tagging lets you label the parts of documents. This in turn enables you to use that tagged content in a wide variety of ways, including on the World Wide Web.

Avenue.quark gives you a unique means of tagging text and pictures in QuarkXPress documents. You can tag elements manually or use tagging rule sets to automate part of the tagging process.

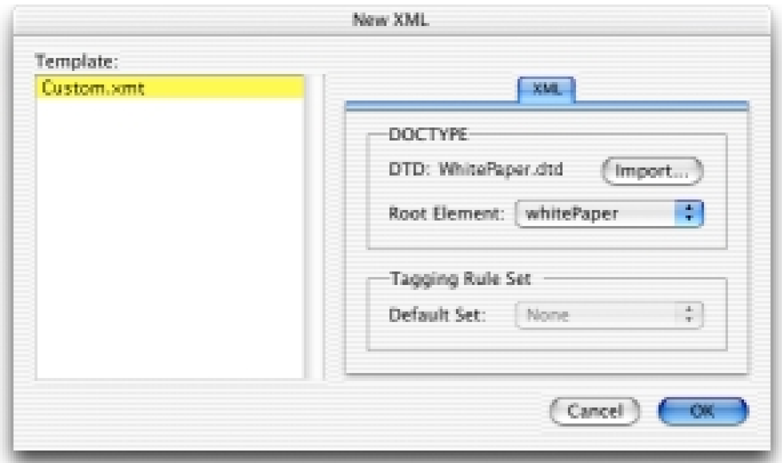
CREATING, OPENING, AND SAVING XML DOCUMENTS

Avenue.quark lets you create and open XML documents from the ***File*** *menu. You can save XML documents using buttons on the* ***XML Workspace*** *palette.*

CREATING AN XML DOCUMENT

To create a new XML document:

1 Choose **File** & **New** & **XML**, or press C+Shift+X (Mac OS) or Ctrl+Shift+X (Windows). The **New XML** dialog box displays.



The New XML dialog box lets you create a new XML document.

2 To base the new XML document on an XML template, select an item in the **Template** list. To create a new XML document that is *not* based on a template, click **Import** in the **DOCTYPE** area, select a DTD, and then choose a root

element from the **Root Element** pop-up menu. For information about XML templates, see “Working with XML Templates” in this chapter.

3 If you are basing the new XML document on a template and you plan to use rule-based tagging, choose a default tagging rule set from the **Default Set** pop-up menu.

4 Click **OK**. The new XML document displays in a new **XML Workspace** palette. Once you complete this step, the DTD and root element of the XML document cannot be changed.

By default, a new XML document contains only those elements that are mandatory at the root level according to the DTD, plus any mandatory children of those elements. If the DTD requires one of a list of non-optional elements (for example, (a | b | c)), avenue.quark uses the first element in the list (here, a). If the DTD requires one of a list of elements, and one or more of those elements is optional (for example, (a | b | c?)), avenue.quark leaves the parent element empty.

Avenue.quark creates only valid XML documents (that is, documents that adhere to their DOCTYPE).

If you want to create a new document based on an XML template file that’s not in the QuarkXPress “Templates” folder, choose **File** & **Open** and open the XML template file.

OPENING AN XML DOCUMENT

Avenue.quark lets you open XML documents created by avenue.quark or any other valid XML document that includes its DTD or references an available DTD. To open an XML document in avenue.quark:

1 Choose **File** & **Open** (C+O on Mac OS, Ctrl+O on Windows).

2 *Windows only:* Choose **XML (*.xml)** from the **Files of type** pop-up menu.

3 Use the controls in the dialog box to locate the document you want to open; then select the document in the list. Avenue.quark can open only XML documents that have a file extension of “.xml”.

4 Click **Open.**

5. If the XML file was generated by avenue.quark, avenue.quark attempts to open the QuarkXPress project that most recently contributed content to it. If avenue.quark cannot find the QuarkXPress project, a dialog box displays allowing you to open the XML file without the project or to cancel the **Open** procedure.

   If an “XML document error” dialog box displays, this means the project you are opening contains an error. To help you easily locate the error, the dialog box displays the name of the DTD containing the error and all its lines, with the problem area highlighted. At the bottom, the dialog box lists the exact location of the error within the DTD. If the DTD contains more than one error, click **Next** to see the location of the second problem. To resolve errors, click **OK** in the error dialog box and open the DTD in question to correct it.

6. If the QuarkXPress project is available, avenue.quark compares the content in the project to the corresponding content in the XML document and notifies you of any differences.

   Avenue.quark supports the UTF-8 and UTF-16 (Unicode) encodings, and automatically adds an encoding specification when you save an XML file. If an XML file does *not* have an encoding specification, avenue.quark assumes its encoding to be UTF-8. For information about encodings, see Appendix C, “Understanding Encodings,” in Chapter 7, “Appendices.”

OPENING .XML AND .XMT FILES IN QUARKXPRESS (WINDOWS ONLY)

Although avenue.quark lets you create files with the suffixes “.xml” and “.xmt”, it does not register these suffixes in the Windows registry. Because these suffixes are not registered, double-clicking an “.xmt” or “.xml” file does not open that file in QuarkXPress by default.

To open an XML file (a file that ends in “.xml”) with avenue.quark, launch QuarkXPress with avenue.quark installed, choose **File** & **Open,** select the file, and then click **Open.**

To open an avenue.quark XML template (a file that ends in “.xmt”) with avenue.quark:

1. Put the XML template file in the “Templates” folder (in your QuarkXPress application folder).
2. Launch QuarkXPress with avenue.quark installed.
3. Choose **File** & **New** & **XML.**
4. Select the XML template’s file name in the **Template** list and click **Open.**

REGISTERING THE .XML AND .XMT SUFFIXES (WINDOWS ONLY)

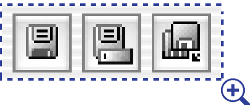
If you would like to register the ".xmt" file type with QuarkXPress in Windows, do the following:

1. Open a folder window.
2. Choose **View** & **Folder Options** (Windows 98 or Windows 2000) **View** & **Options** (Windows NT), or **Tools** & **Folder Options** (Windows XP).
3. Click the **File Types** tab to view registered file types.
4. Use the **New Type** button to create a new registered file type.
5. Enter "avenue.quark XML template" in the **Description of type** field.
6. Enter "xmt" in the **Associated extension** field.
7. Click **New** in the **Actions** area.
8. When the **New Action** dialog box displays, enter "Open" in the **Action** field.
9. Click **Browse,** locate your QuarkXPress application file, and then click **Open**.

You can use the same procedure for XML files (".xml"). If a registered file type already exists for XML files, just edit its **Open** action to point to the QuarkXPress application file. See the Windows documentation for more specific instructions.

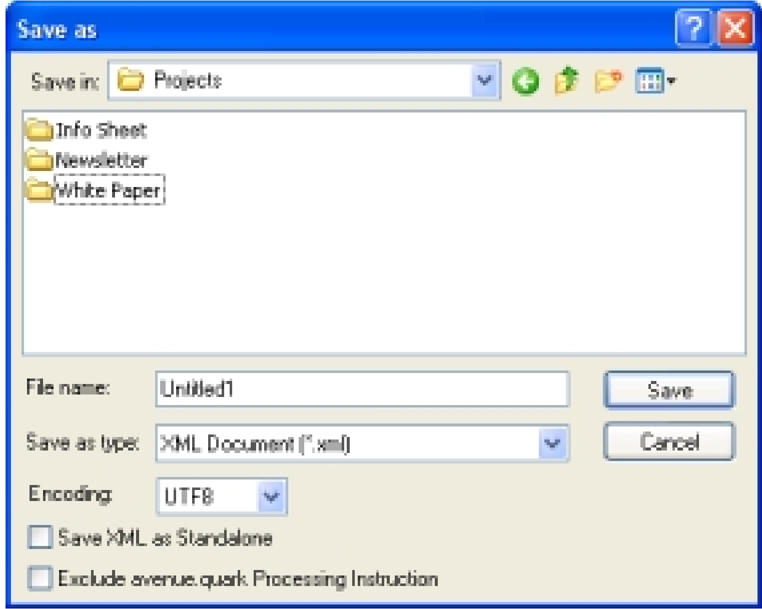
SAVING AN XML DOCUMENT

Buttons on the **XML Workspace** palette lets you save XML documents under their current names or under a new name.



From left to right: the Save, Save As, and Revert to Saved buttons.

- Click the **Save** button to save the active XML document with its current name.
- Click the **Save As** button to save the active XML document with a new name. The **Save As** dialog box displays; name the XML file; choose a type from the **Type** pop-up menu; choose an encoding method from the **Encoding** pop-up menu; check or uncheck **Save XML as Standalone**; and then click **Save**.



The Save as dialog box lets you name an XML file, specify whether to save the file as an XML document or avenue.quark template, and specify an encoding method.

- Click the **Revert to Saved** button to revert to the last-saved version of the active XML document.

When you save an XML document in avenue.quark, a number of things happen:

- If **Save XML as Standalone** is checked, avenue.quark adds a copy of the DTD to the file's internal subset, so the file can be opened and validated on computers where the DTD is unavailable. (If **Save XML as Standalone** is unchecked, the appropriate DTD must be available in order for avenue.quark to open the XML file.)
- If **Exclude avenue.quark processing instruction** is unchecked, avenue.quark adds several processing instructions to the file, indicating the name of the QuarkXPress layout that most recently contributed content to the document and the location of that content in the QuarkXPress layout.
- If the XML document contains an empty ID attribute for which a value is required, a dialog box displays. At that point, you can complete the save (resulting in an invalid XML document), or cancel the save and view the empty attribute in the **XML Workspace** palette.
- If the XML document contains an ID attribute with the same value as another ID attribute in the same document, a dialog box displays. At that point, you can complete the saving process (resulting in an invalid XML document),

or cancel the saving process and view the duplicate attribute in the **XML Workspace** palette.

- If the XML document contains an empty IDREF attribute for which a value is required, a dialog box displays. At that point, you can complete the saving process (resulting in an invalid XML document), or cancel the saving process and view the empty attribute in the **XML Workspace** palette.
- If the XML document contains an IDREF attribute with a value that does not match the value of any ID attribute in the active XML document, a dialog box displays. If the IDREF attribute value refers to an ID attribute value in an XML document referred to by the active XML document, you can save the same. If not, you can cancel the saving process and view the problematic IDREF attribute in the **XML Workspace** palette.
- If the XML document contains empty CDATA attributes for which a value is required, an underscore (_) is inserted in those attributes.

WORKING WITH XML TEMPLATES

An XML template is an avenue.quark XML document that contains a DTD, a root element specification, and an optional default tagging rule set. An XML template may also contain a "starter" set of elements, attributes, comments, processing instructions, PCDATA *blocks, and content. The purpose of XML templates is to save users from having to repeat the same setup steps over and over again when creating a series of XML documents that are all based on the same DTD, root element, and tagging rule set.*

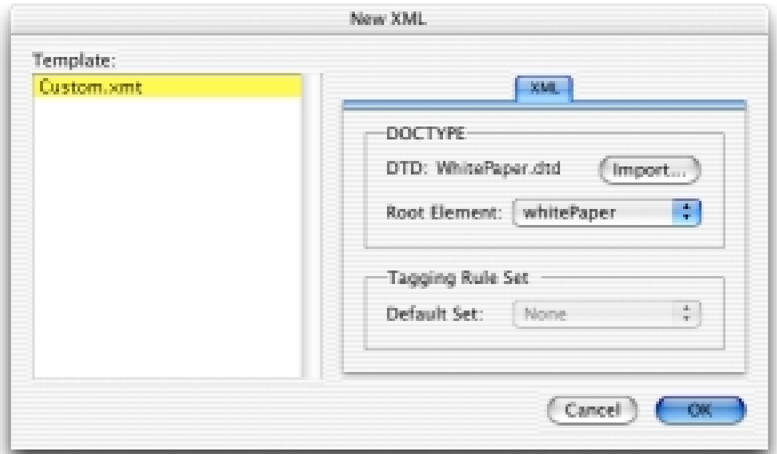
To be available in avenue.quark, XML templates must be stored in the "Templates" folder, which is in the same folder as the QuarkXPress application. XML templates must have a file name suffix of ".xmt", even on Mac OS.

CREATING AN XML TEMPLATE

To create a new XML template:

1 Choose **File** & **New** & **XML**, or press C+Shift+X (Mac OS) or Ctrl+Shift+X (Windows). The **New XML** dialog box displays.

2 If you want to base the template on another template, select that template's name in the **Template** list and then go to step 4.



The New XML dialog box lets you create new XML templates.

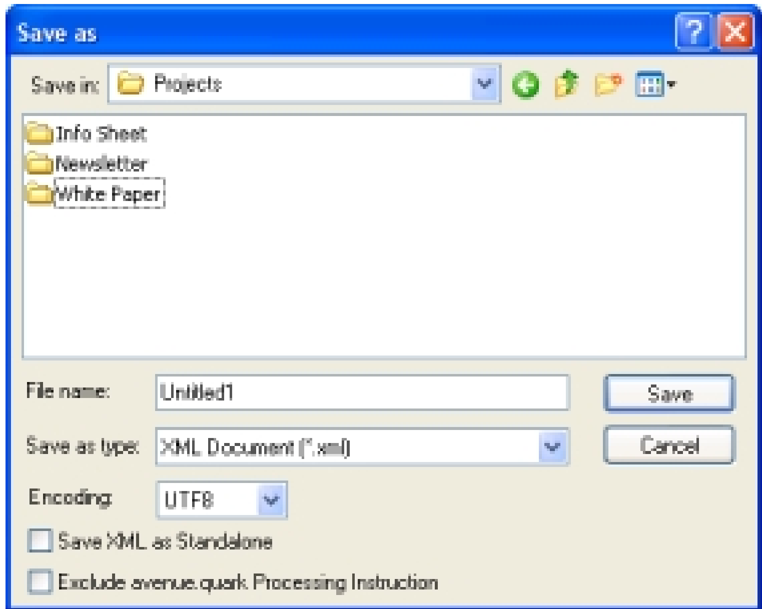
3 If you want to create the template from scratch, select **Custom** in the **Template** list; click **Import** in the **DOCTYPE** area and select a DTD file; then choose a root element from the **Root Element** pop-up menu.

4 If you are basing the new template on an existing template and you plan to use rule-based tagging, choose a default tagging rule set from the **Default Set** pop-up menu.

5 Click **OK**. The new XML document displays in a new **XML Workspace** palette.

6 Add any elements, attributes, comments, processing instructions, PCDATA blocks, or content that you want the template to contain.

7 Click the **Save As** button. The **Save as** dialog box displays.



The Save as dialog box lets you name an XML file, specify whether to save the the file as an XML document or avenue.quark template, and specify an encoding method for the file.

8 Enter a name for the file in the **Save Current XML as** field. Keep in mind that XML templates must have a file name that ends in ".xmt", even on Mac OS.

9 Choose **avenue.quark Template** from the **Type** pop-up menu.

10 Choose an encoding method from the **Encoding** pop-up menu.

11 If you want the template to be available in the **New XML** dialog box, navigate to the "Templates" folder inside your QuarkXPress folder. (Templates that are *not* stored in the "Templates" folder can be opened by choosing **File** & **Open**.)

12 Click **Save**.

Templates are always saved as stand-alone documents because they must contain a DTD.

WORKING WITH XML DOCUMENT CONTENT

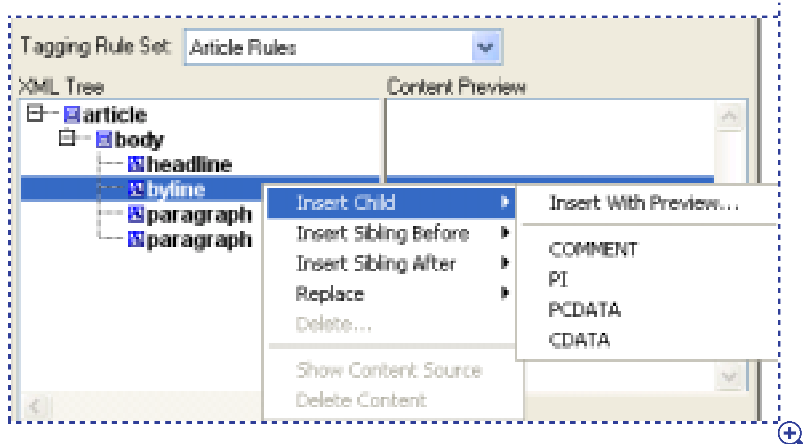
The ***XML Workspace*** *palette's hierarchical* ***XML Tree*** *list makes it easy for you to view and work with the content of XML documents. A handy context menu makes it easy for you to insert, delete, and replace elements, comments,* PCDATA *blocks, and processing instructions.*

INSERTING OR REPLACING AN ELEMENT, COMMENT, PCDATA BLOCK, OR PROCESSING INSTRUCTION

You can insert a new element, comment, PCDATA block, or processing instruction above, below, or as a child of the item selected in the **XML Tree** list (**XML Workspace** palette). You can also replace the selected item with a different item if the DTD permits it. To insert or replace an item in the active XML document:

1 In the **XML Workspace** palette, select an item in the **XML Tree** list.

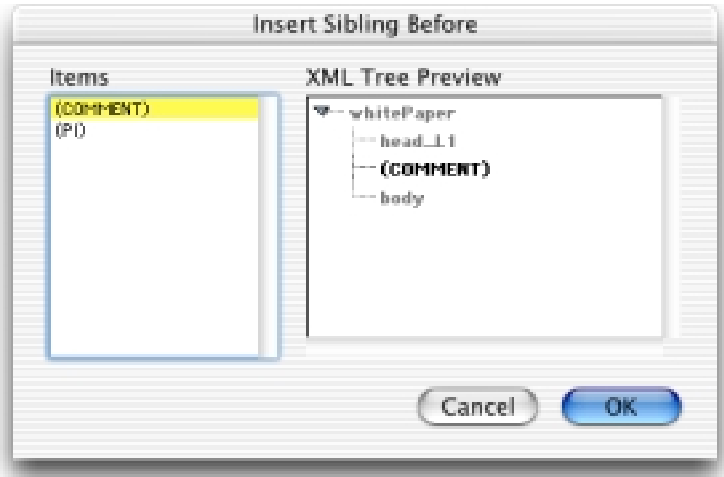
2 Control+click (Mac OS) or right-click (Windows) the item to display the **XML Tree** context menu, and then choose **Insert Child**, **Insert Sibling Before**, **Insert Sibling After**, or **Replace**. A submenu displays.



The submenu for the Insert Child, Insert Sibling Before, Insert Sibling After, and Replace commands lets you choose what kind of element to insert or substitute.

3 If you want to insert or substitute an element, comment, processing instruction, or PCDATA block without a preview, simply choose it from the submenu.

4 If you would like to see a preview before you insert or replace, choose **Insert with Preview**. The **Insert Child**, **Insert Sibling Before**, or **Insert Sibling After Preview** dialog box displays.



Insert Sibling Before dialog box.

The **Items** list displays a list of items that may be inserted or substituted. The **XML Tree Preview** list shows the selected item and any of its mandatory children (in black), in the context of the XML tree (in gray); you can think of it as a preview of how the document will look after the change. If any elements will need to be *deleted* in the process, they display in red struck-through text. Click **OK** to complete the insertion or replacement, or **Cancel** to stop it.

Some elements have mandatory children. If you insert such an element, its mandatory children must also be inserted. Clicking an element's > disclosure triangles (Mac OS) or ⊞ disclosure box (Windows) displays any child elements that must also be inserted along with that element. Clicking this icon does *not* display any optional children an element might have.

If an inserted element requires one of a list of non-optional elements (for example, (a | b | c)), avenue.quark uses the first element in the list (here, a). If the element requires one of a list of elements, and one or more of those elements is optional (for example, (a | b | c?)), avenue.quark leaves the element empty.

If the element you want is unavailable, it means the DTD does not permit new elements of that type to be inserted in this position.

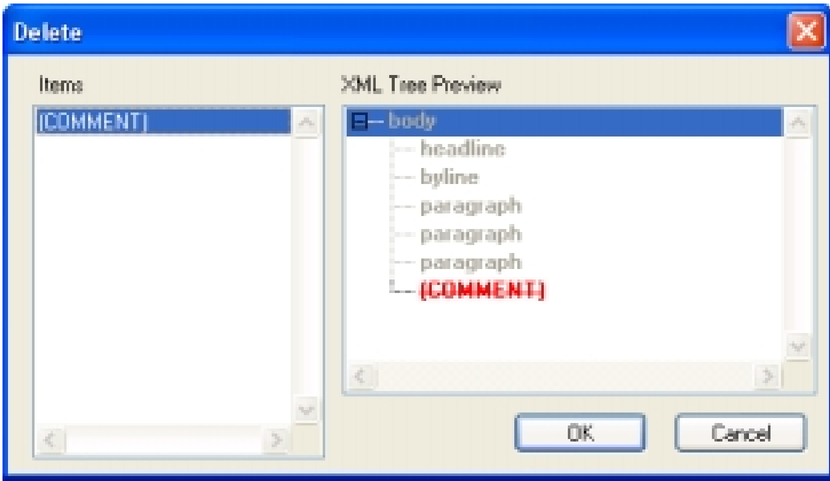
What is a PCDATA block? A PCDATA block is an avenue.quark construction that lets you divide the text in an element into separate parts. PCDATA blocks are visible only in avenue.quark; in the exported XML, consecutive PCDATA blocks are merged.

DELETING AN ELEMENT, COMMENT, PCDATA BLOCK, OR PROCESSING INSTRUCTION

To delete an element, comment, PCDATA block, or processing instruction:

1 In the **XML Workspace** palette, select the item you want to delete in the **XML Tree** list.

2 Control+click (Mac OS) or right-click (Windows) the item to display the **XML Tree** pop-up menu, and then choose **Delete**. The **Delete** dialog box displays.



Delete dialog box.

If the **Delete** menu item is unavailable, it means the DTD does not permit the deletion of the element you've selected.

The **Items** list displays the item being deleted. The **XML Tree Preview** list shows what the XML tree will look like after the deletion; items that will be deleted are indicated by red struck-through text.

3 Click **OK** to complete the deletion, or **Cancel** to stop it.

TAGGING TEXT

When you use avenue.quark to tag content, you must use a DTD; see "Working With DTDs" in Chapter 1, "Avenue.quark Basics". Tagging text is the process of associating that text with element types in the appropriate DTD.

There are several ways to tag text in avenue.quark. You can tag text manually, automate tagging with a tagging rule set, or enter content manually.

TAGGING TEXT MANUALLY

Use manual text tagging when you want precise control over exactly which text goes into which elements. To manually tag text from the active QuarkXPress layout and copy it into the active XML document:

1 Scroll through the QuarkXPress layout to the page containing the text you want to tag, and then select that text using the **Content** tool E.

2 In the **XML Workspace** palette, scroll through the **XML Tree** list until you can see the element or attribute with which you want to tag the selected text. (If you need to create that element or attribute, see "Working with XML Document Content" in this chapter.)

Only attributes specified as CDATA attributes can be used to tag text. Fixed and empty attributes cannot be used to tag text.

3 Click and drag the selected text to the element or attribute name in the **XML Tree** list. The text is copied into the element or attribute. If the element or attribute already contains text, the text will be replaced.

You can only drag text to elements or attributes in the **XML Workspace** palette when the QuarkXPress **Drag and Drop Text** feature is on in the **Preferences** dialog box **Interactive** pane (**QuarkXPress** & **Preferences** on Mac OS or **Edit** & **Preferences** on Windows).

Although checking or unchecking **Drag and Drop Text** affects your ability to drag text onto elements and attributes in the **XML Workspace** palette, the avenue.quark drag-and-drop feature is not the same as the QuarkXPress **Drag and Drop Text** feature.

4 To verify that the content has been properly tagged, select the element in the **XML Tree** list; the text contained by the element or attribute displays in the **Content** field.

If you drag text to a NMTOKENS or IDREFS attribute, spaces in that text will be interpreted as delimiters between items in a list of NMTOKEN or IDREF values.

When you change text on a master page, it is considered a local change, so the change is not reflected on the layout page.

Spell checking a layout may change the positioning of XML markers in the text. This, in turn, can result in incorrectly tagged content. To avoid this problem, check spelling before you begin the tagging process.

TAGGING TEXT WITH RULE-BASED TAGGING

Rule-based tagging lets you automate part of the process of tagging text and pictures. To make use of rule-based tagging, you create a tagging rule set (see Chapter 5, “Tagging Rule Sets”), then use that tagging rule set to automate part of the tagging process.

To use rule-based tagging to tag text from the active QuarkXPress layout into the active XML document:

1 In the **XML Workspace** palette, choose a tagging rule set from the **Tagging Rule Set** pop-up menu.

2 Scroll through the **XML Tree** list until you can see the element where you want to begin rule-based tagging. For rule-based tagging to work, there must be at least one rule for this element or its children in the selected tagging rule set.

3 Select a range of text using the **Content** tool E or a text box using the **Item** tool e.

4 Press C (Mac OS) or Ctrl (Windows), then click and drag the selected text or text box to the element name in the **XML Tree** list. The text is copied into the XML document according to the rules in the selected tagging rule set. Avenue.quark adds new elements to the XML document as necessary to accommodate the tagged content. If an ambiguous tagging situation arises, avenue.quark displays a dialog box, asking you what you want to do.

5 To verify that the content has been properly tagged, select each element in the **XML Tree** list; the text contained by the element displays in the **Content** field.

6 To verify that avenue.quark has tagged all the text you want it to tag, choose **Show Tagged Content** from the **Utilities** menu. Tagged text and pictures display with the colors indicated in the **avenue.quark** pane of the **Preferences** dialog box (**QuarkXPress** & **Preferences** & **avenue.quark** on Mac OS or **Edit** & **Preferences** & **avenue.quark** on Windows).

It can take a considerable amount of time to process a long QuarkXPress layout with rule-based tagging, especially if the DTD (Document Type Definition) is deeply nested. A progress bar displays while the tagging process executes; as long as you see this progress bar, tagging is still underway.

If an “Out of memory –108” alert displays while you are tagging a layout, quit QuarkXPress immediately without saving the active XML document. (If you save the active XML document at this point, it may be saved in an invalid form).

If you receive an “Out of memory –108” alert, and you are using Mac OS, select the QuarkXPress application icon in the Finder. Choose **File** & **Get Info**; increase the memory allocation in the **Preferred Size** field. Relaunch QuarkXPress and try the tagging operation again.

Text in symbol fonts such as Zapf Dingbats and Wingdings is encoded in the same way as text in standard fonts such as Arial and Helvetica. This can lead to unexpected results. Therefore, we recommend that you avoid tagging text that uses a symbol font.

Spell checking a layout may change the positioning of XML markers in the text. This, in turn, can result in incorrectly tagged content. To avoid this problem, check spelling before you begin the tagging process.

When you use avenue.quark to tag the content of a QuarkXPress layout as XML, avenue.quark stores information in the XML document indicating which content in the QuarkXPress layout has been tagged. If you make changes to the QuarkXPress layout while the XML document is closed, avenue.quark can’t ensure that the XML content will match its corresponding QuarkXPress content. To keep an XML document and a QuarkXPress layout synchronized, always save the QuarkXPress project before you save the XML document. If you use the **Revert to Saved** feature with one of the files, use it with both of them.

TAGGING SELECTED TEXT

1. Select the text.
2. Click and begin to drag the selected text to the **XML Workspace** palette.
3. Press C (Mac OS) or Ctrl (Windows).
4. Finish dragging the text to the target element or attribute, and then release the mouse button.

EDITING TAGGED TEXT

Once you have tagged text in a QuarkXPress layout, you can edit that text in the QuarkXPress layout and it will automatically be updated in the **XML**

Workspace palette. If the XML document is not open at the time, the text is updated the next time both files are open.

Automatic content updating works only with elements; not with attributes.

If you want to break the link between a QuarkXPress layout and an element in the active QuarkXPress layout, select the element or attribute in the **XML Tree** list and then click the **Break Dynamic Link** button .

TAGGING PICTURES

To tag pictures from the active QuarkXPress layout and copy their file names into the active XML document:

1 Scroll through the QuarkXPress layout to the page containing the picture you want to tag, and then select its picture box.

2 In the **XML Workspace** palette, scroll through the **XML Tree** list until you can see the element or attribute with which you want to tag the selected picture. To create an element or attribute, see "Working with XML Document Content" in this chapter.

Only attributes specified as CDATA attributes may be used to tag pictures. Fixed and empty attributes cannot be used to tag pictures.

3 Press C (Mac OS) or Ctrl (Windows); then click and drag the selected picture to the element or attribute name in the **XML Tree** list. The name of the picture is copied into the selected element or attribute.

If the target element or attribute already contains a name, the new name always replaces the old one.

If a picture has been pasted into a picture box, rather than imported by choosing **File** & **Get Picture**, it cannot currently be tagged by avenue.quark.

MANUALLY ENTERING NEW CONTENT

In addition to copying content from a QuarkXPress layout, avenue.quark lets you add content by entering it directly into the XML document. To add new content to an empty element, attribute, or comment in the active XML document:

1 In the **XML Tree** list, select the element, attribute, or comment to which you want to add content.

Attributes may contain manually-entered content only if they are specified as CDATA attributes. Fixed and empty attributes cannot contain manually entered content.

2 Enter the content in the **Content** field. You can also paste content from the clipboard into the **Content** field. Note that such text will lose any formatting it might have, and will be pasted as plain ASCII text.

3 To indicate that you're finished editing the selected element, press the Tab key.

You can manually add content only to elements that do not contain content from a QuarkXPress layout. If an element contains content from a QuarkXPress layout, edit that content in the QuarkXPress layout; the copy of it in the XML document will be automatically updated.

PREVIEWING TAGGED TEXT

*Once your content has been tagged, you can preview how it will look when saved as XML. To preview the XML document displayed in the active **XML Workspace** palette, click the **Preview XML** button; the **Preview XML** dialog box displays.*

The Preview XML dialog box lets you preview the XML that will be created when you save the active XML document.

You can copy text from the **Preview XML** dialog box, but you cannot edit or delete it.

Although upper-ASCII characters (characters above ASCII 127) display unaltered in the **Preview XML** dialog box, such characters are converted to the appropriate codes at export, depending on the encoding method you choose in the **Save As** dialog box.

Appendices

APPENDIX A: XML QUICK REFERENCE

This section provides a review of XML features and conventions for quick reference.

THE PARTS OF AN XML DOCUMENT

An XML document consists of the following parts, in this order:

1. An XML declaration (optional, but highly recommended).
2. A DOCTYPE declaration and DTD (optional), including comments, processing instructions, and entity references.
3. XML elements (and their attributes), comments, processing instructions, and entity references.

XML DECLARATION

The XML declaration, if included, must be the first line in a XML document. It indicates the version of XML that the document adheres to, and whether the file includes any references to other files. For example:

```
<?xml version="1.0" standalone="no"?>
```

DOCTYPE DECLARATION (INCLUDING DTD)

The DOCTYPE declaration — which specifies the document's DTD — goes after the XML declaration and before the opening tag of the root element. There are two potential parts to any DTD: The external subset and the internal subset. If a document has only an external subset, it looks like this:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE rootElement SYSTEM "URL of DTD">
<rootElement>Content goes here.</rootElement>
```

If a document has only an internal subset, it looks like this:

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE rootELement [
  <!-- DTD goes here -->
]>
<rootElement>Content goes here.</rootElement>
```

If a document has both an external subset and an internal subset, it looks like this:

```
<?xml version="1.0" standalone="no"?>
<!-- External DTD -->
<!DOCTYPE rootElement SYSTEM "URL of DTD" [
  <!– Internal DTD goes here -->
]>
<rootElement>Content goes here.</rootElement>
```

ELEMENTS

An element consists of an opening tag (<tagName>), some content, and a closing tag (</tagName>):

```
<tagName>Content goes here.</tagName>
```

An exception is the empty tag, which may be a single tag with a forward slash before the closing >:

```
<emptyTag/>
```

All elements must be properly nested, meaning that the most recently opened tag must be closed before you can close any other tags. For example, the following line would be illegal in an XML document because it does not close <tag2> before closing <tag1>:

```
<tag1><tag2>Content goes here.</tag1></tag2>
```

Each XML document must have a root element that contains all the other elements in the document.

Element names are case-sensitive. Each element name must begin with a letter or an underscore (_); subsequent characters in the name can be letters, underscores, numbers, hyphens and periods, but not spaces or tabs.

ATTRIBUTES

Elements may have attributes as part of their opening tag (or, for empty elements, as part of the single opening/closing tag). An attribute consists of an attribute name followed by an equals sign and then an attribute value in quotation marks. For example:

```
<elementName attributeName="attributeValue">Content</elementName>
<elementName attributeName="attributeValue" />
```

COMMENTS

A comment consists of text between a <!– and a –>. The content of comments should be ignored by XML processors. Comments cannot contain “--” and they cannot contain other comments.

```
<!— This is a comment. Characters such as < and > are legal here. -->
```

PROCESSING INSTRUCTIONS

A processing instruction consists of text between a `<?` and a `?>`. Processing instructions are read *only* by XML processors and cannot contain content. The syntax for processing instructions is as follows:

```
<?target instruction?>
```

CHARACTER REFERENCES

A character reference is way of representing Unicode characters in parsed character data. The syntax for character references is as follows:

```
&#UnicodeValueOfCharacter;
```

ENTITY REFERENCES

An entity reference is a name that represents a specific character, text string, or file. Entity references in an XML document are always between an ampersand (&) and semicolon (;). For example, `>` represents a greater-than sign (<), which cannot be included in XML content except as an entity reference.

The meaning of each entity reference used in an XML document must be defined in the document's DTD, with the exception of the following predefined character entity references, which can be used without being defined:

CHARACTER	ENTITY REFERENCE
<	`<`
>	`>`
&	`&`
"	`"`
'	`'`

WELL-FORMED XML

To be well-formed, an XML document must follow these rules:

- The first line should be an XML declaration.
- There must be an end tag for every start tag (except for single empty tags).
- Single empty tags must end with `/>` .
- There must be a root element that contains all other elements.
- All elements must be properly nested, meaning that the most recently opened tag must be closed before you can close any other tags.
- All attribute values must be enclosed in quotation marks (`"`).
- All tags must begin with `<` and all entities must begin with `&`.
- The only entity references that may be used — unless the document has a DTD — are the predefined character entity references listed above.

VALID XML

A valid XML document is an XML document that is well-formed and adheres to the DTD specified by its `DOCTYPE` declaration.

APPENDIX B: DTD QUICK REFERENCE

This section provides a review of DTD features and conventions for quick reference.

THE PARTS OF A DTD

A DTD may be composed of the following parts, in no particular order:

- Element type declarations
- Attribute declarations
- Comments
- Entity reference declarations
- Notation declarations
- Processing instructions
- Parsed entity references
- Conditional sections

ELEMENT TYPE DECLARATIONS

The syntax for an element type definition is as follows:

```
<!ELEMENT elementName (elementContent)>
```

Element names are case-sensitive. Each element name must begin with a letter or an underscore (_); subsequent characters in the name can be letters, underscores, numbers, hyphens, and periods, but not spaces or tabs.

Element content may consist of parsed character data (that is, text and entity references, expressed as `#PCDATA`) and/or other element types. The following symbols can be inserted after any element name or closing parenthesis in the element content definition:

SYMBOL	MEANING
None	Exactly one
+	One or more
*	Zero or more
?	Zero or one

To require one element to be followed by another, use a comma:

```
<!ELEMENT elementName (element1, element2)>
```

To indicate that content can include one element *or* another, use a |:

```
<!ELEMENT elementName (element1 | element2)>
```

To allow an element to contain a combination of specific elements and `#PCDATA` in any order, use the following syntax:

```
<!ELEMENT elementName (#PCDATA | element1 | element2)*>
```

To allow an element to contain any combination of elements and `#PCDATA` in any order, use the following syntax (note omission of parentheses):

```
<!ELEMENT elementName ANY>
```

To define an empty element, use the following syntax (note omission of parentheses):

```
<!ELEMENT elementName EMPTY>
```

ATTRIBUTE DECLARATIONS

The syntax for a single attribute definition is as follows:

```
<!ATTLIST elementName attributeName attributeType defaultValue>
```

Attribute names are case-sensitive. Each attribute name must begin with a letter or an underscore (_); subsequent characters in the name can be letters, underscores, numbers, hyphens, and periods, but not spaces or tabs.

Attribute types may be as follows:

ATTRIBUTE TYPE	MEANING
CDATA	Character data and entity references, between quotation marks (“”)
ID	Must contain a unique name* for each element of this type
IDREF	The unique ID name* of an element in the XML file
ENTITY	An unparsed external entity reference name* defined in the DTD
ENTITIES	A list of ENTITY names, separated by spaces
Enumerated	A list of names*, separated by | characters, in parentheses
NMTOKEN	A value containing only NameChar characters**
NMTOKENS	A list of NMTOKEN values, separated by spaces
NOTATION	The name of a notation defined in the DTD
Enumerated NOTATION	A list of NOTATION values, separated by | characters, in parentheses

**Names must begin with a letter or an underscore (_); subsequent characters in the name can be letters, underscores, numbers, hyphens, and periods, but not spaces or tabs.*

***NameChar characters include letters, underscores, numbers, hyphens, or periods, but not spaces or tabs.*

Default attribute values may be as follows:

ATTRIBUTE TYPE	MEANING
#REQUIRED	This attribute must be specified by the element
#IMPLIED	This attribute may or may not be used
#FIXED value	If not specified, this attribute is assumed to be value; if specified, it must be value
defaultValue	If not specified, this attribute is assumed to be defaultValue

COMMENTS

A comment consists of text between a <!— and a —>. The content of comments should be ignored by XML processors. Comments cannot contain "--" and they cannot contain other comments.

```
<!— This is a comment. Characters such as < and > are legal here. -->
```

CHARACTER REFERENCES

A character reference is way of representing Unicode characters in parsed character data. The syntax for character references is as follows:

```
&#UnicodeValueOfCharacter;
```

ENTITY REFERENCE DECLARATIONS

There are five types of entities. The syntax for their declaration is as follows:

TYPE	SYNTAX
Parsed internal	<!ENTITY entityName "text of entity">
Parsed external	<!ENTITY entityName SYSTEM "URL of file"> — **OR** — <!ENTITY entityName PUBLIC "name of file" "URL of file">
Unparsed external	<!ENTITY entityName SYSTEM "URL of file" NDATA notationName> — **OR** — <!ENTITY entityName PUBLIC "name of file" "URL of file NDATA notationName>
Internal parameter	<!ENTITY %entityName "text of entity">
External parameter	<!ENTITY %entityName SYSTEM "URL of file"> — **OR** — <!ENTITY %entityName PUBLIC "name of file" "URL of file">

The syntax for using the first three types of entity reference is &entityName;. The syntax for using a parameter entity is %entityName;. Parameter entity references are always parsed and can be used *only* in a DTD.

NOTATION DECLARATIONS

Notation declarations should be specified in one of the two following ways:

```
<!NOTATION notationName SYSTEM "External Identifier">
<!NOTATION notationName PUBLIC "External Identifier Name" "Backup URL">
```

The external identifier should be the name of an application that can process or display files to which this notation is applied. For example:

```
<!NOTATION gif SYSTEM "Microsoft Internet Explorer">
```

Note that it is up to the application that processes the XML to pass the URL to the application indicated by the external identifier.

PROCESSING INSTRUCTIONS

A processing instruction consists of text between a `<?` and a `?>`. Processing instructions are read *only* by XML processors and cannot contain content. The syntax for processing instructions is as follows:

```
<?target instruction?>
```

APPENDIX C: UNDERSTANDING ENCODINGS

Let's say you've just exported an XML file from avenue.quark, and when you look at it in your text editor, you see a lowercase "a" with an accent where you thought you had a trademark symbol. In fact, a lot of your special symbols are incorrect. What happened?

Your text editor probably doesn't support the encoding used by your XML file. This section explains the topic in detail.

WHAT IS AN ENCODING?

An encoding is specification that maps a set of characters to corresponding numeric values. For example, the ASCII encoding maps the character "M" to the numeric value 77, "N" to 78, "O" to 79, and so forth.

A text file's encoding allows a program to translate the text file into the proper characters on the screen. Without the encoding, a text file is just a stream of numbers. If you view a text file using the wrong encoding, you're likely to see rows of strange characters, because the application opening the file will map the numeric values to the wrong set of characters.

All of the following are encodings:

- ASCII
- MacRoman (used by Mac OS)
- Windows Latin 1 (used by Windows)
- UTF-8
- UTF-16 (Unicode)
- Shift-JIS

Avenue.quark supports the UTF-8 and UTF-16 encodings.

LOWER AND UPPER CHARACTER RANGES

You can divide most encodings into two parts: the first 128 characters (the lower range), and all the characters after that (the upper range).

Generally speaking, the lower range of most encodings is mapped to the same characters. This range includes the characters a–z, A–Z, 0–9, a handful of punctuation characters, plus some special control characters.

Characters in the upper range can create problems. For example, MacRoman and Windows Latin 1 have lower ranges that are nearly identical. So if you take a file that uses only characters from this range and transfer that file from Mac OS to Windows, it looks fine, but if the file contains *upper*-range characters, you might get some strange results, because many of the upper-range values are mapped to different characters on each platform. For example, a character that shows up as a trademark symbol in Mac OS might show up as a superscript lowercase "a" in Windows.

When you get such incorrect character displays, it's either because the application displaying the text doesn't know the encoding of that text, or because the application isn't capable of correctly displaying text with the file's specified encoding.

SPECIFYING ENCODINGS

You can indicate the encoding of an XML file by including an encoding specification in the file's XML declaration, like this:

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

If an XML file doesn't contain an encoding specification, avenue.quark assumes that the file uses the UTF-8 encoding.

When you save an XML file from avenue.quark, you specify the document's encoding using the **Encoding** pop-up menu, and avenue.quark automatically generates the appropriate encoding attribute.

ENCODINGS AND DTDS

XML lets you specify the encoding of an XML file. However, it doesn't provide a way to specify the encoding of a free-standing DTD file.

Fortunately, avenue.quark does. To specify the encoding of a free-standing DTD, just add the following text as the first line in the file:

```
<? xml encoding="encodingName" ?>
```

For example, to specify a free-standing DTD as a UTF-16 DTD, just add the following line to the beginning of the file:

```
<? xml encoding="UTF-16" ?>
```

APPENDIX D: SAMPLE AVENUE.QUARK SCENARIO

Avenue.quark lets you use a DTD to extract structured content from QuarkXPress layouts and store that content in the file system or in a database. The following section describes how the process works using a sample situation.

THE SITUATION

Let's say your organization has created a large number of articles in QuarkXPress format, and you'd like to export the content in XML format and store it in a database so you can make it available to your customers on the Web. The articles all use the same QuarkXPress template and style sheets.

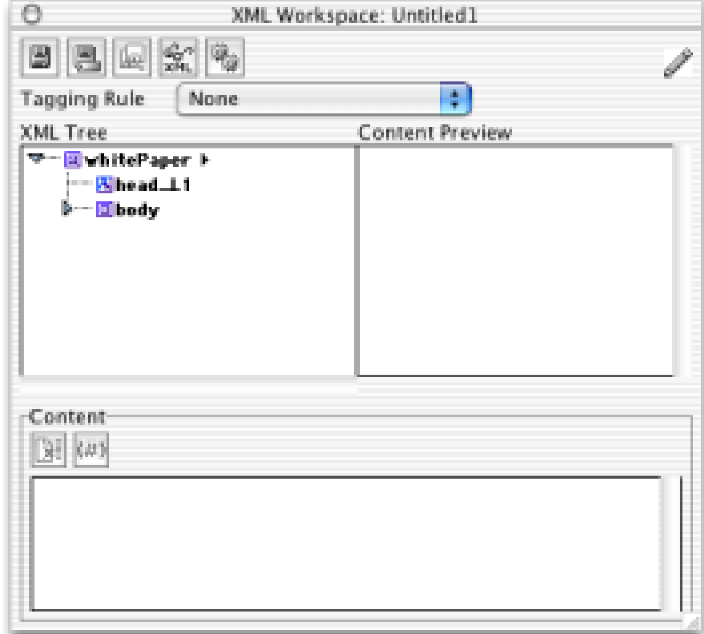
1 Create or choose a DTD.

Before you can extract the articles' content in a structured format, you must have a structure to contain that content. The DTD provides that structure. There are two ways to acquire a DTD for use with avenue.quark:

- Choose an existing XML DTD. There are several industry-standard XML DTDs available as of this writing, and as XML is more widely adopted, more will become available. If you choose to do this, give careful consideration to the DTD you want to use; an inappropriate DTD can make life difficult for the people who need to use it.
- Develop your own DTD. Read "Working With DTDs" in this chapter, then carefully analyze the parts that go into an article and develop a corresponding DTD. This process can be time-consuming, but it will pay off if the resulting DTD is appropriate to its use. Generally, it's an editorial or administrative group that selects a DTD.

2 Create an XML document.

Create a new XML document in avenue.quark and specify the DTD you chose in Step 1. Any mandatory elements in the DTD are automatically inserted in the XML document.



XML Workspace palette for a new XML document

3 Create a tagging rule set.

One of the unique features of avenue.quark is rule-based tagging. In rule-based tagging, you create a set of *tagging rules* that tell avenue.quark, for example, that a paragraph that uses the "Headline" style sheet should usually be tagged as a <Title>. You can also use tagging rule sets to specify how particular character style sheets, text colors, and local formatting styles should be tagged. (To create tagging rule sets, see Chapter 5, "Tagging Rule Sets.")

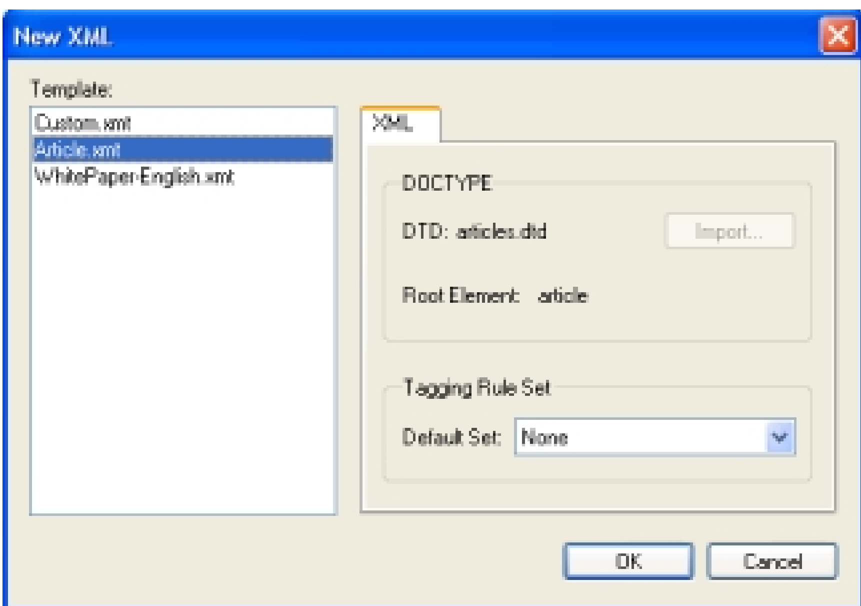
4 Save the XML document as a template.

Save the XML document as a template named "article.xmt." The template contains the technical document DTD and the tagging rule set you created in Step 3. You can use this template to create as many XML files as you want, on the same computer or on several computers.

5 Display the QuarkXPress layout you want to tag.

6 Create a new XML document based on the XML template.

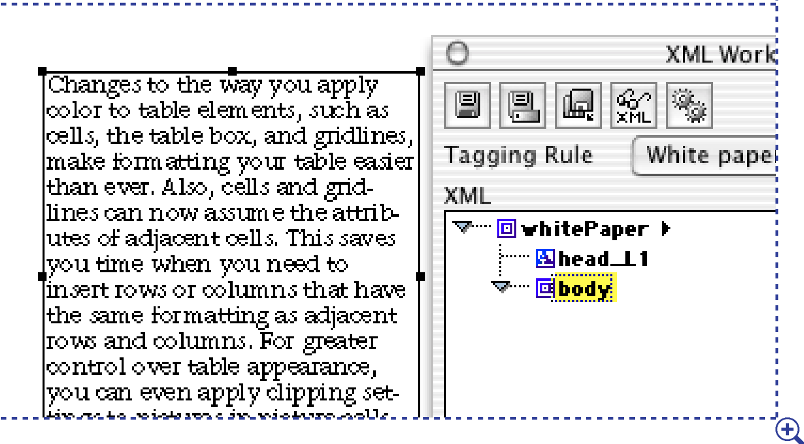
When you create a new avenue.quark XML document, the first thing you must do is choose a template from the **Template** list; the new XML document will be based on this template. For this example, use the "article.xmt" from Step 4.



This article.xmt template makes it easy to tag a QuarkXPress layout.

7 Perform rule-based tagging.

To perform rule-based tagging, C+drag (Mac OS) or Ctrl+drag (Windows) the box containing the article to the <body> element in the **XML Tree** list. Avenue.quark automatically tags the layout using the rules in the tagging rule set.



To use rule-based tagging, C+drag (Mac OS) or Ctrl+drag (Windows) the box to the appropriate element in the XML Tree list. Avenue.quark uses the tagging rule set to tag as much of the content as it can.

8 Perform any necessary manual tagging.

Some of your layouts may be ready after rule-based tagging has been completed. Others may have additional content that needs to be tagged manually, or occurrences of content that could be tagged in more than one way. To resolve such situations, drag the content in question onto the appropriate element in the **XML Workspace** palette. (You can only drag text to elements or attributes in the **XML Workspace** palette when the QuarkXPress **Drag and Drop Text** feature is on [**QuarkXPress** & **Preferences** & **Interactive** pane on Mac OS or **Edit** & **Preferences** & **Interactive** pane on Windows].)

9 Use your structured content on the Web and elsewhere.

Once your content is in XML format, you can use a variety of tools to publish it on the Web. For example, you can serve it as straight XML and view it using a newer Web browser such as Microsoft® Internet Explorer 5.0. XML-tagged content can also be used in a wide variety of other ways, for everything from electronic information exchange to the generation of printed documents.