Integrating Office with the Rest of the World

Office 2003 XML

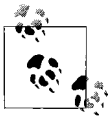
Evan Lenz, Mary McRae & Simon St.Laurent

O'REILLY®

CHAPTER 2

The WordprocessingML Vocabulary

Microsoft Office Word 2003 marks the introduction of XML as a native format for Word documents. Any Word document can now be opened in Word and saved as XML, thereby freeing documents from the tyranny of Word's proprietary *.doc* format. This new format, called WordprocessingML, opens up a multitude of possibilities for generating and processing Word documents. (Read Chapter 3 first if you want some immediate gratification regarding use cases for WordprocessingML.) This chapter includes a basic introduction to WordprocessingML, along with some general technical observations and guidelines for learning more. It is meant to complement, rather than replace, a detailed investigation of the WordprocessingML schema.



An authoritative and thorough source for learning is the Microsoft-supplied XSD schema for WordprocessingML. The "Microsoft Office 2003 XML Reference Schemas" package has been released under a royalty-free license and includes each of the WordprocessingML schema documents, as well as accompanying documentation. It can be found by starting at *http://www.microsoft.com/office/xml/*.

Introduction to WordprocessingML

WordprocessingML is Microsoft's XML format for Word documents. It's what you get when you select Save As... and choose "XML Document." WordprocessingML is a lossless format, which means that it contains all the information that Word needs to re-open a document, just as if it had been saved in the traditional *.doc* format—all text, formatting, styles, document metadata, images, macros, revision history, Smart Tags, etc. (The one exception is that WordprocessingML does not embed TrueType fonts, which is only a disadvantage if the users opening the document do not have the needed font installed on their system.) Indicative of Word's tremendous size and legacy, the WordprocessingML schema file approaches 7,000 lines in length. Fortunately, a little bit of knowledge about WordprocessingML can go a long way.

It was only recently that Microsoft began calling Word's XML format "WordprocessingML," whereas previously it was called, simply, "WordML" (as still reflected in the schema's namespace URI). Why they decided to adopt this new name isn't entirely clear...though it certainly is *wordier*.

To gain an advanced understanding of WordprocessingML, you'll need to first understand the fundamentals of Word itself. While this chapter briefly touches on Word's global architecture and design, books such as the following can provide a more solid foundation:

Word Pocket Guide, by Walter Glenn (O'Reilly)
Word 2000 in a Nutshell, by Walter Glenn (O'Reilly)

In this chapter, we'll examine several increasingly detailed examples of WordprocessingML. First, we'll take a look at the definitive "Hello, World" example for WordprocessingML. Next, after learning some tips for working with WordprocessingML, we'll take a tour through an example WordprocessingML document as output by Word. Then, we'll systematically cover Word's primary formatting constructs: runs, paragraphs, tables, lists, sections, etc. Finally, we'll take another look at one of Word's most important features: the style. Understanding how styles work—how they interact with direct formatting and how they relate to document templates—is essential to an overall understanding of WordprocessingML and Word in general.

A Simple Example

Example 2-1 shows a WordprocessingML document that one might create by hand in a plain text editor. This example represents the simplest non-empty WordprocessingML document possible.

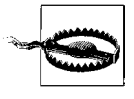
Example 2-1. A simple WordprocessingML document created by hand

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml">
  <w:body>
    <w:p>
      <w:r>
        <w:t>Hello, World!</w:t>
      </w:r>
    </w:p>
  </w:body>
</w:wordDocument>
```

The first thing to note about this example is the `mso-application` processing instruction (PI). This is a generic PI used by various applications within the Microsoft Office

System. Its purpose is to associate the given *.xml* file with a particular application in the Office suite. In this case, the file is associated with Microsoft Word. This has a double effect: not only is the Word application launched when a user double-clicks the file, but Windows Explorer renders the file using a special Word XML icon. This behavior is enabled through an Explorer shell that is automatically installed with Office 2003. All XML documents saved by Word will include this PI. We'll see more uses of the `mso-application` PI in Chapter 7 and Chapter 10.

As mentioned above, Example 2-1 shows the simplest non-empty WordprocessingML document possible. The `w:body` element is the only required child element of the `w:wordDocument` root element. It technically can be empty, but that would make for a pretty boring first example. The `w:p` element stands for "paragraph," `w:r` stands for "run," and `w:t` stands for "text." The namespace prefix `w` maps to the primary WordprocessingML namespace: `http://schemas.microsoft.com/office/word/2003/wordml`.



Beware the default namespace! Word, in its longstanding attempt to be everything to everybody, does something funny when you try to open a WordprocessingML document that uses a default namespace, rather than the `w` (or some other) prefix, for elements in the WordprocessingML namespace. It sees the naked (un-prefixed) `body` element and thinks "This must be HTML!" The easiest way to avoid this problem is to always use an XML declaration (e.g., `<?xml version="1.0"?>`) at the beginning of an XML document that will be opened by Word. Word will consistently recognize the document as XML if the XML declaration is present.

With few exceptions, all text in a given document is contained within a `w:t` element that's contained within a `w:r` element that's contained within a `w:p` element. A final thing to note is that, except for the `w:wordDocument` element, none of the elements in Example 2-1 (`w:body`, `w:p`, `w:r`, and `w:t`) can have attributes. As we'll see, properties are instead assigned (to paragraphs and runs) using child elements. Figure 2-1 shows the result of opening our example document in Word. We see "Hello, World!" in the default font and font size, in the default view. Word supplies these defaults, because they are not explicitly specified in our WordprocessingML document.

Tips for Learning WordprocessingML

Learning WordprocessingML—particularly how Word behaves when it encounters various markup constructs—is an iterative process. You go back and forth between the text editor and the Word application, closing the document in Word so you can make changes to it elsewhere, and then re-opening it to see what effects those changes have. You make hypotheses and you test them. Anything you can do to speed up the iterations of this process will help. Below are several pieces of advice to consider as you begin this educational journey.

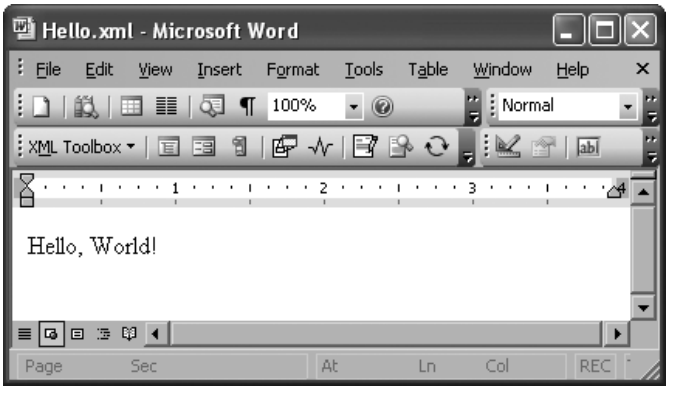


Figure 2-1. Our hand-edited WordprocessingML file, opened in Word

Experiment
: Since Microsoft has released fairly limited documentation of WordprocessingML so far, it is often best to learn through experimentation. Create a document in Word that uses various formatting features you are interested in. Save the document as XML. Then, investigate the WordprocessingML for the document, making note of how various document structures are represented as XML. Internet Explorer can be a good tool for viewing WordprocessingML documents. (See the sidebar "Using Internet Explorer to Inspect WordprocessingML Documents.")

Don't try to learn everything
: This tip offsets the first one. It is sometimes possible to get hung up on particular theoretical questions or problems when experimenting with WordprocessingML. But if you want to remain productive, you should be prepared to suspend understanding at various turns in your investigation. The beauty of WordprocessingML is that you can accomplish quite a lot without understanding everything in the markup. For example, to create a stylesheet that generates WordprocessingML documents, you would only need to prepare the document in Word itself, save it as XML, and then copy and paste the bulk of it into your stylesheet, zeroing in on only the elements that contain dynamic content.

Use the Reveal Formatting task pane
: Word's Reveal Formatting task pane (press Shift-F1) provides a very helpful intermediate view of formatting properties between the WordprocessingML itself and how the document actually looks. Moreover, if you check the "Distinguish style source" checkbox (at the bottom of the task pane), it will identify the source of specific formatting properties, distinguishing between those that are defined in a style and those that are applied as direct formatting. This chapter includes some example screen shots that use the Reveal Formatting task pane.

Use the XML Toolbox

The XML Toolbox was quietly released by Microsoft as a plug-in for Word. It is Word's equivalent of View Source, and it is a godsend. It lets you view the underlying WordprocessingML for a document or selection right from within Word. You can also manually insert WordprocessingML, using the "Insert XML" dialog, shown in Figure 2-2. Ultimately, it is not a substitute for saving as XML, as it leaves out some things (such as document metadata and spelling errors). One caveat is that the XML Toolbox plug-in requires .NET Programmability support. This means that the .NET Framework 1.1 must have been installed prior to the Office 2003 installation. Get and read about this plug-in at *http://msdn.microsoft.com/library/en-us/dnofftalk/html/odc_office01012004.asp*

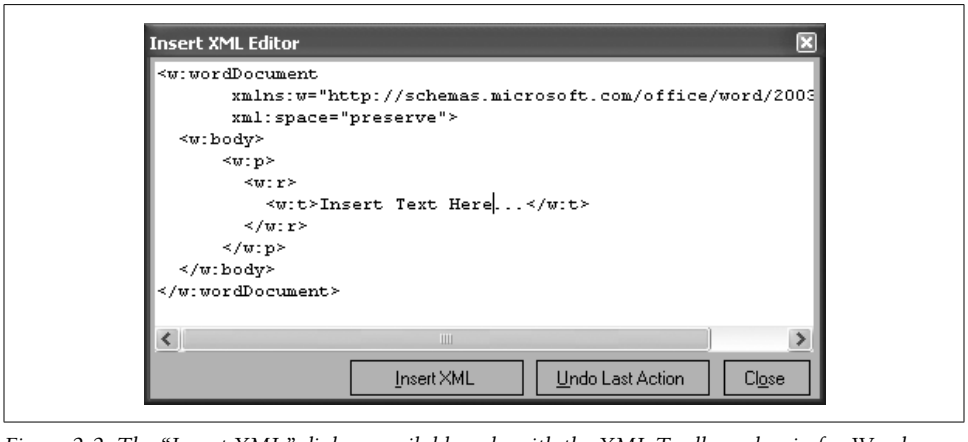


Figure 2-2. The "Insert XML" dialog, available only with the XML Toolbox plug-in for Word

WordprocessingML's Style of Markup

If you have any XML or HTML markup background, then WordprocessingML's style of markup may surprise you. WordprocessingML was *not* designed from a clean slate for the purpose of creating documents in XML markup. Instead, it is an unveiling of the internal structures that have been present in Microsoft Word for years. Though certain features have been added to make WordprocessingML usable outside the context of Word, by and large it represents a serialization of Word's internal data structures: various kinds of objects associated with myriad property values. Indeed, the object-oriented term "properties" permeates the WordprocessingML schema. If you want to make a run of text bold, you set the bold property. If you want to indent a particular paragraph, you set its indentation property. And so on.

No Mixed Content

Mixed content describes the presence of text content and elements inside the same parent element. It is standard fare in the world of markup, especially when using

Using Internet Explorer to Inspect WordprocessingML Documents

Internet Explorer's default tree-view stylesheet for XML documents provides a handy, readable way to investigate the structure of WordprocessingML documents. However, if you try opening a WordprocessingML document in IE (e.g., by right-clicking the file and selecting Open With → Internet Explorer), IE turns around and launches Word, because it too is now trained to recognize and honor the `mso-application` processing instruction. There are two techniques for getting around this.

The first technique is to simply remove the `mso-application` PI before opening the WordprocessingML document in IE:

1. Save the Word document as XML and then close it.
2. Open the newly saved WordprocessingML document in Notepad.
3. Delete or comment out the `mso-application` PI and re-save.

IE will now display the document using its pretty XML tree view, and will continue to do so even if the document is subsequently updated by Word to include the `mso-application` PI. Once you've initially opened it in IE, you can refresh IE to see how changes to the document from within Word affect the underlying WordprocessingML.

The second technique involves making a temporary global system change, obviating the need to comment out the `mso-application` PI for each and every document you want to inspect.

1. Open the Registry Editor by selecting Start → Run and typing `regedit`.
2. Find the sub-key named *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Office\11.0\Common\Filter\text/xml*.
3. Right-click the Word.Document string value entry, and select Rename.
4. Change the name to something like Word.DocumentDISABLED.

This will make it easy to restore the setting later, by simply renaming it again and removing the "DISABLED" part. With the WordprocessingML filter effectively disabled, IE will now open WordprocessingML documents using its default XML tree-view stylesheet just like any other XML document. Windows Explorer, however, will still continue to associate WordprocessingML documents with Word, which is probably what you will always want.

document-oriented markup. For example, in HTML, to make a sentence bold and only partially italicized, you would use code such as the following:

```
<b>This sentence has <i>mixed</i> formatting.</b>
```

WordprocessingML, however, never uses mixed content. All of the text in a WordprocessingML document resides in `w:t` elements, and `w:t` elements can only contain text (and no elements). The above sentence is represented much differently in

WordprocessingML. The hierarchy is flattened into a sequence of runs having different formatting properties:

```
<w:r>
  <w:rPr>
    <w:b/>
  </w:rPr>
  <w:t>This sentence has </w:t>
</w:r>
<w:r>
  <w:rPr>
    <w:b/>
    <w:i/>
  </w:rPr>
  <w:t>mixed</w:t>
</w:r>
<w:r>
  <w:rPr>
    <w:b/>
  </w:rPr>
  <w:t> formatting.</w:t>
</w:r>
```

As you can see, all of the text occurs by itself (no mixed content), within `w:t` elements.

Properties Are Set Using Empty Sub-Elements

The above snippet illustrates another general principle in WordprocessingML's style of markup: properties are assigned using empty sub-elements (e.g., `w:b` and `w:i` in the above example). For runs, the `w:rPr` element contains a set of empty elements, each of which sets a particular property on the run. Similarly, for paragraphs (`w:p` elements), the `w:pPr` element contains the paragraph formatting properties. For tables, table rows, and table cells, there are the `w:tblPr`, `w:trPr`, and `w:tcPr` elements, respectively. In each case, the `*Pr` element must come first, so that the general structure of paragraphs, runs, tables, table rows, and table cells looks like this:

```
Object
   Properties
   Content
```

The properties are defined first, and the content follows. If you have any experience with RTF (Rich Text Format), then this pattern may look familiar. Before the advent of WordprocessingML, RTF was the most open format in which Word was willing to save documents. A look at the same sentence after saving it as RTF is demonstrative:

```
{\b\insrsid3691043 This sentence has }
{\b\i\insrsid3691043 mixed}
{\b\insrsid3691043  formatting.}
```

The parallels should be fairly easy to draw, without understanding every detail. There are three runs (delineated by curly braces). The first run has bold turned on by virtue of the `\b` command. The second run has both bold and italic turned on by

virtue of the \b and \i commands. And the third run goes back to using just bold and no italic. From this perspective, WordprocessingML may look more like an XML format for RTF—an estimation that is not too far off the mark.

To learn more about RTF, consider the *RTF Pocket Guide* (O'Reilly), by Sean M. Burke.

No Hierarchical Document Structures

Nested markup describes the use of element nesting to arbitrary depths. In addition to formatting text, nested markup is useful for structuring documents. For example, a Docbook document may have sections and sub-sections nested to an arbitrary depth, like this:

```
<article>
  <section>
    <title>Section 1</title>
    <para>This is the first section.</para>
    <section>
      <title>Section 1A</title>
      <para>This is a sub-section.</para>
    </section>
  </section>
</article>
```

The above document is represented much differently in WordprocessingML. The hierarchy is flattened into a sequence of four paragraphs having different properties. Below is the w:body element, excerpted from such a document:

```
<w:body>
  <w:p>
    <w:pPr>
      <w:pStyle w:val="Heading1"/>
    </w:pPr>
    <w:r>
      <w:t>Section 1</w:t>
    </w:r>
  </w:p>
  <w:p>
    <w:r>
      <w:t>This is the first section.</w:t>
    </w:r>
  </w:p>
  <w:p>
    <w:pPr>
      <w:pStyle w:val="Heading2"/>
    </w:pPr>
    <w:r>
      <w:t>Section 1A</w:t>
    </w:r>
  </w:p>
```

```
    <w:p>
      <w:r>
        <w:t>This is a sub-section.</w:t>
      </w:r>
    </w:p>
  </w:body>
```

In Word, the paragraph is the basic block-oriented element, and paragraphs may not contain other paragraphs. Word does, however, provide a workaround for hierarchical documents, through use of the `wx:sub-section` element. In fact, if you were to open the above document and then save it from within Word, the result would include `wx:sub-section` elements that reflect the hierarchy intended by the heading paragraphs. We'll look at how this works in detail later, in "Outline Levels and Sub-Sections."

All Attributes Are Namespace-Qualified

One more peculiarity worth noting about WordprocessingML markup is its use of namespace-qualified attributes. In most XML vocabularies, attributes are not in a namespace. They are generally thought to "belong" to the element to which they are attached. As long as the element is in a namespace, then no naming ambiguities should arise. Namespace qualification, however, can be useful for "global attributes" that can be attached to different elements. Such attributes do not belong to any particular element. The `xml:space` attribute is a good example of a global attribute. XSLT also has some global attributes, such as the `xsl:exclude-result-prefixes` attribute, which can occur on any literal result element (in any namespace). These are considered good use cases for qualifying attributes with a namespace.

WordprocessingML, however, does not follow this convention. While there are some "global attributes" in WordprocessingML (such as the `w:type` attribute, which appears on the `aml:annotation` element, which we'll see), WordprocessingML does not restrict its use of namespace qualification to those cases. Instead, it universally qualifies all attributes across the board. For this reason, the key thing to remember when working with attributes in WordprocessingML is that they *always* must have a namespace prefix (because there's no such thing as a default namespace for attributes in XML).

A Simple Example Revisited

Example 2-2 shows how our "Hello, World" example looks after opening it in Word, selecting Save As…, and saving the file with a new name, *HelloSaved.xml*. For the sake of readability, we've added line breaks and indentation, neither of which affects the meaning of the file. The highlighted lines in this example correspond to the lines that were present in our original hand-edited WordprocessingML document in Example 2-1. Everything else is new.

Example 2-2. The same Word document, after Word saves it as XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml"
  xmlns:v="urn:schemas-microsoft-com:vml"
  xmlns:w10="urn:schemas-microsoft-com:office:word"
  xmlns:sl="http://schemas.microsoft.com/schemaLibrary/2003/core"
  xmlns:aml="http://schemas.microsoft.com/aml/2001/core"
  xmlns:wx="http://schemas.microsoft.com/office/word/2003/auxHint"
  xmlns:o="urn:schemas-microsoft-com:office:office"
  xmlns:dt="uuid:C2F41010-65B3-11d1-A29F-00AA00C14882"
  w:macrosPresent="no" w:embeddedObjPresent="no" w:ocxPresent="no"
  xml:space="preserve">
  <o:DocumentProperties>
    <o:Title>Hello, World</o:Title>
    <o:Author>Evan Lenz</o:Author>
    <o:LastAuthor>Evan Lenz</o:LastAuthor>
    <o:Revision>4</o:Revision>
    <o:TotalTime>15</o:TotalTime>
    <o:Created>2003-12-06T22:45:00Z</o:Created>
    <o:LastSaved>2003-12-18T07:59:00Z</o:LastSaved>
    <o:Pages>1</o:Pages>
    <o:Words>2</o:Words>
    <o:Characters>12</o:Characters>
    <o:Lines>1</o:Lines>
    <o:Paragraphs>1</o:Paragraphs>
    <o:CharactersWithSpaces>13</o:CharactersWithSpaces>
    <o:Version>11.5604</o:Version>
  </o:DocumentProperties>
  <w:fonts>
    <w:defaultFonts w:ascii="Times New Roman" w:fareast="Times New Roman"
                    w:h-ansi="Times New Roman" w:cs="Times New Roman"/>
  </w:fonts>
  <w:styles>
    <w:versionOfBuiltInStylenames w:val="4"/>
    <w:latentStyles w:defLockedState="off" w:latentStyleCount="156"/>
    <w:style w:type="paragraph" w:default="on" w:styleId="Normal">
      <w:name w:val="Normal"/>
      <w:rsid w:val="00B15979"/>
      <w:rPr>
        <wx:font wx:val="Times New Roman"/>
        <w:sz w:val="24"/>
        <w:sz-cs w:val="24"/>
        <w:lang w:val="EN-US" w:fareast="EN-US" w:bidi="AR-SA"/>
      </w:rPr>
    </w:style>
    <w:style w:type="character" w:default="on"
             w:styleId="DefaultParagraphFont">
      <w:name w:val="Default Paragraph Font"/>
      <w:semiHidden/>
    </w:style>
    <w:style w:type="table" w:default="on" w:styleId="TableNormal">
      <w:name w:val="Normal Table"/>
```

Example 2-2. The same Word document, after Word saves it as XML (continued)

```
      <wx:uiName wx:val="Table Normal"/>
      <w:semiHidden/>
      <w:rPr>
        <wx:font wx:val="Times New Roman"/>
      </w:rPr>
      <w:tblPr>
        <w:tblInd w:w="0" w:type="dxa"/>
        <w:tblCellMar>
          <w:top w:w="0" w:type="dxa"/>
          <w:left w:w="108" w:type="dxa"/>
          <w:bottom w:w="0" w:type="dxa"/>
          <w:right w:w="108" w:type="dxa"/>
        </w:tblCellMar>
      </w:tblPr>
    </w:style>
    <w:style w:type="list" w:default="on" w:styleId="NoList">
      <w:name w:val="No List"/>
      <w:semiHidden/>
    </w:style>
  </w:styles>
  <w:docPr>
    <w:view w:val="web"/>
    <w:zoom w:percent="100"/>
    <w:proofState w:spelling="clean" w:grammar="clean"/>
    <w:attachedTemplate w:val=""/>
    <w:defaultTabStop w:val="720"/>
    <w:characterSpacingControl w:val="DontCompress"/>
    <w:validateAgainstSchema/>
    <w:saveInvalidXML w:val="off"/>
    <w:ignoreMixedContent w:val="off"/>
    <w:alwaysShowPlaceholderText w:val="off"/>
    <w:compat/>
  </w:docPr>
  <w:body>
    <wx:sect>
      <w:p>
        <w:r>
          <w:t>Hello, World!</w:t>
        </w:r>
      </w:p>
      <w:sectPr>
        <w:pgSz w:w="12240" w:h="15840"/>
        <w:pgMar w:top="1440" w:right="1800" w:bottom="1440" w:left="1800"
                 w:header="720" w:footer="720" w:gutter="0"/>
        <w:cols w:space="720"/>
        <w:docGrid w:line-pitch="360"/>
      </w:sectPr>
    </wx:sect>
  </w:body>
</w:wordDocument>
```

The first thing that may come to mind when looking at this example is "Why does the XML contain so much more information when all I did was save it?" Or perhaps you've begun to panic.

Don't. While all of this XML is certainly daunting at first glance, we'll see that for the most part its meaning is straightforward. Take comfort in the fact that, while Word may create markup that's quite verbose, it can handle markup that minimally conforms to its schema without complaining at all. This liberality in what Word accepts makes it much easier to write applications that generate WordprocessingML.

Let's take a tour through this document, examining each top-level element in turn. Getting an overall, top-down view of what goes into a WordprocessingML document will help bring context to the more nitty-gritty, bottom-up examination of the vocabulary that will follow later in this chapter.

The w:wordDocument Element

The root element of Example 2-2, `w:wordDocument`, has a large number of attributes:

```
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml"
  xmlns:v="urn:schemas-microsoft-com:vml"
  xmlns:w10="urn:schemas-microsoft-com:office:word"
  xmlns:sl="http://schemas.microsoft.com/schemaLibrary/2003/core"
  xmlns:aml="http://schemas.microsoft.com/aml/2001/core"
  xmlns:wx="http://schemas.microsoft.com/office/word/2003/auxHint"
  xmlns:o="urn:schemas-microsoft-com:office:office"
  xmlns:dt="uuid:C2F41010-65B3-11d1-A29F-00AA00C14882"
  w:macrosPresent="no" w:embeddedObjPresent="no"
  w:ocxPresent="no" xml:space="preserve">
```

Actually, most of these are technically namespace declarations. They are present on every WordprocessingML document that Word outputs, regardless of whether all the namespaces are actually used in the document. In WordprocessingML, you can safely leave out all the namespace declarations except the ones you actually use, which will minimally include the primary WordprocessingML namespace (normally mapped to the `w` prefix). Below is a list of the namespaces declared in this document, along with a brief description of the purpose of each.

`http://schemas.microsoft.com/office/word/2003/wordml`
: Mapped to the `w` prefix. All of the core WordprocessingML elements and attributes are in this namespace.

`urn:schemas-microsoft-com:vml`
: Mapped to the `v` prefix. Elements in this namespace represent embedded Vector Markup Language (VML) images.

`urn:schemas-microsoft-com:office:word`
: Mapped to the `w10` prefix. This namespace is used for legacy elements from Word Ten. It is used in HTML output.

`http://schemas.microsoft.com/schemaLibrary/2003/core`
: Mapped to the `sl` prefix. The `sl:schema` and `sl:schemaLibrary` elements are used with Word's custom XML schema functionality, and are introduced in Chapter 4.

`http://schemas.microsoft.com/aml/2001/core`
: Mapped to the `aml` prefix. The Annotation Markup Language (AML) elements are used to describe tracked changes, comments, and bookmarks.

`http://schemas.microsoft.com/office/word/2003/auxHint`
: Mapped to the `wx` prefix. Elements in this namespace provide "auxiliary hints" for processing WordprocessingML documents outside of Word. They represent derivative information that is useful to us but that is of no internal use to Word. See "Auxiliary Hints in WordprocessingML," later in this chapter.

`urn:schemas-microsoft-com:office:office`
: Mapped to the `o` namespace. This is the namespace for "shared" document properties and custom document properties. They are shared in that they also apply to other Office applications, such as Excel.

`uuid:C2F41010-65B3-11d1-A29F-00AA00C14882`
: Mapped to the `dt` prefix. This is the XML Data Reduced (XDR) namespace, which, in WordprocessingML, qualifies the `dt` (data type) attributes of a document's custom document property elements.

While some confusing legacy is evident in this list, the overall distinction between namespaces is helpful, particularly between the `wx` and `w` namespaces, as we'll see.

The `xml:space` attribute is set to `preserve`, in order that whitespace characters (and even any instances of the empty `w:tab` element) are interpreted correctly. As a matter of best practice, you should include `xml:space="preserve"` on the root element of any WordprocessingML document you create.

The remaining three attributes of the `w:wordDocument` element are all optional and default to the value `no`.

```
w:macrosPresent="no" w:embeddedObjPresent="no" w:ocxPresent="no"
```

These are consistency checks for when certain kinds of base64-encoded binary objects are embedded in the document. Specifically, `w:macrosPresent` must be set to `yes` when the `w:docSuppData` element is present (containing toolbar customizations, VBA macros, etc.); `w:embeddedObjPresent` must be set to `yes` when the `w:docOleData` element is present (containing OLE objects from other applications, such as Excel); and `w:ocxPresent` must be set to `yes` when a `w:ocx` element is present somewhere in the body of the document (representing a control from Word's Control Toolbox). Unless your document contains any such objects, you can safely leave out these attributes.

The child elements of `w:wordDocument`, as included in this example, represent only a portion of the root element's complete content model. Below is a list of all possible

child elements in the order they are supposed to occur, according to the WordprocessingML schema. Word tends to be lenient about WordprocessingML documents that contain these elements in a different order, which suggests it does not validate documents against the published schema when they are loaded. However, to be on the safe side, you should ensure that these elements are in the correct order in WordprocessingML documents that you create. As mentioned before, `w:body` is the only required child element of `w:wordDocument`. Only the highlighted elements in this list are actually present in Example 2-2.

```
w:ignoreSubtree
w:ignoreElements
o:SmartTagType
o:DocumentProperties
o:CustomDocumentProperties
sl:schemaLibrary
w:fonts
w:frameset
w:lists
w:styles
w:divs
w:docOleData
w:docSuppData
w:shapeDefaults
w:bgPict
w:docPr
w:body
```

Apart from the highlighted elements, the `w:lists` element is the only one in the above list that will receive further coverage in this chapter.

The o:DocumentProperties Element

The `o:DocumentProperties` element in Example 2-2, shown again below, is in the general Office namespace (mapped to the `o` prefix), because it includes properties, such as metadata and statistics, that are common to both Word and Excel:

```
<o:DocumentProperties>
  <o:Title>Hello, World</o:Title>
  <o:Author>Evan Lenz</o:Author>
  <o:LastAuthor>Evan Lenz</o:LastAuthor>
  <o:Revision>4</o:Revision>
  <o:TotalTime>15</o:TotalTime>
  <o:Created>2003-12-06T22:45:00Z</o:Created>
  <o:LastSaved>2003-12-18T07:59:00Z</o:LastSaved>
  <o:Pages>1</o:Pages>
  <o:Words>2</o:Words>
  <o:Characters>12</o:Characters>
```

```
    <o:Lines>1</o:Lines>
    <o:Paragraphs>1</o:Paragraphs>
    <o:CharactersWithSpaces>13</o:CharactersWithSpaces>
    <o:Version>11.5604</o:Version>
  </o:DocumentProperties>
```

These elements are also serialized as such when Word saves a document as HTML. They correspond primarily to the properties you see when you open the document Properties dialog (by selecting File → Properties). Figure 2-3 shows the Statistics tab of the file Properties dialog.

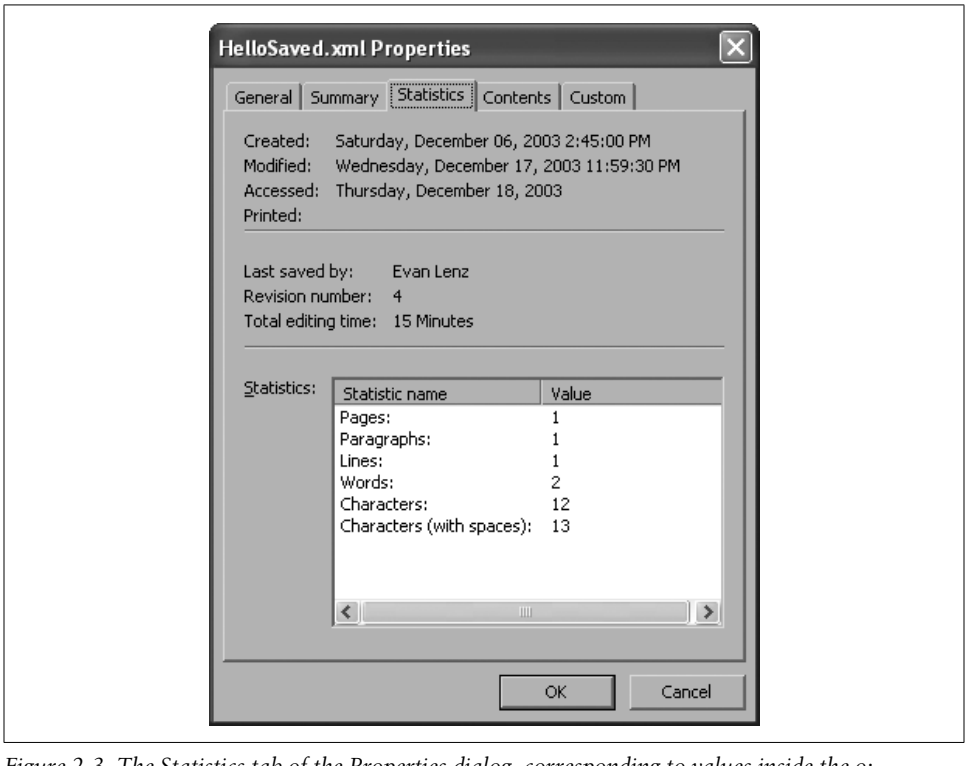


Figure 2-3. The Statistics tab of the Properties dialog, corresponding to values inside the o:DocumentProperties element

There are 12 more valid child elements of `o:DocumentProperties` not shown here, making a total of 26. A number of these can be added to a document from within Word, at user option. For example, there is an element corresponding to each of the fields in the Summary tab of the file Properties dialog, shown in Figure 2-4.

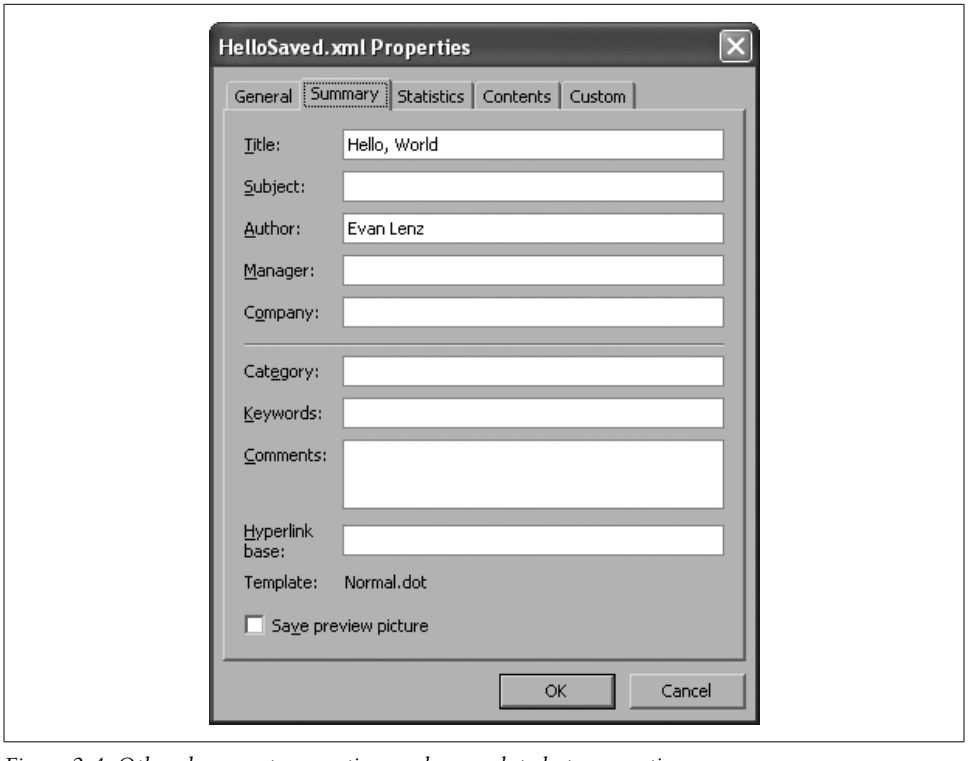


Figure 2-4. Other document properties can be populated at user option

The w:fonts Element

The `w:defaultFonts` element inside the `w:fonts` element specifies the *default font* for a document.

```
<w:fonts>
  <w:defaultFonts w:ascii="Times New Roman" w:fareast="Times New Roman"
                  w:h-ansi="Times New Roman" w:cs="Times New Roman"/>
</w:fonts>
```

A document's default font is applied to all of the document's paragraph styles that do not explicitly specify a font. Normally, when you create a new blank document in Word, the default font setting as specified in the *Normal.dot* document template is copied into the document. But our hand-coded WordprocessingML document (Example 2-1) isn't "normal" in this sense. It was created outside of Word and contains no default font definition at all. Word gracefully handles this scenario when it loads the document by automatically inserting a default font, as shown in Example 2-2. Times New Roman is thus the "default default" font. In fact, Times New Roman is also the default font assigned to the *Normal.dot* template when Word is first installed, or when it is forced to create a new *Normal.dot* template because someone deleted the *Normal.dot* file.

The attributes on the w:defaultFonts element indicate which font should be used for each character encoding range among ASCII, high ANSI, complex scripts, and East Asian characters. In Example 2-2, Times New Roman is the default font for all of these ranges.

The w:fonts element may also contain zero or more w:font elements (zero in the case of Example 2-2) following the w:defaultFonts element. The w:font elements are optional; you don't need to include a corresponding w:font element just to use a particular font. The only purpose of this element is to provide Word with descriptive information about a font (using its seven possible child elements) that could be useful in the event that the font is not available on a user's machine. In that case, Word can choose a reasonable alternative based on the information about the font provided in the document.

The w:styles Element

The w:styles element includes definitions of all of a document's styles. Before looking at the WordprocessingML syntax for defining styles, let's establish some basic terminology. A *style* is a group of formatting properties that can be applied as a unit. There are four possible *style types* in Word:

```
paragraph
character
table
list
```

These style types apply respectively to paragraphs, runs, tables, and lists. Every paragraph, run, table, and list in a Word document is necessarily associated with a style of the corresponding type. If a paragraph, run, table, or list in a WordprocessingML document doesn't explicitly specify an associated style (as is the case in Example 2-2), then it takes on the document's *default style* of the appropriate style type. Thus, styles are always involved, regardless of whether you specifically make use of them.

Normally, when you create a new blank document in Word, all of the styles defined in the *Normal.dot* document template are copied into the document. These include, at minimum, a default style definition for each style type. However, our hand-coded WordprocessingML document does not include the w:styles element. Just as Word automatically creates the w:fonts element when absent, Word automatically inserts four w:style elements, corresponding respectively to the four style types (paragraph, character, table, and list):

```
Normal
Default Paragraph Font
Normal Table
No List
```

These four Word-defined styles are what we see inside the w:styles element in Example 2-2. Effectively, they are implicitly present in any WordprocessingML document that does not explicitly define them. (However, to explicitly refer to them from within the body of the document, they must also be explicitly present in the document's w:styles element.) These "default default" styles are also the same four style definitions that are automatically copied into the *Normal.dot* template when Word is first installed, or when it is forced to create a new *Normal.dot* template.

Now let's take a look at the content of the w:styles element, extracted from Example 2-2. Preceding the style definitions themselves are two elements:

```
<w:versionOfBuiltInStylenames w:val="4"/>
<w:latentStyles w:defLockedState="off" w:latentStyleCount="156"/>
```

The w:versionOfBuiltInStylenames and w:latentStyles elements are used to refer to particular built-in styles when document formatting protection is turned on. Since document protection is an important ingredient in building custom XML solutions in Word, these elements will be covered in Chapter 4. For now, all you need to know is that there are no formatting restrictions on this document. In fact, this document would be interpreted no differently if we were to remove these two (optional) elements.

Next, there are four w:style elements, one for each of the "default default" styles listed above:

```
<w:style w:type="paragraph" w:default="on" w:styleId="Normal">
  <w:name w:val="Normal"/>
  <w:rPr>
    <wx:font wx:val="Times New Roman"/>
    <w:sz w:val="24"/>
    <w:sz-cs w:val="24"/>
    <w:lang w:val="EN-US" w:fareast="EN-US" w:bidi="AR-SA"/>
  </w:rPr>
</w:style>
<w:style w:type="character" w:default="on"
         w:styleId="DefaultParagraphFont">
  <w:name w:val="Default Paragraph Font"/>
  <w:semiHidden/>
</w:style>
<w:style w:type="table" w:default="on" w:styleId="TableNormal">
  <w:name w:val="Normal Table"/>
  <wx:uiName wx:val="Table Normal"/>
  <w:semiHidden/>
  <w:rPr>
    <wx:font wx:val="Times New Roman"/>
  </w:rPr>
  <w:tblPr>
    <w:tblInd w:w="0" w:type="dxa"/>
    <w:tblCellMar>
      <w:top w:w="0" w:type="dxa"/>
      <w:left w:w="108" w:type="dxa"/>
      <w:bottom w:w="0" w:type="dxa"/>
```

```
        <w:right w:w="108" w:type="dxa"/>
      </w:tblCellMar>
    </w:tblPr>
  </w:style>
  <w:style w:type="list" w:default="on" w:styleId="NoList">
    <w:name w:val="No List"/>
    <w:semiHidden/>
  </w:style>
```

For now, we'll only look at the lines that are highlighted. The `w:type` attribute of each `w:style` element indicates the style type (`paragraph`, `character`, `table`, or `list`). The presence of `w:default="on"` denotes that this style is the default style for its style type. This attribute's default value is `off`.

Each style has two different names, as indicated by the `w:styleId` attribute and the `w:name` element. The `w:styleId` attribute is for intra-document references only; it must be unique within the file. Styles can be referred to either from within the document's body (to associate a paragraph with a certain paragraph style, for example) or from within another style definition (to derive the style from another style, for example). The `w:styleId` attribute is unused apart from these internal associations. In fact, Word doesn't preserve its value when it opens the document. When a document is subsequently saved as XML, Word auto-generates a value for the `w:styleId` attribute, usually deriving it from the style's primary name.

The *primary name* of a style is denoted by the `w:val` attribute of the `w:name` element. The primary name of a style is what the user sees in the Style drop-down menu in the Word UI. Also, for styles that came from a template, the primary name uniquely identifies the style in the attached template and is the basis by which styles are updated when the "Automatically update document styles" document option is turned on. This name, like the `w:styleId` attribute, must be unique within the file. Otherwise, Word will try to fix things up, probably not in the way that you intended.

For certain built-in styles, the style name displayed in the Word UI differs from the primary name of the style. For example, the "Normal Table" style appears as "Table Normal" in the UI. This (dubious) privilege is restricted to Word's built-in style names; there is no way in WordprocessingML to define a custom style whose UI name differs from its primary name. Word, however, does throw us a bone when it saves such styles as XML. The `wx:uiName` element clues us in to the distinction:

```
<wx:uiName wx:val="Table Normal"/>
```

This element is strictly informational. If you were to remove it or change the `wx:val` attribute's value, Word would behave no differently when opening the file. Elements and attributes in the namespace designated by the `wx` prefix are for our benefit only and are of no internal use to Word.

The w:docPr Element

Have you ever wondered whether a particular option in the Word UI represents a property of the document you are editing as opposed to a property of the application's state? The answer to your question may lie inside the `w:docPr` element, which, like one of its siblings mentioned earlier, stands for "document properties." However, unlike the information inside the `o:DocumentProperties` element, these document properties are unique to Word and describe particular aspects of a document's state, options, and default settings, rather than metadata or statistics that are common to multiple Office applications.

The Tools → Options… dialog in the Word UI, with its many tabs, is rather notorious for being unclear about what exactly the user is modifying, whether global application options or document options. By investigating the contents of the `w:docPr` element, you can begin to identify which of these options are document-specific and which of them aren't.

The `*Pr` naming convention that `w:docPr` follows is common in WordprocessingML. As we'll see, a number of other elements follow this convention, such as `w:pPr` (paragraph properties), `w:rPr` (run properties), `w:tblPr` (table properties), `w:trPr` (table row properties), `w:tcPr` (table cell properties), and `w:listPr` (list properties). In fact, the baseline content model of these elements is also similar: a sequence of mostly empty elements, each standing for a particular property and each having zero or more attributes to set the values of that property. The most commonly used attribute is `w:val`. You may have noticed by now that WordprocessingML favors putting not only elements but also attributes in its namespace, which means you should get used to typing those `w` prefixes. (The `attributeFormDefault` value is set to `qualified` in each of the WordprocessingML schema documents.)

The `w:docPr` element has 84 optional child elements. They are declared in the WordprocessingML schema as an ordered sequence (as opposed to a repeating choice group), which suggests that they *must* occur in the declared order. In reality, Word does not enforce this order, though it does appear to follow it in the WordprocessingML documents it creates.

Now, let's look at the `w:docPr` element as output by Word in Example 2-2:

```
<w:docPr>
  <w:view w:val="web"/>
  <w:zoom w:percent="100"/>
  <w:proofState w:spelling="clean" w:grammar="clean"/>
  <w:attachedTemplate w:val=""/>
  <w:defaultTabStop w:val="720"/>
  <w:characterSpacingControl w:val="DontCompress"/>
  <w:validateAgainstSchema/>
  <w:saveInvalidXML w:val="off"/>
  <w:ignoreMixedContent w:val="off"/>
  <w:alwaysShowPlaceholderText w:val="off"/>
  <w:compat/>
</w:docPr>
```

The 11 child elements shown here provide a fairly representative sampling of these options.

The w:view element determines what view to use when opening the document. The default view for a WordprocessingML document that does not specify a view is web, which is also Word's default view for opening XML documents in general. That explains why we see the value web in this example:

```
<w:view w:val="web"/>
```

This value is the result of Word re-saving a WordprocessingML document that we constructed by hand, without specifying a view. The five possible values of view are print, outline, normal, web, and master-pages (similar to outline but applies only to documents that refer to sub-documents).

The w:zoom element denotes the zoom percentage that should be set when opening the document:

```
<w:zoom w:percent="100"/>
```

If you change the zoom percentage from within Word and re-save (provided that you also make a substantive change to the document's content to ensure that the file is actually updated), Word will save the document, recording the zoom level that you last used. Alternatively, you could directly edit the zoom property in the WordprocessingML, causing Word to display the document at some other zoom percentage the next time someone opens the file.

The w:proofState element records the state of the grammar and spelling checkers (clean or dirty) at the time Word saved the document:

```
<w:proofState w:spelling="clean" w:grammar="clean"/>
```

Since actual spelling and grammar errors are recorded in the body of the document, this state check reflects not whether there *are* errors in the document, but whether Word had a chance to finish checking for errors before the user saved the document. Thus, its primary purpose is as an optimization hint for Word when it opens the document. Its absence, however, could conceivably be a useful warning for applications that otherwise rely on Word having completed its proofing.

The w:attachedTemplate property is one of the two elements representing Templates and Add-Ins options (along with the w:linkStyles element):

```
<w:attachedTemplate w:val=""/>
```

Its value in this example is empty, which means simply that the default *Normal.dot* template is attached. Should you attach a different template (through the Tools → Templates and Add-Ins… dialog) and re-save, then this value would be populated with the specific file location of a template. Alternatively, you could manually edit the XML attribute value so that the next time Word opens the document, the new template will already be attached by virtue of your manual change. Note, however, that unless the w:linkStyles element is also present inside the w:docPr element (as explained later), the

fact that a template is merely attached has no immediate effect on the document. The w:attachedTemplate element defines a loose association whose potential is only realized when the w:linkStyles element is also present.

The w:validateAgainstSchema, w:saveInvalidXML, w:ignoreMixedContent, and w:alwaysShowPlaceHolderText properties (among several others not included in this example) are specific to Word's custom XML schema functionality (only available in Office 2003 Professional or standalone Word 2003), which is discussed in Chapter 4.

The w:defaultTabStop element sets the interval between default tab stops in the document:

```
<w:defaultTabStop w:val="720"/>
```

While the Word UI exposes this value in inches (when you select Format → Tabs...), the underlying value is stored in *twips*, or 20ths of a point, or 1,440ths of an inch. (Completing this equation, there are 72 points in an inch.) Since the value of the w:val attribute is 720 twips, the default tab stops for paragraphs in this document occur every half inch. Thus, when Word opens the document, it displays the short vertical lines beneath the ruler, spaced every half inch, as shown in Figure 2-5.

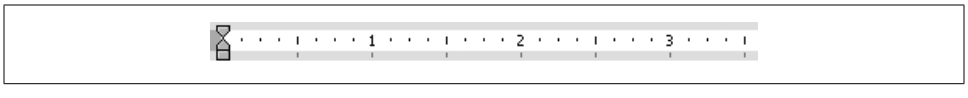


Figure 2-5. Default tab stops every half inch, or 720 twips

Once again, Word supplies this value as an application default, because our original hand-edited document (Example 2-1) did not specify a default tab stop interval. As we'll see, individual paragraphs can define their own custom tab stops too. For those paragraphs, the default tab stops only take effect to the right of the last custom stop.

The w:characterSpacingControl element is one of several Asian Typography options.

```
<w:characterSpacingControl w:val="DontCompress"/>
```

There are three possible self-describing values (DontCompress, CompressPunctuation, or CompressPunctuationAndJapaneseKana) that can be used to sets the compression option for East Asian characters. The default value that Word outputs, as evident in our example, is DontCompress. Of course, this doesn't have any real effect on our document, since it does not contain Asian characters.

Finally, the w:compat element is among the few w:docPr children that may themselves contain child elements (w:mailMerge, w:hdrShapeDefaults, w:footnotePr, w:endnotePr, and w:docVars being the only others). It has 51 possible child elements, corresponding to the compatibility options for a document that are set in the Compatibility tab of the Tools → Options... dialog, as shown in Figure 2-6.

The w:compat element is empty in Example 2-2, because our document does not set any particular compatibility options.

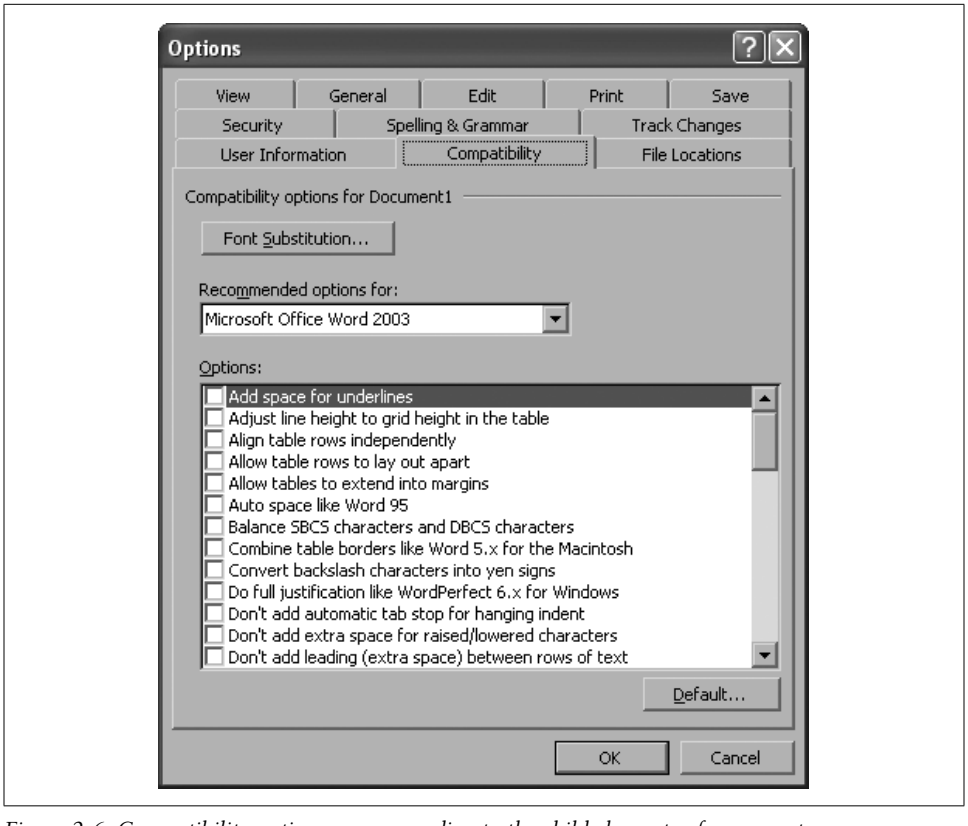


Figure 2-6. Compatibility options, corresponding to the child elements of w:compat

Before moving on, it would be good to point out one more common WordprocessingML convention. Among `w:docPr`'s 84 possible child elements, 49 are declared using the same type in the WordprocessingML schema: the `onOffProperty`. The declaration for the `onOffProperty` type in the WordprocessingML schema is as follows:

```
<xsd:complexType name="onOffProperty">
  <xsd:attribute name="val" type="onOffType" default="on"/>
</xsd:complexType>
```

The `onOffType` type referred to here allows for two possible values: `on` or `off`. As you can see, the attribute declaration for `w:val` specifies a default value of `on`. This means that for the elements inside the `w:docPr` element that are defined with this type, the presence of `w:val="on"` is always implied (and thus redundant), unless overridden by the value `off`. However, this has no bearing at all on Word's behavior when the property element itself is absent. Default behavior in those cases varies depending on the property, and the WordprocessingML schema itself does not generally cast any light on that question, although annotations therein do sometimes help. Experimentation is probably the best way to determine Word's default behavior when particular property elements are absent.

The wx:sect Element

Finally, we get to the content of our document, residing inside the w:body element. Our hand-coded original (Example 2-1) directly contained a w:p (paragraph) element inside the body. After saving, we now see that the paragraph element has been inserted into an intervening wx:sect element. As mentioned earlier, the namespace mapped to the wx prefix signals a piece of information that may be *useful to us* in processing the XML as output by Word, but that is *ignored by Word* when opening a WordprocessingML file. The wx elements and attributes are of no use to Word internally. In this case, we could remove the wx:sect element's start and end tags, leaving only its contents for Word to read, and Word would behave no differently the next time it opens the file.

That's all well and good, you might be thinking, but what is the wx:sect element *for*? As you might guess, it stands for "section." As is true with many Word documents, our "Hello World!" example document contains only one section, so it's not particularly useful in this case. To learn what sections are and how they are defined using w:sectPr elements, see "Sections" later in this chapter. And to learn how the wx:sect element is a useful aid to external processing, see "Section Containers" later in this chapter.

The w:body Element

It may seem strange to talk about the w:body element after the wx:sect element, when until now we've been traversing our original example in document order. As already noted, however, the wx:sect element is a completely optional intervening element between w:body and its content. So, while in Example 2-2 it is the wx:sect element that contains a w:p element, that content model really belongs to w:body. Using a DTD-like syntax, we can express w:body's entire content model (much more simply than its XSD definition), like this:

```
(w:p|w:tbl|w:cfChunk|w:proofErr|w:permStart|w:permEnd)*, w:sectPr?
```

In other words, w:body may contain any number of w:p, w:tbl, w:cfChunk, w:proofErr, w:permStart, and w:permEnd elements, in any order, followed by an optional w:sectPr element. The w:p element represents a paragraph, the w:tbl element represents a table, and the w:cfChunk element represents a "context-free" chunk of inline default fonts, styles, list definitions, paragraphs, and tables.* We'll describe the purpose of the w:proofErr, w:permEnd, and w:permStart elements later, in the section entitled "Proofing, Protection, and Annotation Markings."

* At least, that is how the WordprocessingML schema advertises it. A plethora of experiments yields few answers as to how this element is actually supposed to be used or how it is supposed to behave. Word tends to fix things up, merging such inline definitions with the document's global definitions. This is one area where more documentation from Microsoft is certainly needed.

The w:sectPr element, included in Example 2-2, defines the section properties for the last (and first, in this case) section of the document. See "Sections," later in the chapter, for more information on how w:sectPr elements are interpreted.

The first part of the w:body element's content model (that is, not including the optional w:sectPr element) is worth repeating:

```
(w:p | w:tbl | w:cfChunk | w:proofErr | w:permStart | w:permEnd)*
```

That's because it also functions as the content model for six other elements in WordprocessingML, namely w:hdr, w:ftr, w:footnote, w:endnote, w:tc, and w:txbxContent. (The only exception is that w:tc may also contain an optional preceding w:tcPr element.) The first two of these elements stand for "header" and "footer," respectively; they occur in the property definitions for a particular section, i.e., inside the w:sectPr element. Footnotes and endnotes may occur inside any "run," or w:r, element. The w:tc element represents a table cell; thus, tables may contain tables. Finally, the w:txbxContent element represents a text box that is embedded inside a VML (Vector Markup Language) image embedded somewhere inside a document's content.

This content model is actually more open than implied above. The WordprocessingML schema also allows any element from any *other* namespace to occur here. This enables annotations from the AML (Annotation Markup Language) namespace, as well as tags from a custom XML schema to be embedded inside WordprocessingML. (See Chapter 4.)

Document Structure and Formatting

Now that you've been inundated with information about lots of document-level constructs, let's move into the actual content of a Word document and how it is represented in WordprocessingML. All Word documents contain three levels of hierarchy: one or more *sections* containing zero or more *paragraphs* containing zero or more *characters*. A *run* is a grouping of contiguous characters that have the same properties. Tables can occur where paragraphs can, and list items are just a special kind of paragraph. You cannot have nested structures in WordprocessingML—sections within sections, or paragraphs within paragraphs. The one exception to this rule is that tables may contain tables.

Runs

A "run" is the basic leaf container for a document's content and is represented by the w:r element. As we've seen, the w:r element may contain w:t elements, which contain text. Including the w:t element, there are 24 valid child elements of the w:r element, representing things like text, images, deleted text, hyphens, breaks, tabs, footnotes, endnotes, footnote and endnote references, page numbers, field text, etc. We'll look at just a few of these.

The w:r element may occur in five separate element contexts: w:p, w:fldSimple, w:hlink, w:rt, and w:rubyBase. The first one, the paragraph, is the most common. The w:fldSimple element represents a Word field, the w:hlink element represents a hyperlink in Word, and the w:rt ("ruby text") and w:rubyBase elements are used together for laying out Asian ruby text.

The run is not an essential part of a Word document in the same way that paragraphs and sections are. Rather, it is WordprocessingML's way of grouping multiple characters (or other objects) that have the same property settings. To illustrate this point, consider the following WordprocessingML paragraph:

```
<w:p>
  <w:r><w:t>H</w:t></w:r>
  <w:r><w:t>e</w:t></w:r>
  <w:r><w:t>l</w:t></w:r>
  <w:r><w:t>l</w:t></w:r>
  <w:r><w:t>o</w:t></w:r>
  <w:r><w:t> </w:t></w:r>
  <w:r><w:t>w</w:t></w:r>
  <w:r><w:t>o</w:t></w:r>
  <w:r><w:t>r</w:t></w:r>
  <w:r><w:t>l</w:t></w:r>
  <w:r><w:t>d</w:t></w:r>
</w:p>
```

The above paragraph is exactly equivalent to the paragraph below:

```
<w:p>
  <w:r>
    <w:t>Hello world</w:t>
  </w:r>
</w:p>
```

When Word saves a document as XML, it merges consecutive runs that have the same property settings. It also merges consecutive w:t elements into a single w:t element. In the above paragraph's case, all of the run properties are assigned through the document's default paragraph and character styles, because no explicit, local property settings are applied (through the w:rPr element).

Text and whitespace handling

The w:t element, which stands for "text," has no attributes and may only contain text. Being one of the few string-valued elements in Word, it is also one of the few contexts in which whitespace is significant. The handling of whitespace within the w:t element can be summarized in three basic rules:

1. Each space character (#x20) is preserved as a space and shows up as a space in Word.
2. Each line-feed character (#xA) and character reference to a carriage-return (#xD) is converted into a space.

3. Each tab character (#x9) is replaced by a `w:tab` element (broken out into a separate run).

The one exception is that when `xml:space="default"` is present, tab characters are instead converted to spaces (and `w:tab` elements ignored altogether).

Tabs and breaks

The run inside the following WordprocessingML paragraph contains text as well as a text-wrapping break and a tab, represented by the `w:br` and `w:tab` elements.

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml"
  xml:space="preserve">

  <w:body>
    <w:p>
      <w:r>
        <w:t>This is the first line.</w:t>
        <w:br/>
        <w:t>This is a tab:</w:t>
        <w:tab/>
        <w:t>And this is some more text.</w:t>
      </w:r>
    </w:p>
  </w:body>

</w:wordDocument>
```

The first thing to note here is that the presence of `xml:space="preserve"` is necessary for the `w:tab` element to be interpreted correctly. Otherwise, the tab is stripped out when the document is loaded (even though it technically doesn't constitute whitespace as far as XML is concerned). Again, for this reason, `xml:space="preserve"` should be included on the root element of any WordprocessingML document you create.

The `w:br` element, like its HTML counterpart, inserts a break within the text flow. It is short for `<w:br w:type="text-wrapping"/>`. The `w:type` attribute may have two other values: `column` and `page`, representing column and page breaks. Figure 2-7 shows the result of opening this document in Word, with formatting marks turned on.

The bent arrow at the end of the first line indicates that this is a text-wrapping break (represented in WordprocessingML by the `w:br` element) rather than the end of the paragraph. (Word users can insert text-wrapping breaks by pressing Shift-Enter). The right-pointing arrow on the second line denotes the presence of a tab. The `w:tab` element inserts a tab into the text flow, according to the tab settings for the current paragraph. In this case, since the tab stops for this paragraph are not specified either

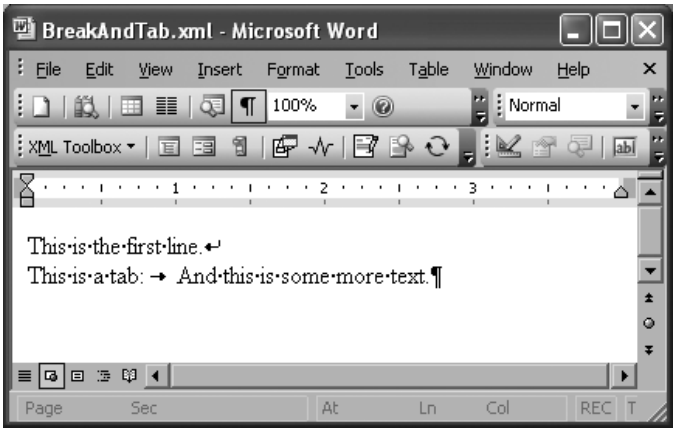


Figure 2-7. A text-wrapping break and a tab inside a single paragraph

locally or in the Normal paragraph style, the tab stops default to the application default: every half inch (as specified by the document's `w:defaultTabStop` element).

Run properties

Among all the valid child elements of `w:r`, the `w:rPr` element is special. It stands for "run properties." All of the other children of `w:r` may occur in any order, but the `w:rPr` element, when present, must come first. Its child elements collectively set properties on the run, controlling primarily how text inside the run is to be displayed. There are 42 possible child elements of the `w:rPr` element, all of which are empty elements. Their various attribute values specify formatting properties such as font, font size, font color, bold, italic, underline, strikethrough, character spacing, text effects, etc. They correspond to the properties you see in Word's Font dialog box, accessed by selecting Format → Font…, as shown in Figure 2-8.

When font settings are applied using a local `w:rPr` element, such settings are called "local settings," "manual formatting," or "direct formatting," as distinct from font settings applied through a selection's associated paragraph and character styles. *Individual font properties applied through direct formatting always override the corresponding properties defined in the associated paragraph or character styles.*

Example 2-3 shows the use of some of these formatting elements, each of which is highlighted.

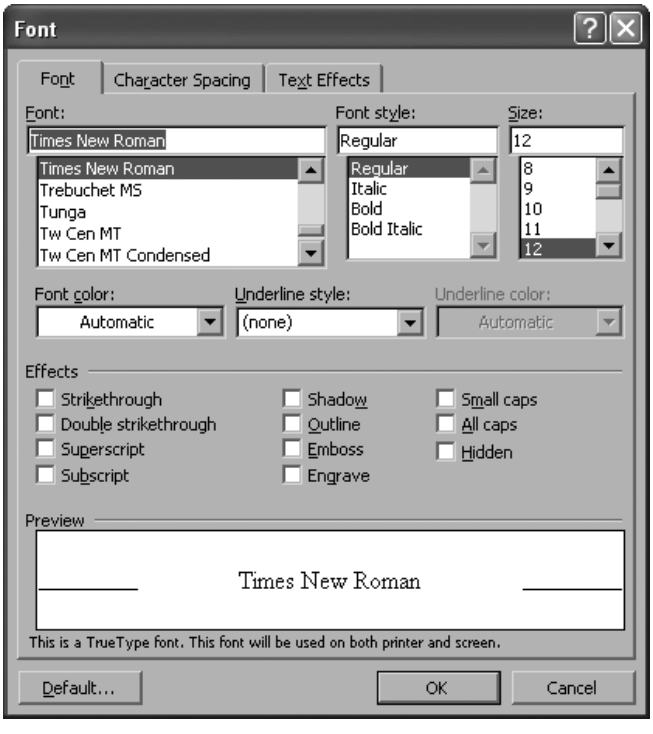


Figure 2-8. Word's font settings which correspond to run properties

Example 2-3. Applying various font properties

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml"
  xml:space="preserve">
  <w:body>
    <w:p>
      <w:r>
        <w:rPr>
          <w:i w:val="on"/> <!-- turns italics on -->
          <w:b/>            <!-- turns bold on -->
        </w:rPr>
        <w:t>This run is bold and italic. </w:t>
        <w:br/>
      </w:r>
      <w:r>
        <w:rPr>
          <w:u w:val="single"/> <!-- single underline -->
          <w:rFonts w:ascii="Arial"/>
        </w:rPr>
        <w:t>This is Arial and underlined.</w:t>
        <w:br/>
```

Example 2-3. Applying various font properties (continued)

```
      </w:r>
      <w:r>
        <w:rPr>
          <w:sz w:val="56"/>   <!-- 28-point font size -->
        </w:rPr>
        <w:t>This is big.</w:t>
      </w:r>
    </w:p>
  </w:body>
</w:wordDocument>
```

This example contains a single paragraph that contains three runs, each of which contains text. The first two runs also contain trailing text-wrapping breaks (`w:br` elements), effectively separating the text of each run onto its own line. Each run has different run properties specified in the `w:rPr` element. These properties, since they are applied as direct formatting, override the corresponding settings in the Normal style (the "default default" paragraph style, as we saw earlier).

The first run introduces the `w:b` and `w:i` elements:

```
<w:rPr>
  <w:i w:val="on"/> <!-- turns italics on -->
  <w:b/>            <!-- turns bold on -->
</w:rPr>
```

The `w:b` and `w:i` elements stand for "bold" and "italic," respectively. They are among 19 of `w:rPr`'s 42 possible child elements that, like many of `w:docPr`'s children, are declared with the `onOffProperty` type in the WordprocessingML schema. This means that the default value of the `w:val` attribute is `on`. Thus, `w:val="on"` on the `w:i` element above is technically redundant. As might be guessed, by turning these properties on, all of the text within the run will be formatted in bold weight and italic style.

The presence of the `w:val` attribute is necessary to turn *off* a particular property, overriding its setting in the style. For example, if you want to turn off bold for a particular portion of text that's associated as a whole with a style in which the bold property is turned on, then you would include `<w:b w:val="off"/>` inside the `w:rPr` element.

The second run in Example 2-3 introduces the `w:u` and `w:rFonts` elements:

```
<w:rPr>
  <w:u w:val="single"/> <!-- single underline -->
  <w:rFonts w:ascii="Arial"/>
</w:rPr>
```

The `w:u` element is similar to `w:b` and `w:i`, in that it is empty and has a `w:val` attribute. The difference is that, instead of having only the values `on` and `off`, you have a choice between 18 different values, including `single` (as in this example) and `none`. These values correspond to the choices in the "Underline style" drop-down menu in Word's Font dialog.

This run also specifies the Arial font, overriding the default Times New Roman font of the Normal style. This is done using the `w:rFonts` element, which has the same declared type in the WordprocessingML schema as the global `w:defaultFonts` element we saw earlier. Specifically, it allows the same attributes for specifying the fonts of different character sets: `w:ascii`, `w:h-ansi`, `w:cs`, and `w:fareast`. In this case, only the `w:ascii` attribute is supplied, which means that the other character sets still assume the default font.

The third and final run in our single-paragraph document sets the font size using the `w:sz` element:

```
<w:rPr>
  <w:sz w:val="56"/>   <!-- 28-point font size -->
</w:rPr>
```

The value of the `w:val` attribute in this case is measured in half-points, or 10 twips, or 144ths of an inch. Thus, while its value is 56 in the XML, the actual font size (in full points) is 28.

Finally, we see the result of opening this document in Word in Figure 2-9.

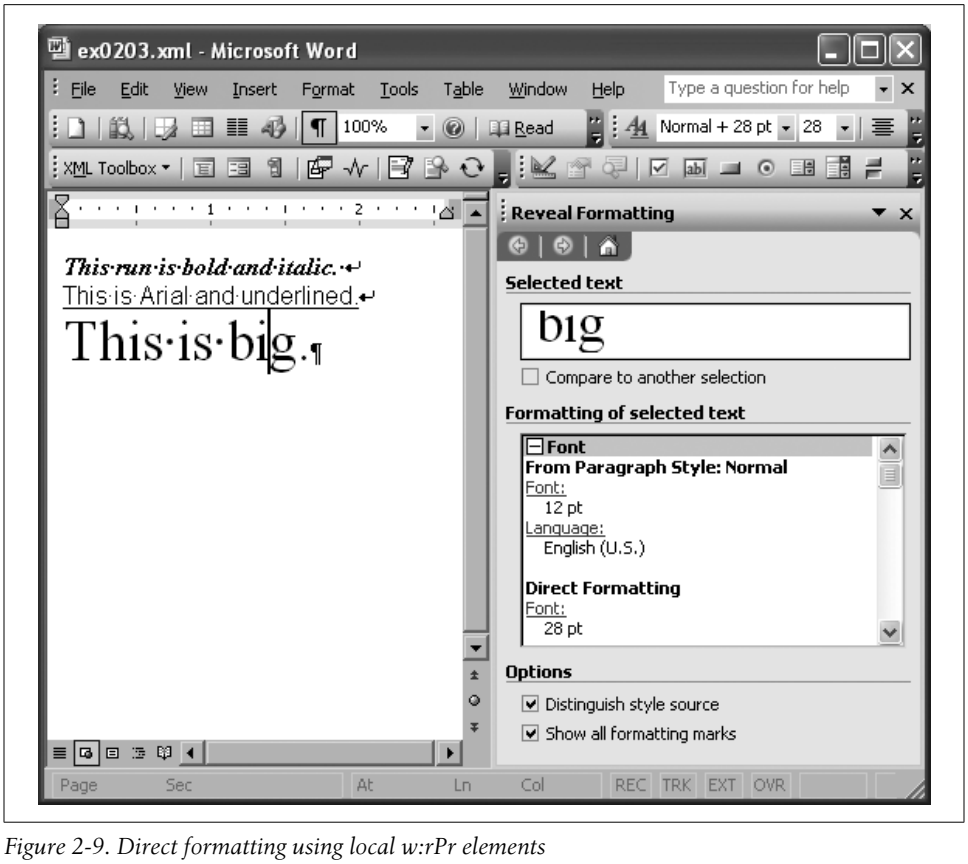


Figure 2-9. Direct formatting using local w:rPr elements

Figure 2-9 also shows how direct formatting is represented in the Word UI. In this case, the cursor is inside the third run, containing the text "This is big." There are two things worth noting about how this direct formatting is represented:

- The style drop-down box, as shown at the top right of the window, says "Normal + 28 pt." This is how all direct formatting is represented here (style name + individual property settings).
- The Reveal Formatting task pane, because "Distinguish style source" is checked, distinguishes between the font size as set in the Normal style (12 pt) and the overriding font size as applied through Direct Formatting (28 pt).

Associating a run with a character style

In addition to specifying direct formatting, a run can explicitly associate itself with one of its document's character styles. This is done using the `w:rStyle` element. Below are three runs excerpted from a document in which the "Hyperlink" character style is defined. All three runs are associated with the "Hyperlink" style, but the middle run also applies some direct formatting (italics):

```
<w:r>
  <w:rPr>
    <w:rStyle w:val="Hyperlink"/>
  </w:rPr>
  <w:t>This just </w:t>
</w:r>
<w:r>
  <w:rPr>
    <w:rStyle w:val="Hyperlink"/>
    <w:i/>
  </w:rPr>
  <w:t>looks</w:t>
</w:r>
<w:r>
  <w:rPr>
    <w:rStyle w:val="Hyperlink"/>
  </w:rPr>
  <w:t> like a hyperlink.</w:t>
</w:r>
```

Figure 2-10 shows the result of opening this document in Word, assuming it has defined the "Hyperlink" style in its `w:styles` element (rendering the font blue and underlined).

Once again, the Reveal Formatting task pane shows the distinction between the properties applied through direct formatting ("Italic") and the properties defined in a style ("Font color: Blue" and "Underline"). It also reveals the character style for this run: "Hyperlink."

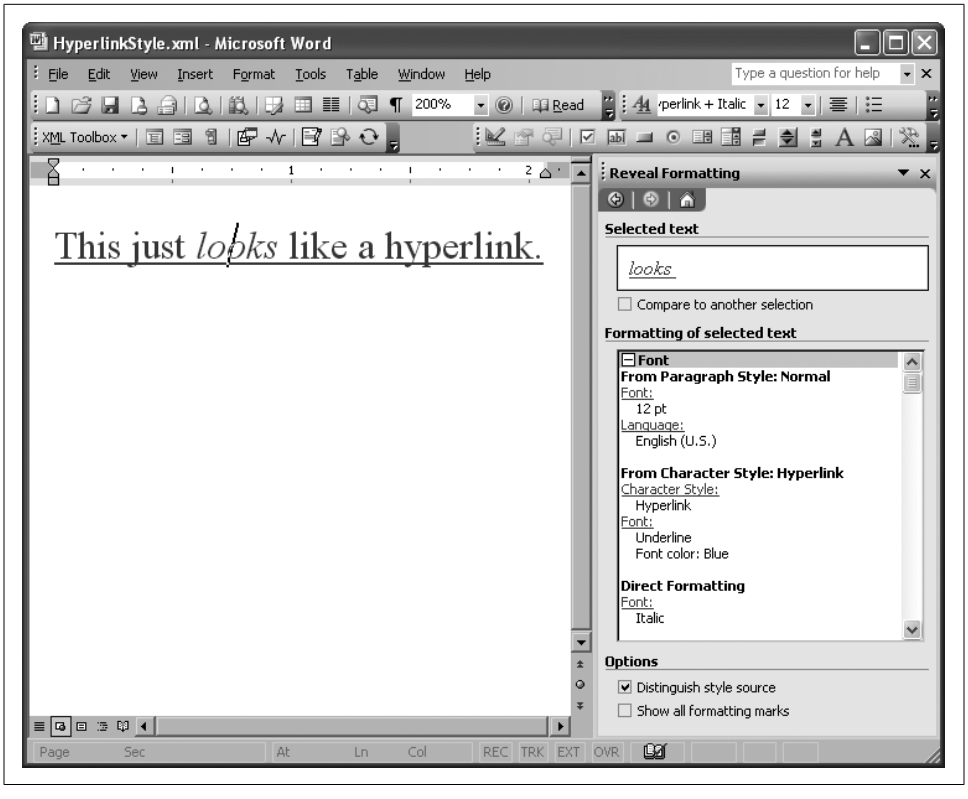


Figure 2-10. A run of text associated with the "Hyperlink" style

Paragraphs

Paragraphs are the basic block-oriented element in Word. All text content within a document is contained within paragraphs, whether it's inside the main body of the document, a table cell, a header, a footer, a footnote, an endnote, or a textbox embedded in an image. Normally, a new paragraph is created whenever a user hits the Enter key while editing.

In WordprocessingML, a paragraph is represented by the `w:p` element. The area inside the `w:p` element could be called a "run-level" context, because it is a context in which runs (`w:r` elements) may appear. Similarly, the area inside the `w:body` element is a "block-level" context, because it is a context in which paragraphs and tables may appear. The traditional distinction between a block and an *inline* element (or run) is that blocks are laid out on separate lines, whereas inline elements (runs) are laid out continuously, without any hard line breaks.

The content model of the w:p element is simple enough that it's worth showing here (using a DTD-like notation):

```
w:pPr?,
(w:r|w:proofErr|w:permStart|w:permEnd|w:fldSimple|w:hlink|w:subDoc)*
```

This follows the same pattern as w:r's content model: an optional properties element followed by any of a number of element choices in any order. (We didn't show w:r's entire content model because it has so many element choices.)

Three of the elements in w:p's content model, as we've seen, may also occur as children of w:body. The w:proofErr, w:permStart, and w:permEnd elements are thus both block-level and run-level elements. They are explained later in "Proofing, Protection, and Annotation Markings."

The w:fldSimple element represents a Word field, and the w:hlink element represents a hyperlink in Word. You may recall that these elements are also run-level contexts, i.e., they themselves may contain runs. The w:subDoc element represents a link to a sub-document of the current document.

As is the case with the w:body element, w:p's content model is actually more open than implied above. The WordprocessingML schema also allows any element from any *other* namespace to occur here. This enables annotations from the AML (Annotation Markup Language) namespace, as well as tags from a custom XML schema to be embedded inside WordprocessingML. As we'll see in Chapter 4, Word renders custom XML tags differently depending on whether they occur at the block level (inside w:body) or run level (inside w:p).

Paragraph properties

Among all the valid child elements of w:p, the w:pPr element is special. It stands for "paragraph properties." All of the other children of w:p may occur in any order, but the w:pPr element, when present, must come first. Its child elements collectively set properties on the paragraph, controlling how the paragraph will be displayed. There are 34 possible child elements of the w:pPr element, many but not all of which are empty elements. Their various attribute values and child elements specify paragraph properties such as alignment, indentation, spacing, tab stops, widow/orphan control, paragraph borders, etc. Most of these properties correspond to the properties you see in Word's Paragraph dialog box, accessed by selecting Format → Paragraph..., as shown in Figure 2-11.

When paragraph settings are applied using a local w:pPr element, such settings are called "local settings," "manual formatting," or "direct formatting," as distinct from settings applied through a paragraph's associated paragraph style. *Individual paragraph properties applied through direct formatting always override the corresponding properties defined in the associated paragraph style*. If this sounds familiar, it should. It's the same basic rule as for font settings. Local w:rPr and w:pPr elements always

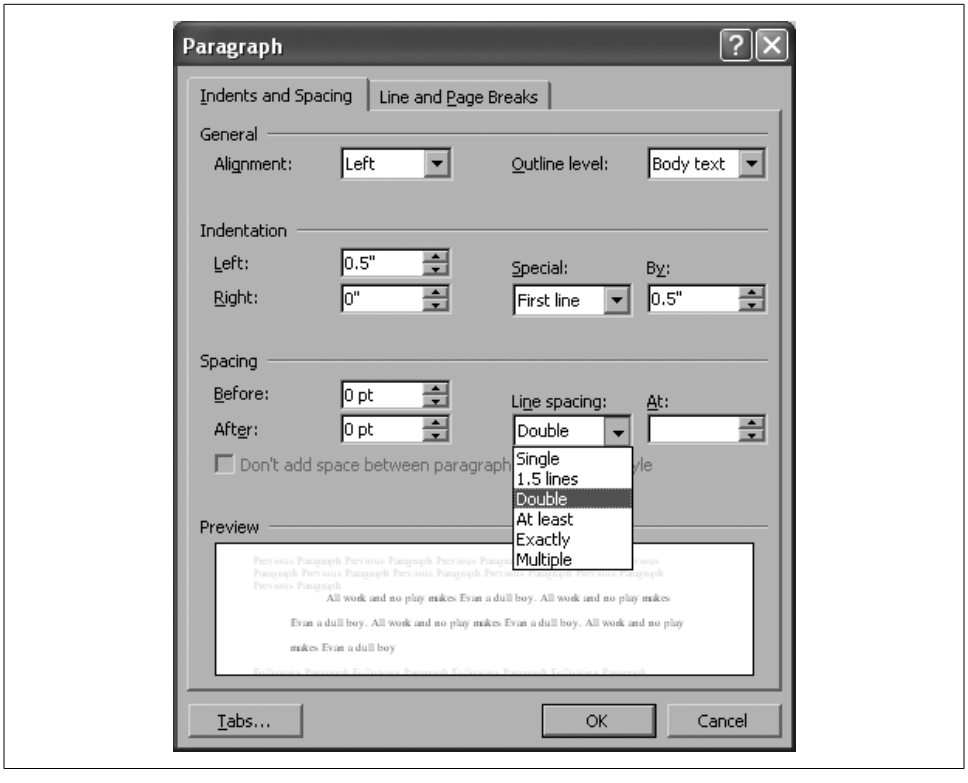


Figure 2-11. Word's Paragraph dialog, corresponding to properties inside the w:pPr element

override settings applied through (explicit or default) style association. Also, the properties within the `w:rPr` and `w:pPr` elements are completely disjoint from each other, so there is no possibility of conflict between these two elements.

Example 2-4 shows the use of some of these paragraph formatting elements, each of which is highlighted.

Example 2-4. Applying various paragraph properties

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml"
  xml:space="preserve">
  <w:body>
    <w:p>
      <w:pPr>
        <w:jc w:val="center" />
      </w:pPr>
      <w:r>
        <w:t>All work and no play makes Evan a dull boy.</w:t>
      </w:r>
```

Example 2-4. Applying various paragraph properties (continued)

```
    </w:p>
    <w:p />
    <w:p>
      <w:pPr>
        <w:spacing w:line="480" w:line-rule="auto" />
        <w:ind w:left="720" w:first-line="720" />
      </w:pPr>
      <w:r>
        <w:t>All work and no play makes Evan a dull boy. All work and no play makes Evan a
             dull boy. All work and no play makes Evan a dull boy. All work and no play
             makes Evan a dull boy.</w:t>
      </w:r>
    </w:p>
    <w:p>
      <w:pPr>
        <w:ind w:left="2880" w:right="2880" />
      </w:pPr>
      <w:r>
        <w:t>All work and no play makes Evan a dull boy.</w:t>
      </w:r>
    </w:p>
  </w:body>
</w:wordDocument>
```

The result of opening this document in Word is shown in Figure 2-12. Also, the Format → Paragraph… dialog shown earlier in Figure 2-11 reflects the paragraph settings of the third paragraph of this example (note that the second paragraph is empty).

Example 2-4 contains four paragraphs. The second paragraph is empty and does not apply any direct formatting. The other three each specify paragraph properties that override the corresponding settings in the Normal style (the "default default" paragraph style).

The first paragraph is centered. The `w:jc` element represents the paragraph justification settings:

```
<w:jc w:val="center" />
```

Its `w:val` attribute value may be `left`, `center`, `right`, `both`, or one of several other options specific to East Asian text. The first four values correspond to the "Left," "Centered," "Right,", and "Justified" options in the Alignment drop-down menu in the Format → Paragraph… dialog.

The second non-empty paragraph is double-spaced, indented on the left, and has a first-line indent. The double-spacing effect is achieved through the `w:spacing` element:

```
<w:spacing w:line="480" w:line-rule="auto" />
```

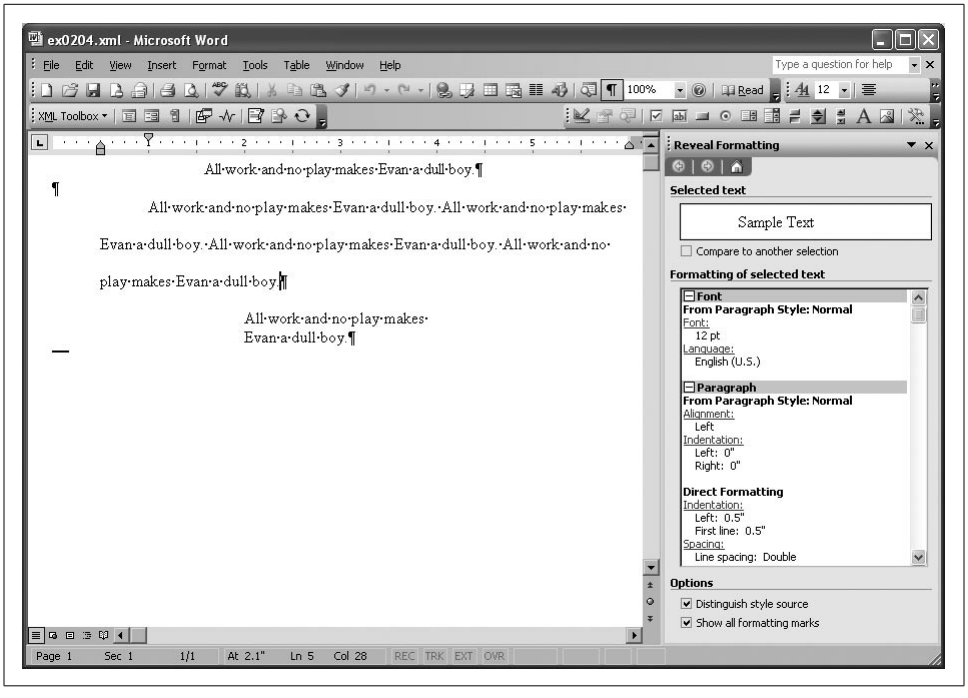


Figure 2-12. Applying paragraph properties as direct formatting

Unlike the `w:jc` element, which has specific keywords corresponding to each of the UI options, the `w:spacing` element specifies its values numerically—in twips. The `w:line` attribute's value of `480` (equivalent to 24 points), in conjunction with the `w:line-rule` attribute's value of `auto`, represent the overall setting of "Double" in the Line Spacing drop-down menu in the Format → Paragraph… dialog, as shown earlier in Figure 2-11. When the `w:line-rule` attribute's value is `auto`, then the `w:line` attribute's value is interpreted in a pre-defined way, regardless of the current paragraph's font size. A value of `480` means "Double," `360` means "1.5 line," and `240` means "Single." The actual line spacing distance is automatically adjusted according to the current font size, but the `w:line` attribute's value stays the same. The other possible values of `w:line-rule` are `exact` and `at-least`. These correspond to the "Exactly" and "At least" options in the Line Spacing drop-down menu and affect how the `w:line` value is interpreted. For example, a value of `exact` would fix the line spacing distance to the specified value in the `w:line` attribute, regardless of the current font size. The `w:spacing` element also has other attributes (not present in this example) that are used to determine the spacing before and after the paragraph itself.

The indentation of the third paragraph (following the empty second paragraph) is specified using the `w:ind` element:

```
<w:ind w:left="720" w:first-line="720" />
```

The w:left attribute specifies the left indentation distance as 720 positive twips, or half an inch to the right of the page margin. (Negative indent values move the text into the page margin.) The w:first-line attribute specifies a first-line indent of another half inch. The effect of these settings on Word's ruler is shown in Figure 2-13.

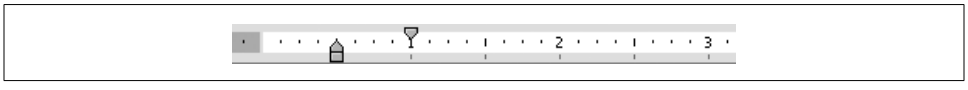


Figure 2-13. A half-inch left indent and a half-inch first-line indent

The w:ind element may also have a w:hanging attribute which specifies a hanging indent. Its presence is mutually exclusive with the w:first-line attribute, because the same paragraph cannot have both first-line and hanging indents. If our example used a hanging indent rather than a first-line indent, then the WordprocessingML would look like this:

```
<w:ind w:left="720" w:hanging="720" />
```

And the ruler would look like Figure 2-14.

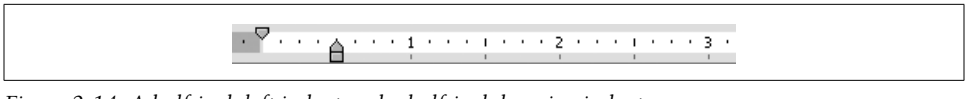


Figure 2-14. A half-inch left indent and a half-inch hanging indent

Interestingly enough, you can also supply negative values for the w:first-line and w:hanging attributes. Since a hanging indent is essentially the opposite of a first-line indent, Word interprets a negative value as if you had supplied a positive value of the other type of indent. In fact, when it subsequently saves the document as WordprocessingML, it replaces one attribute with the other attribute (w:hanging with w:first-line or vice versa) and its negative value with its opposite (positive) value. For example, if you open a document that has this:

```
<w:ind w:hanging="-720" />
```

then Word will normalize it to this instead:

```
<w:ind w:first-line="720" />
```

The two are equivalent.

The last paragraph in Example 2-4 has both right and left indents:

```
<w:ind w:left="2880" w:right="2880" />
```

The positive value (in twips) of 2880 in each of the w:left and w:right attributes means that the paragraph will be indented two inches from the margin on each side.

The w:left, w:right, w:first-line, and w:hanging attributes all measure distance in twips. You can alternatively measure distance in character spaces, by using the w:ind

element's other four optional attributes instead: `w:left-chars`, `w:right-chars`, `w:first-line-chars`, and `w:hanging-chars`.

Defining tab stops

Paragraphs can specify custom tab stops, overriding the document's default tab stop interval. This is done using the `w:tabs` child element of a paragraph's `w:pPr` element. Example 2-5 shows a paragraph with custom tab stops as well as some tabs inside the paragraph that make use of those stops.

Example 2-5. Defining custom tab stops

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml"
  xml:space="preserve">
  <w:body>
    <w:p>
      <w:pPr>
        <w:tabs>
          <w:tab w:val="left" w:pos="720" />
          <w:tab w:val="center" w:pos="3600" />
          <w:tab w:val="right" w:pos="6480" />
        </w:tabs>
      </w:pPr>
      <w:r>
        <w:tab/>
        <w:t>Left-aligned tab</w:t>
        <w:tab/>
        <w:t>Centered tab</w:t>
        <w:tab/>
        <w:t>Right-aligned tab</w:t>
      </w:r>
    </w:p>
  </w:body>
</w:wordDocument>
```

Each `w:tab` element within the `w:tabs` element defines a different tab stop. Both the `w:val` and `w:pos` attributes are required. The `w:val` attribute indicates the type of tab stop, controlling the alignment of text around it. Its value must be one of `left`, `center`, `right`, `decimal`, `bar`, `list`, or `clear`. (The value `clear` enables tab stops defined in an associated paragraph style to be explicitly cleared.) The `w:pos` attribute specifies the position of the tab stop on the ruler, as the number of twips to the right of the left page margin. The `w:tab` element may also have an optional `w:leader` attribute, which sets the style of the empty space in front of the tab. These properties correspond to the settings found in Word's Format → Tabs... dialog, shown in Figure 2-15, which here is populated with the same tab stops as defined in Example 2-5.

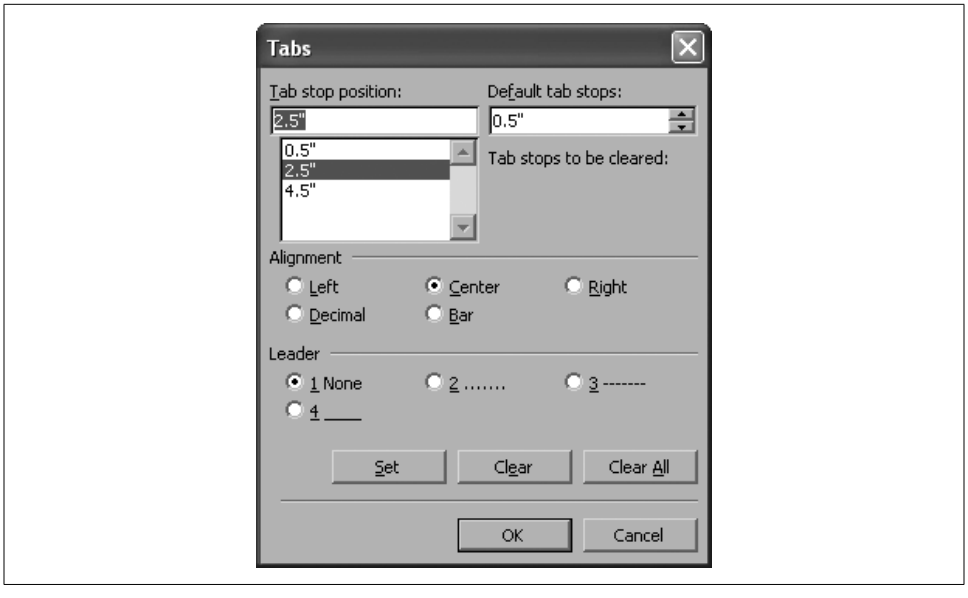


Figure 2-15. Tab stop definitions, corresponding to Example 2-5

Finally, the result of opening this file in Word is shown in Figure 2-16, with formatting marks turned on.

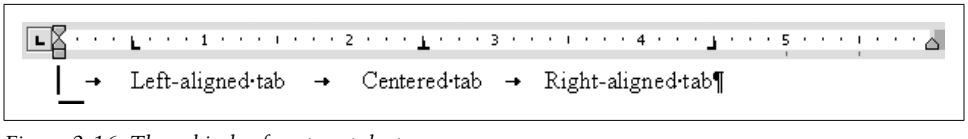


Figure 2-16. Three kinds of custom tab stops

The custom tab stops can be seen on the ruler, and the tabs themselves are signified by arrows in the document content. The document's default tab stops (every half inch) are signified by small vertical lines below the ruler and do not resume until after the last custom tab, beginning at the 5-inch mark.

Paragraph mark properties

You may be surprised to learn that the `w:rPr` element ("run properties") may also occur as a child of the `w:pPr` element. Actually, it shows up quite often when editing documents in Word. For example, if you turn bold on, type a short paragraph, and hit Enter, then the resulting paragraph in WordprocessingML will look like this:

```
<w:p>
  <w:pPr>
    <w:rPr>
      <w:b/>
    </w:rPr>
  </w:pPr>
```

```
    <w:r>
      <w:rPr>
        <w:b/>
      </w:rPr>
      <w:t>This text is bold.</w:t>
    </w:r>
  </w:p>
```

This may look redundant, but it isn't. By now, you should be familiar with the purpose of the second `w:rPr` element above. It sets the properties (in this case, bold) on the run in which it is contained. However, the first `w:rPr` element (inside the `w:pPr` element) functions differently than you might expect. Rather than setting properties of the runs inside the paragraph, it represents properties of the paragraph's *paragraph mark*. If we removed the first `w:rPr` element altogether, it would have no actual effect on the formatting of our document. In fact, we wouldn't even see a difference in the Word UI—unless paragraph marks are turned on. In that case, we might notice whether or not the paragraph mark itself is displayed in bold weight.

The run properties, or font settings, of a paragraph mark, though they do not directly affect the paragraph's formatting, do have an effect on Word's *behavior* when subsequently editing the document. For that reason, you can think of the paragraph mark properties as containing information about your document's editing state rather than its actual formatting. For example, one practical effect of setting bold on a paragraph mark is that if the user selects the paragraph mark (by double-clicking it) and drags and drops it to create a new paragraph, bold will be turned on by default for runs entered in the new paragraph.

In practice, Word synchronizes the font settings of the paragraph mark with the font settings of the last run in the paragraph. For example, if you are typing a paragraph and you hit Enter when italics are turned on, then the paragraph mark of the paragraph you just created will also have italics turned on, as will the paragraph mark of the following paragraph, at least initially. If, on the other hand, you turn italics off right before you hit the Enter key, then the last part of your paragraph will still be italicized, but the paragraph mark won't be, and neither will the following paragraph's paragraph mark.

One final example may help elucidate the function of paragraph mark properties. Consider the WordprocessingML document in Example 2-6. It is devoid of any text content, but it does have one empty paragraph whose paragraph mark has italics turned on.

Example 2-6. An empty paragraph with italics turned on

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml"
  xml:space="preserve">
  <w:body>
```

Example 2-6. An empty paragraph with italics turned on (continued)

```
    <w:p>
      <w:pPr>
        <w:rPr>
          <w:i/>
        </w:rPr>
      </w:pPr>
    </w:p>
  </w:body>
</w:wordDocument>
```

If we open this document in Word, we'll see nothing but a blank document with a flashing cursor—an *italicized* flashing cursor. This, again, reflects the document's editing state, rather than its formatting. Any time you create a new paragraph while editing, Word tries to remember the formatting properties you had in effect on the last paragraph—even when you create an empty paragraph, save the document, close it, and open it again later, which is what Example 2-6 demonstrates.

It's good to clear up the potential confusion surrounding `w:pPr`'s seemingly redundant `w:rPr` child. Now that you're cognizant of what instances of this element do *not* represent, you can safely exclude them from WordprocessingML documents that you create. Their absence will have negligible impact on the user's editing experience. Don't worry—Word will still work its magic.

Associating a paragraph with a paragraph style

In addition to specifying direct formatting, a paragraph can explicitly associate itself with one of its document's paragraph styles. This is done using the `w:pStyle` element. Below is a paragraph excerpted from a document in which the "Heading1" paragraph style is defined:

```
<w:p>
  <w:pPr>
    <w:pStyle w:val="Heading1" />
  </w:pPr>
  <w:r>
    <w:t>This is a heading</w:t>
  </w:r>
</w:p>
```

This paragraph will be formatted according to the explicitly associated paragraph style, provided that the containing document has a style definition that looks something like this:

```
<w:style w:type="paragraph" w:styleId="Heading1">
  <w:name w:val="Heading 1"/>
  <!-- other style options -->
  <w:pPr>
    <!-- paragraph property settings -->
  </w:pPr>
```

```
    <w:rPr>
      <!-- font property settings -->
    </w:rPr>
  </w:style>
```

Tables

Tables may occur anywhere that paragraphs may occur (and vice versa), which most commonly is directly inside the `w:body` element (or inside an intervening `wx:sect` element when the WordprocessingML is output by Word). The other contexts in which paragraphs and tables may occur are the `w:hdr`, `w:ftr`, `w:footnote`, `w:endnote`, `w:tc`, `w:txbxContent`, and `w:cfChunk` elements, which we already introduced briefly.

The basic structure of the `w:tbl` element looks like this:

```
<w:tbl>
   <w:tblPr>...</w:tblPr>
   <w:tblGrid>
     <w:gridCol w:val="..."/>
     <w:gridCol w:val="..."/>
     ...
   </w:tblGrid>
   <w:tr>
     <w:tc>...</w:tc>
     <w:tc>...</w:tc>
    ...
   </w:tr>
   <w:tr>...</w:tr>
   ...
</w:tbl>
```

The content model for the `w:tbl` element, using a DTD-like syntax, is:

```
aml:annotation*, w:tblPr, w:tblGrid,
(w:tr | w:proofErr | w:permStart | w:permEnd)+
```

In other words, the `w:tbl` element may contain zero or more `aml:annotation` elements, followed by a `w:tblPr` element and a `w:tblGrid` element, followed by one or more `w:tr`, `w:proofErr`, `w:permStart`, or `w:permEnd` elements, in any order. The `w:tblPr` element contains table-wide properties. The `w:tblGrid` element contains `w:gridCol` elements that define the widths of columns in the table.

Table rows are represented by the `w:tr` element. The content model of the `w:tr` element, using the same notation, is:

```
w:tblPrEx?, w:trPr?, (w:tc | w:proofErr | w:permStart | w:permEnd)+
```

The `w:tblPrEx` element contains exceptions to the table-wide properties for this row only. The `w:trPr` element contains table row properties for this row.

Table cells are represented by the `w:tc` element. The content model of the `w:tc` element, using the same notation, is:

```
w:tcPr?,(w:p | w:tbl | w:cfChunk | w:proofErr | w:permStart | w:permEnd)*
```

Thus, after optionally specifying the table cell properties (with the w:tcPr element), we are once again inside a block-level context. At this point, paragraphs may contain the text for the table cell, or another table can be nested inside this one.

We've repeatedly seen the trio of w:proofErr, w:permStart, and w:permEnd—now at row-level, cell-level, block-level, and run-level contexts. See "Proofing, Protection, and Annotation Markings," later in this chapter, to find out what exactly these elements are for and how they function.

Example 2-7 shows a simple table that references one of its document's table styles and additionally utilizes several table formatting features.

Example 2-7. A sample table with a style and merged cells

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml"
  xml:space="preserve">
  <w:styles>
    <w:style w:type="table" w:styleId="MyTableStyle">
      <w:name w:val="My Table Style" />
      <w:tblPr>
        <w:tblBorders>
          <w:top w:val="single"/>
          <w:left w:val="single"/>
          <w:bottom w:val="single"/>
          <w:right w:val="single"/>
          <w:insideH w:val="single"/>
          <w:insideV w:val="single"/>
        </w:tblBorders>
        <w:tblCellMar>
          <w:left w:w="108" w:type="dxa" />
          <w:right w:w="108" w:type="dxa" />
        </w:tblCellMar>
      </w:tblPr>
    </w:style>
  </w:styles>
  <w:body>
    <w:tbl>
      <w:tblPr>
        <w:tblStyle w:val="MyTableStyle" />
      </w:tblPr>
      <w:tr>
        <w:tc>
          <w:p>
            <w:r>
              <w:t>First row, first column</w:t>
            </w:r>
          </w:p>
        </w:tc>
        <w:tc>
          <w:tcPr>
```

Example 2-7. A sample table with a style and merged cells (continued)

```
            <w:vmerge w:val="restart" />
          </w:tcPr>
          <w:p>
            <w:r>
              <w:t>First row, second column (merged with second row, second
              column)</w:t>
            </w:r>
          </w:p>
        </w:tc>
      </w:tr>
      <w:tr>
        <w:tc>
          <w:p>
            <w:r>
              <w:t>Second row, first column</w:t>
            </w:r>
          </w:p>
        </w:tc>
        <w:tc>
          <w:tcPr>
            <w:vmerge />
          </w:tcPr>
          <w:p/>
        </w:tc>
      </w:tr>
    </w:tbl>
  </w:body>
</w:wordDocument>
```

The result of opening this WordprocessingML document in Word is shown in Figure 2-17.

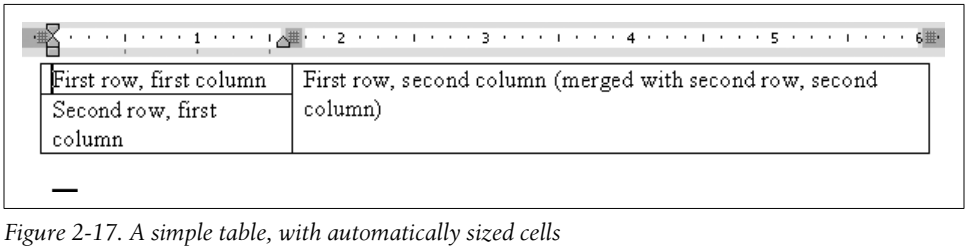


Figure 2-17. A simple table, with automatically sized cells

There are a few things to note about this table:

- The table is associated with "MyTableStyle," which is defined within the document.
- The "MyTableStyle" style adds borders and cell-spacing to the table.
- Word opens the document without complaint, even though it doesn't have a `w:tblGrid` element; Word automatically sizes the cells to contain the content.

- The w:vmerge element is a table cell property that is used to vertically merge one table cell with another table cell below it—similar to its horizontal equivalent, the w:hmerge element.
- The w:tbl element as generated by Word tends to be much more verbose than this example, explicitly specifying many individual property settings.

There is a lot that this example doesn't cover. To give you an idea just how much more there is to tables, the w:tblPr element has 17 possible child elements (many of which contain their own children), the w:trPr element has 12 possible child elements, and the w:tcPr element has 13 possible child elements. That's not to mention the w:tblPrEx (exceptions for a specific row), w:tblStylePr (for table-style conditional override properties), and w:tblpPr (for specifying the position of a table) elements. If you're writing WordprocessingML for tables, the main things you'll need to configure are the properties of the table, rows, and cells. These work in the same way as the paragraph properties that we've looked at in detail earlier, so we won't go into them here. A quick look at the properties dialogs for tables should give you an idea of what's involved.

Lists

Lists are a rather strange beast in WordprocessingML. Though tables can get pretty hairy, they at least are generally structured the way you would expect: tables containing rows containing cells. Lists, on the other hand, have no such explicit structure in WordprocessingML. Instead, a list consists of a sequence of paragraphs that function as list items. They do not have a common container, nor, unfortunately, does Word provide an auxiliary hint for list containers when outputting WordprocessingML. The member paragraphs of a list are linked to one of its document's "list definitions." These are responsible for maintaining the identity of a single list. When numbering restarts, for example, a new list definition is automatically created. These list definitions, in turn, are linked to one of the document's "base list definitions", which, if there is no subsequent list style link to traverse, define the actual formatting properties of the list. If the phrase "spectacularly convoluted" comes to mind, just wait until you see an example of this.

What makes a paragraph a list item

A paragraph participates as a member of a list under one of these separate circumstances:

- It has a w:listPr element inside its w:pPr element, which refers to a specific list definition (via the w:ilfo element).
- It is associated with a paragraph style that includes list formatting.

Let's take a look at how the first mechanism works. The following paragraph is a member of a list:

```
<w:p>
  <w:pPr>
    <w:listPr>
      <w:ilvl w:val="0"/>
      <w:ilfo w:val="1"/>
    </w:listPr>
  </w:pPr>
  <w:r>
    <w:t>This is item one.</w:t>
  </w:r>
</w:p>
```

The `w:ilfo` element (whose name may stand for something like "item list format," though Microsoft has not documented what it actually means) refers to one of the document's list definitions, identified by the number `1`. The `w:ilvl` element specifies at what level of nesting this list item occurs. It is incremented each time a list is nested within another list. Since there are nine possible levels of list indentation in Word (starting at `0`), its value can be anywhere from `0` to `8`. It basically says, "Once you find the definition for how each level of this list is supposed to look, sign me up for the formatting and indentation that are defined for level `0`." Finding the list definition is the trick. But before we figure out how that's done, let's take a look at how WordprocessingML lists compare with HTML lists.

Comparing HTML and WordprocessingML lists

Below is a simple nested list in HTML:

```
<ol>
  <li>
    <p>This is top-level item 1</p>
    <ol>
      <li>This is second-level item 1</li>
      <li>This is second-level item 2</li>
    </ol>
  </li>
  <li>This is top-level item 2</li>
</ol>
```

In WordprocessingML, a list like this is expressed much differently. Instead of using a hierarchical structure to express the list hierarchy, we must represent the list as a flat sequence of four sibling paragraphs, assigning them to the same list but to different levels within the list:

```
<w:p>
  <w:pPr>
    <w:listPr>
      <w:ilvl w:val="0"/>
      <w:ilfo w:val="1"/>
    </w:listPr>
```

```
    </w:pPr>
    <w:r>
      <w:t>This is top-level item 1</w:t>
    </w:r>
  </w:p>
  <w:p>
    <w:pPr>
      <w:listPr>
        <w:ilvl w:val="1"/>
        <w:ilfo w:val="1"/>
      </w:listPr>
    </w:pPr>
    <w:r>
      <w:t>This is second-level item 1</w:t>
    </w:r>
  </w:p>
  <w:p>
    <w:pPr>
      <w:listPr>
        <w:ilvl w:val="1"/>
        <w:ilfo w:val="1"/>
      </w:listPr>
    </w:pPr>
    <w:r>
      <w:t>This is second-level item 2</w:t>
    </w:r>
  </w:p>
  <w:p>
    <w:pPr>
      <w:listPr>
        <w:ilvl w:val="0"/>
        <w:ilfo w:val="1"/>
      </w:listPr>
    </w:pPr>
    <w:r>
      <w:t>This is top-level item 2</w:t>
    </w:r>
  </w:p>
```

For this list to display correctly, the document must contain at least one list definition (a `w:list` element with `w:ilfo="1"`, as we'll see) and a corresponding base list definition (`w:listDef` element), which contains the actual formatting information for list items. Each paragraph's `w:ilvl` value represents how far it is nested in the list. The "top-level" paragraphs are each at level `0`, whereas the "second-level" paragraphs are each at level `1`. Figure 2-18 shows how Word renders this WordprocessingML list, using one of its built-in list styles.

Finding the list definitions

Now let's take a look at where the "list definitions" and "base list definitions" are actually defined. Unsurprisingly, they are both to be found inside the top-level `w:lists`

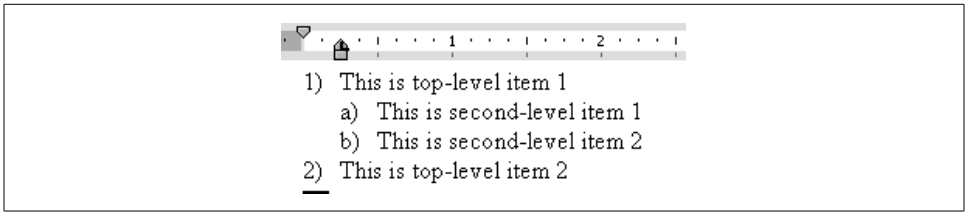


Figure 2-18. A simple nested list in Word

element, whose basic content model is a sequence of `w:listDef` elements followed by a sequence of `w:list` elements:

```
<w:lists>
  <w:listDef ...>
    ...
  </w:listDef>
  <!-- more w:listDef elements -->
  <w:list ...>
    ...
  </w:list>
  <!-- more w:list elements -->
</w:lists>
```

The `w:list` elements represent what we're calling "list definitions," and the `w:listDef` elements represent what we're calling "base list definitions."

Consider the first example list paragraph we saw earlier. This will be our starting point for finding the list definitions in the same way that Word does. Here's the paragraph again:

```
<w:p>
  <w:pPr>
    <w:listPr>
      <w:ilvl w:val="0"/>
      <w:ilfo w:val="1"/>
    </w:listPr>
  </w:pPr>
  <w:r>
    <w:t>This is item one.</w:t>
  </w:r>
</w:p>
```

Since our paragraph's `w:ilfo` element refers to the value `1`, we need to find the list definition identified by the number `1`. In other words, we need to find a `w:list` element that looks something like this (whose `w:ilfo` attribute's value is `1`):

```
<w:list w:ilfo="1">
  <w:ilst w:val="5"/>
</w:list>
```

Now that we've found the list definition, the next step is finding the "base list definition." We do that by looking at the value provided by the `w:ilst` element. In this case, it is referring to a base list definition identified by the number `5`. Recalling that the base list definitions are represented by `w:listDef` elements and that they precede

the w:list elements inside the w:lists element, we continue to search further back in our WordprocessingML document. Eventually, we find what we're looking for:

```
<w:listDef w:listDefId="5">
  ...
  <w:lvl w:ilvl="0">...</w:lvl>
  <w:lvl w:ilvl="1">...</w:lvl>
  <w:lvl w:ilvl="2">...</w:lvl>
  <w:lvl w:ilvl="3">...</w:lvl>
  <w:lvl w:ilvl="4">...</w:lvl>
  <w:lvl w:ilvl="5">...</w:lvl>
  <w:lvl w:ilvl="6">...</w:lvl>
  <w:lvl w:ilvl="7">...</w:lvl>
  <w:lvl w:ilvl="8">...</w:lvl>
</w:listDef>
```

The w:listDef element is identified by its w:listDefId attribute and contains one w:lvl element for each level of list nesting for which it defines formatting. While you can create base list definitions that define fewer levels without a problem, Word's built-in list styles define all nine levels of nesting. The content of the w:lvl element includes all kinds of formatting information, such as indentation, tab stops, the number to start on, number format, and bullet images.

Once Word finds the base list definition, with all its formatting information, it then applies the appropriate level's formatting to the paragraph, according to the value of the w:ilvl element that occurs in the paragraph's list properties. Thus, Word applies the level 0 list item formatting to our example paragraph above.

List Styles

An even more complex variation of this approach occurs is when list styles are used. Unlike paragraph, table, and character styles, which can be directly associated with paragraphs, tables, and runs (via the w:pStyle, w:tblStyle, and w:rStyle elements, respectively), list styles are *not* directly associated with paragraphs in WordprocessingML—there is not a corresponding element for direct list style references. For example, when an end user applies the built-in list style "1 / a / i" to a paragraph, the paragraph is effectively associated with a list definition, but it is not *directly* associated with the "1 / a / i" list style that was applied to it. The resulting WordprocessingML paragraph looks essentially no different from the example paragraph we looked at earlier. Here it is again (with the only difference here being that the w:ilfo element happens to refer to a list definition identified by the number 2):

```
<w:p>
  <w:pPr>
    <w:listPr>
      <w:ilvl w:val="0"/>
      <w:ilfo w:val="2"/>
    </w:listPr>
  </w:pPr>
  <w:r>
```

```
      <w:t>This is item one.</w:t>
    </w:r>
  </w:p>
```

This is what the WordprocessingML looks like when an end user applies a list style to a paragraph. Rather than being directly associated with the list style, the paragraph refers to a list definition using the `w:ilfo` element—no differently than when a list style is not involved. However, the list style association is still retained; it's just that you can't tell that from looking at the paragraph alone. The list style association only becomes evident when we start traversing the graph, and that's where things get complicated. First, the paragraph associates itself with the document's list definition (`w:list` element), identified by the value `2`:

```
<w:list w:ilfo="2">
  <w:ilst w:val="1"/>
</w:list>
```

The list definition, in turn, refers (via the `w:ilst` element) to a base list definition (`w:listDef` element) identified by the value `1`. So far, so good. Now, here is where a few extra levels of indirection appear. Whereas before we were done at this point (the base list definition contained all the formatting properties for each level of the list), now we're only halfway there. This time, the referenced base list definition doesn't contain any formatting properties (inside `w:lvl` elements) at all. Instead, it contains yet another reference—the `w:listStyleLink` element:

```
<w:listDef w:listDefId="1">
  <w:lsid w:val="27DC6005"/>
  <w:plt w:val="Multilevel"/>
  <w:tmpl w:val="0409001D"/>
  <w:listStyleLink w:val="1ai"/>
</w:listDef>
```

This `w:listDef` element refers, via its `w:listStyleLink` element, to a list style definition whose `w:styleId` attribute's value is `1ai`. This corresponds to the "1 / a / i" style that the end user applied. Here is the document's list style definition that it refers to:

```
<w:style w:type="list" w:styleId="1ai">
  <w:name w:val="Outline List 1"/>
  <wx:uiName wx:val="1 / a / i"/>
  <w:basedOn w:val="NoList"/>
  <w:rsid w:val="00283CEE"/>
  <w:pPr>
    <w:listPr>
      <w:ilfo w:val="1"/>
    </w:listPr>
  </w:pPr>
</w:style>
```

As you can see, the list style definition, in turn, contains a reference to yet another list definition (identified by the number `1`). Dizzy yet?

```
<w:list w:ilfo="1">
  <w:ilst w:val="0"/>
</w:list>
```

This list definition refers to yet another base list definition, identified by the number 0. Finally, we are home free, as this base list definition actually contains the list formatting properties Word needs in order to format each level of the list:

```
<w:listDef w:listDefId="0">
  <w:lsid w:val="1B850634"/>
  <w:plt w:val="Multilevel"/>
  <w:tmpl w:val="0409001D"/>
  <w:styleLink w:val="1ai"/>
  <w:lvl w:ilvl="0">
    <w:start w:val="1"/>
    <w:lvlText w:val="%1)"/>
    <w:lvlJc w:val="left"/>
    <w:pPr>
      <w:tabs>
        <w:tab w:val="list" w:pos="360"/>
      </w:tabs>
      <w:ind w:left="360" w:hanging="360"/>
    </w:pPr>
  </w:lvl>
  <w:lvl w:ilvl="1">
    ...
  </w:lvl>
  <w:lvl w:ilvl="2">
    ...
  </w:lvl>
  <w:lvl w:ilvl="3">
    ...
  </w:lvl>
  <w:lvl w:ilvl="4">
    ...
  </w:lvl>
    ...
  <w:lvl w:ilvl="5">
    ...
  </w:lvl>
  <w:lvl w:ilvl="6">
    ...
  </w:lvl>
  <w:lvl w:ilvl="7">
    ...
  </w:lvl>
  <w:lvl w:ilvl="8">
    ...
  </w:lvl>
</w:listDef>
```

In summary, `w:ilfo` refers to `w:list`, which refers to `w:listDef`, which refers to `w:style`, which refers to another `w:list`, which refers to another `w:listDef`. Home, sweet home. Oh yeah, and the last `w:listDef` refers back to the same `w:style` through an element called `w:styleLink` (which you can see in the last code snippet above)—thereby throwing in a little circularity for good measure.

Sections

A *section* in Word is an area or set of areas within a document, characterized by the same page settings, such as margin width, header and footer size, orientation, border, and print settings. These settings are accessible within the Word UI through the File → Page Setup… dialog, shown in Figure 2-19. Figure 2-19 also shows the five different kinds of section breaks you can insert into a document: "Continuous," "New column," "New page," "Even page," and "Odd page."

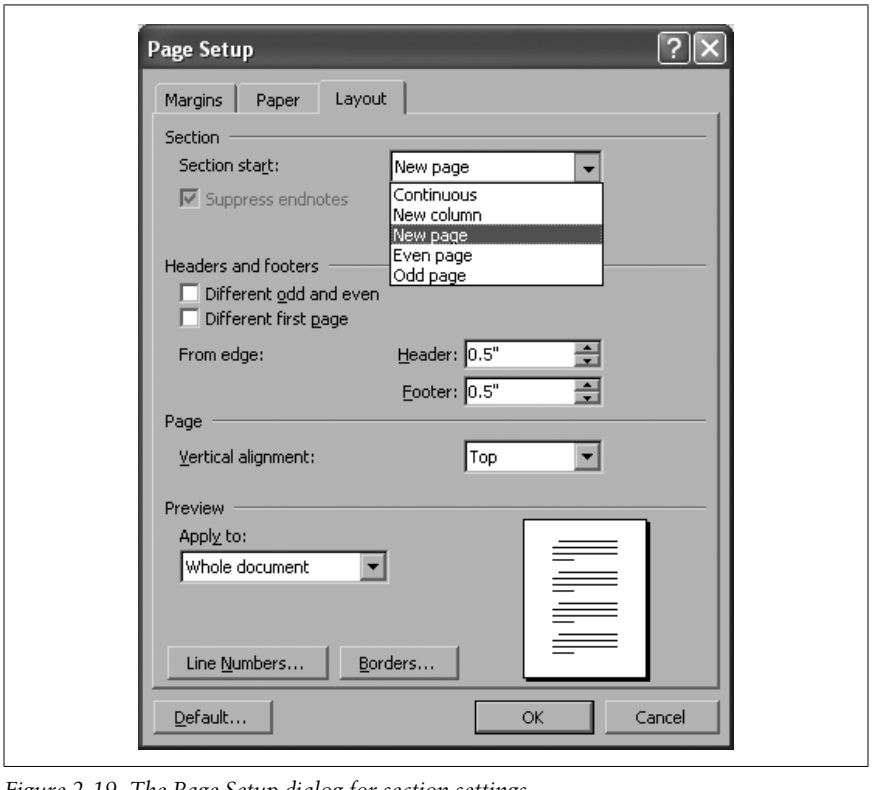


Figure 2-19. The Page Setup dialog for section settings

As mentioned previously, the structure of a Word document consists of one or more sections containing zero or more paragraphs containing zero or more characters. WordprocessingML, however, does not reflect that hierarchy exactly. In fact, there is no section container element in WordprocessingML proper. (As we'll see later in "Section Containers," the `wx:sect` element helps to fill this void by acting as a surrogate container, thereby aiding external processing.) Rather, sections are represented indirectly through the presence of *section breaks*. A section break is signified in WordprocessingML by the presence of a `w:sectPr` element inside the `w:pPr` element of the section's last paragraph. Example 2-8 shows the WordprocessingML for a document

that contains two section breaks, and therefore three sections. The w:sectPr elements are highlighted.

Example 2-8. Multiple sections in a document

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml"
  xml:space="preserve">
  <w:docPr>
    <w:view w:val="normal"/>
  </w:docPr>
  <w:body>
    <w:p>
      <w:pPr>
        <w:sectPr/>
      </w:pPr>
      <w:r>
        <w:t>First section</w:t>
      </w:r>
    </w:p>
    <w:p>
      <w:r>
        <w:t>Second section, first paragraph</w:t>
      </w:r>
    </w:p>
    <w:p>
      <w:pPr>
        <w:sectPr/>
      </w:pPr>
      <w:r>
        <w:t>Second section, second paragraph</w:t>
      </w:r>
    </w:p>
    <w:p>
      <w:r>
        <w:t>Third section, first paragraph</w:t>
      </w:r>
    </w:p>
    <w:p>
      <w:r>
        <w:t>Third section, second paragraph</w:t>
      </w:r>
    </w:p>
    <w:sectPr/>
  </w:body>
</w:wordDocument>
```

The first two w:sectPr elements in this document represent section breaks, because they each occur inside a w:pPr element. One thing to keep in mind about WordprocessingML's way of representing section breaks is that it can be deceiving. Specifically, the w:sectPr elements do not lexically divide the text of the document

according to its true section boundaries. For example, though from a first glance it may look as if the paragraph that says "Second section, second paragraph" belongs to the third and final section, that is not the case. It only looks that way because the `w:sectPr` element comes before the text of the paragraph in which it resides. This potential confusion is all the more reason to look forward to "Section Containers," later in this chapter.

The last `w:sectPr` element in Example 2-8 does *not* occur inside the `w:pPr` element. Rather, it is a child of `w:body`, following the last paragraph in the document. This is where Word always expects to see the final `w:sectPr` element of the document. It does not represent a section break; rather, its job is simply to apply properties to the final (and possibly only) section of the document. If it isn't there when Word loads the document, Word will add it. The presence of `w:sectPr` inside a `w:pPr` element always denotes a section break, but the presence of `w:sectPr` as the last child of the `w:body` element does not. It's important to keep this distinction in mind when generating WordprocessingML documents that have multiple sections.

Figure 2-20 shows what we see when Word opens the document in Example 2-8.

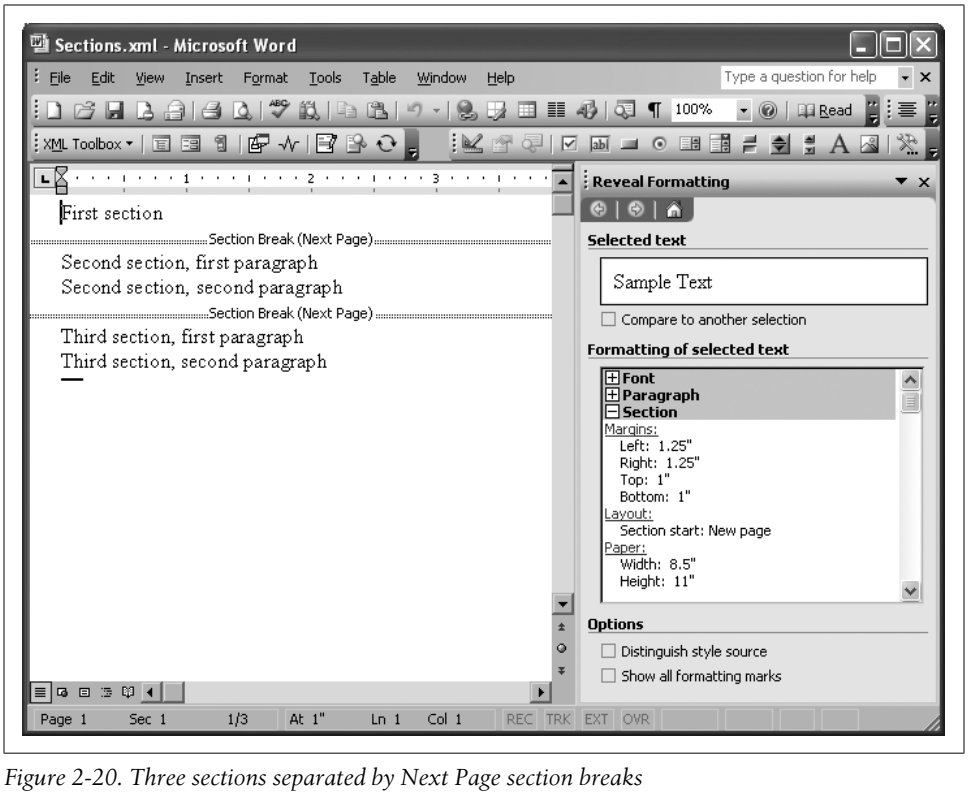


Figure 2-20. Three sections separated by Next Page section breaks

In the "Normal" view (which we see automatically, thanks to Example 2-8's use of the w:view element), all section breaks are visible. The first mystery of the empty w:sectPr section break element is answered: by default it stands for a "Next Page" break. We could have explicitly specified this in our document by using the w:type child element of w:sectPr, like this:

```
<w:sectPr>
  <w:type w:val="next-page"/>
</w:sectPr>
```

Besides next-page, the other possible values (corresponding to the drop-down menu options we saw in Figure 2-19) are next-column, continuous, even-page, and odd-page.

Of course, the insertion of section breaks is not the only responsibility of the w:sectPr element, which stands for "section properties." Its content model, after all, includes 21 possible element children, which collectively represent the settings a user can edit through the File → Page Setup... dialog. The properties specified inside the w:sectPr element apply to the section before the break that it represents (i.e., the section containing the paragraph with which the w:sectPr element is associated).

Normally, when you create a new blank document in Word, all of the page settings defined in the *Normal.dot* document template are copied into the document. These include margins, paper dimensions, vertical alignment, orientation, etc. But our hand-coded WordprocessingML document (Example 2-8) isn't "normal" in this sense. It was created outside of Word and specifies no page settings at all (as the w:sectPr elements are empty). Word gracefully handles this scenario when it loads the document by automatically inserting its application defaults for page settings. These default page settings are the same settings that are automatically copied into the *Normal.dot* template when Word is first installed, or when it is forced to create a new *Normal.dot* template.

We can see Word's application defaults for margins and paper size in the Reveal Formatting task pane in Figure 2-20. The underlying XML representation for these values looks something like this:

```
<w:sectPr>
  <w:pgSz w:w="12240" w:h="15840"/>
  <w:pgMar w:top="1440" w:right="1800" w:bottom="1440" w:left="1800"
           w:header="720" w:footer="720" w:gutter="0"/>
</w:sectPr>
```

All of the attribute values shown here are expressed in twips, or 1,440^{ths} of an inch. The w:pgSz element sets the page size to 8.5" x 11." The w:pgMar element sets the margin widths around the page: one inch on the top and bottom, and 1.25 inches on the right and left. It also sets header and footer areas, each with a height of half an inch.

If you need to override the default page settings for a particular section, you can simply specify your own values, using any of the other child elements of w:sectPr as necessary.

Proofing, Protection, and Annotation Markings

The `w:proofErr`, `w:permStart`, `w:permEnd`, and `aml:annotation` elements have shown up in various places so far without any real explanation. One thing they have in common is that they are all used to mark up ranges of text in a Word document: `w:proofErr` for spelling and grammar errors, `w:permStart` and `w:permEnd` for an editable area within a protected document, and `aml:annotation` for annotating comments, bookmarks, and revisions within a document.

A *range* is a span of text defined by a start character position and an end character position. The distinctive thing about ranges is that they can cross paragraph and section boundaries. From within a VBA application, a commonly used range is the range that corresponds to the user's current selection. Individual sentences and words are also examples of ranges that you can access through the Word object model, but they are not actually stored as part of the information in a Word document. Instead, such ranges are purely derivative and calculated on the fly, as the Word or VBA application demands. However, there are certain kinds of ranges that are necessary to be stored as part of the Word document itself. These include the various kinds of annotations you can make to a document without affecting its actual formatting, and markings that are automatically created, such as proofing marks for grammar and spelling.

There is a problem with representing such ranges of text in XML, because XML only allows you to represent a single tree. The problem of needing to represent multiple, overlapping hierarchies (which is what such annotations amount to) is commonly addressed in XML by inserting markers into the flow for the start and end positions of the range in question. This is exactly what Word does, too.

Figure 2-21 shows a paragraph in Word in which three ranges are overlapping, namely a document protection range, a grammar error range, and a comment annotation range.

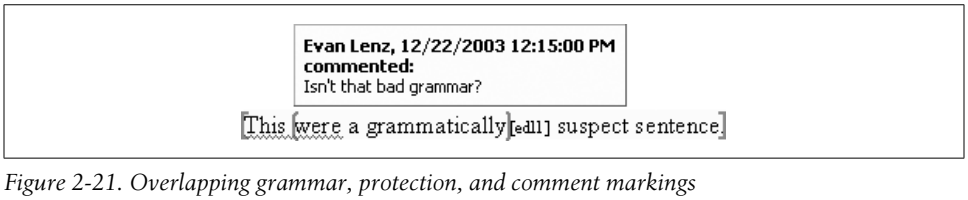


Figure 2-21. Overlapping grammar, protection, and comment markings

The outer brackets surrounding the entire sentence delineate the boundaries of an editing region with particular permissions; the inner parentheses delineate the boundaries of the text about which a comment was made; and the squiggly line under "This were" is a grammar error automatically recognized and flagged as such by Word. Example 2-9 shows the underlying WordprocessingML for this document excerpt, as output by Word. The start and end markers for each range, all of which are empty elements, are highlighted.

Example 2-9. Overlapping protection, proofing, and comment ranges

```
<w:p/>
<w:permStart w:id="0" w:edGrp="everyone"/>
<w:proofErr w:type="gramStart"/>
<w:p>
  <w:r>
    <w:t>This </w:t>
  </w:r>
  <aml:annotation aml:id="0" w:type="Word.Comment.Start"/>
  <w:r>
    <w:t>were</w:t>
  </w:r>
  <w:proofErr w:type="gramEnd"/>
  <w:r>
    <w:t> a grammatically</w:t>
  </w:r>
  <aml:annotation aml:id="0" w:type="Word.Comment.End"/>
  <w:r>
    <w:rPr>
      <w:rStyle w:val="CommentReference"/>
    </w:rPr>
    <aml:annotation aml:id="0" aml:author="Evan Lenz"
                    aml:createdate="2003-12-22T12:15:00Z"
                    w:type="Word.Comment" w:initials="edl">
      <aml:content>
        <w:p>
          <w:pPr>
            <w:pStyle w:val="CommentText"/>
          </w:pPr>
          <w:r>
            <w:rPr>
              <w:rStyle w:val="CommentReference"/>
            </w:rPr>
            <w:annotationRef/>
          </w:r>
          <w:r>
            <w:t>Isn't that bad grammar?</w:t>
          </w:r>
        </w:p>
      </aml:content>
    </aml:annotation>
  </w:r>
  <w:r>
    <w:t> suspect sentence.</w:t>
  </w:r>
  <w:permEnd w:id="0"/>
</w:p>
<w:p/>
```

This example illustrates the use of start and end markers to annotate ranges of text, regardless of whether they overlap each other or other elements, such as paragraphs. This explains, at long last, why these elements crop up in so many places in the

WordprocessingML schema. They need to occur as block-level elements as well as run-level elements. The w:permStart element occurs in this example in a block context, as a sibling of paragraphs, whereas the corresponding w:permEnd element occurs in a run context, before the end of the paragraph. Likewise, the first of the w:proofErr elements occurs as a block-level element, before the beginning of the paragraph, but the second w:proofErr element, which ends the range at the word "were," occurs as a run-level element.

Document protection

Now let's look at how each type of annotation works. The w:permStart and w:permEnd elements work together to identify a range of text that has a particular editing permission enabled. The w:id attribute of each element is used to associate the markers with each other. In this case, we know that they go together, because the w:id attribute value is 0 for both of them:

```
    <w:permStart w:id="0" w:edGrp="everyone"/>
...
      <w:permEnd w:id="0"/>
```

The value of the w:edGrp attribute denotes a group of people who can edit this region of text. In this case, the value is everyone, which means that there are no restrictions for this particular range. This is useful as a way of overriding a global document protection policy in which the rest of the document is off-limits for making changes. For more information on Word's document protection features, see Chapter 4.

Proof errors

The w:proofErr elements in Example 2-9 are used to identify the start and end points of a grammar error. The type of each marker is denoted by the w:type attribute:

```
    <w:proofErr w:type="gramStart"/>
...
      <w:proofErr w:type="gramEnd"/>
```

Since grammar, as well as spelling, errors cannot overlap each other, there is no need for an ID attribute to associate start and end markers with each other. Word knows that a grammar error ends at the first gramEnd marker that it finds after the gramStart marker. Spelling errors are represented in the same way, using the values of spellStart and spellEnd for the w:type attribute. Thus, the w:proofError's w:type attribute has four possible values:

```
gramStart
gramEnd
spellStart
spellEnd
```

Comments and other annotations

Example 2-9 also demonstrates how comments are represented in WordprocessingML. Every comment is represented using three separate `aml:annotation` elements. The three are associated with each other by having the same `aml:id` attribute value (`0` in Example 2-9's case). The first two `aml:annotation` elements are used to denote the start and end of the range that the comment is about:

```
<aml:annotation aml:id="0" w:type="Word.Comment.Start"/>
...
<aml:annotation aml:id="0" w:type="Word.Comment.End"/>
```

The `w:type` attribute values distinguish the start and end markers from each other: `Word.Comment.Start` and `Word.Comment.End`. The third `aml:annotation` element occurs inside a run (`w:r` element) that immediately follows the comment end marker:

```
<w:r>
  <w:rPr>
    <w:rStyle w:val="CommentReference"/>
  </w:rPr>
  ...
</w:r>
```

This run is associated with the `CommentReference` character style, a built-in style that is automatically inserted into the document when you insert a comment. So far, this looks like a normal run that might appear in the flow of document text. The content of the run, however, does not consist of normal document text. Instead, inside the run, we see the third and last `aml:annotation` element for this comment:

```
<aml:annotation aml:id="0" aml:author="Evan Lenz"
                aml:createdate="2003-12-22T12:15:00Z"
                w:type="Word.Comment" w:initials="edl">
  ...
</aml:annotation>
```

The `aml:id` attribute's value is `0`, which associates this annotation with the previous two. The `w:type` attribute is `Word.Comment`, which indicates that this element contains the actual content of the comment. The other three attributes contain metadata about the comment, including who made the comment, their initials, and the date and time they made it.

Inside the `aml:annotation` element is the `aml:content` element, which is used to contain the text of the comment:

```
<aml:content>
  <w:p>
    <w:pPr>
      <w:pStyle w:val="CommentText"/>
    </w:pPr>
    <w:r>
      <w:rPr>
        <w:rStyle w:val="CommentReference"/>
      </w:rPr>
```

```
        <w:annotationRef/>
      </w:r>
      <w:r>
        <w:t>Isn't that bad grammar?</w:t>
      </w:r>
    </w:p>
  </aml:content>
```

The comment text is represented using a sequence of Word paragraphs. These paragraphs are "out-of-band" in the sense that they do not occur in the normal flow of document text. After all, they ultimately occur inside a `w:r` element. A paragraph inside a run isn't normally allowed; it wouldn't make any sense. Only because of the intervening `aml:annotation` and `aml:content` elements is the `w:p` element allowed to occur as a descendant of a `w:r` element.

In addition to comments, the `aml:annotation` element is also used to represent bookmarks and revision markings (recorded when "Track Changes" is turned on). In each case, the type of annotation is identified by the value of the `w:type` attribute, which has these possible values:

```
Word.Insertion
Word.Deletion
Word.Formatting
Word.Bookmark.Start
Word.Bookmark.End
Word.Comment.Start
Word.Comment.End
Word.Insertion.Start
Word.Insertion.End
Word.Deletion.Start
Word.Deletion.End
Word.Comment
Word.Numbering
```

Auxiliary Hints in WordprocessingML

Until now, we've managed to stick to a pretty strict diet of elements and attributes from the WordprocessingML namespace, which has had times more pleasant than others. Now it's time to introduce a set of elements and attributes from another namespace that are designed purely for the purpose of making your life easier. That's right, you guessed it: the `wx` prefix is your friend (so long as it's mapped to the right namespace: `http://schemas.microsoft.com/office/word/2003/auxHint`).

There are quite a few contexts in which elements and attributes from the `wx` namespace appear in WordprocessingML documents saved by Word. We'll be focusing on some of the most significant of these: sections, sub-sections, and list

text, as well as formatting hints. These hints save consumers of WordprocessingML documents much grief and processing power that would otherwise be spent on things like traversing the links of a list definition, for example.

Again, elements and attributes in the wx namespace represent information that could be *useful to us* in handling WordprocessingML but that is of *no internal use to Word*. One implication of this distinction is that, while you may write applications that depend on their presence, it hardly ever makes sense to write applications that output elements or attributes in the wx namespace when generating WordprocessingML—except perhaps when doing incremental processing of an existing document such that you want to maintain the auxiliary information that originally came from Word. Even then, you're not really generating it; you're just forwarding it on.

Section Containers

Earlier in the chapter, in "Sections," we introduced WordprocessingML's non-intuitive way of representing a document's sections—how the presence of a w:sectPr element is implicitly interpreted to mean that the current paragraph is the last one in a section. Without a common container in which paragraphs of the same section are grouped together, it's not only counterintuitive but more difficult to process than it would otherwise be. Fortunately, the wx:sect element, which was introduced way back in Example 2-2, is Microsoft's answer to this problem. Whenever Word saves a document as XML, it doesn't just output the content of the w:body element. Instead, it groups the paragraphs and tables inside the body into wx:sect elements, corresponding to sections in the Word document.

To recognize the helpfulness of this feature, all we need to do is have Word open and to re-save the WordprocessingML document from Example 2-8. No longer is it so difficult to figure out where the section boundaries are:

```
<w:body>
  <wx:sect>
    <w:p>
      <w:pPr>
        <w:sectPr>
          <w:pgSz w:w="12240" w:h="15840"/>
          <w:pgMar w:top="1440" w:right="1800" w:bottom="1440"
                   w:left="1800" w:header="720" w:footer="720"
                   w:gutter="0"/>
          <w:cols w:space="720"/>
        </w:sectPr>
      </w:pPr>
      <w:r>
        <w:t>First section</w:t>
      </w:r>
    </w:p>
  </wx:sect>
  <wx:sect>
    <w:p>
```

```
      <w:r>
        <w:t>Second section, first paragraph</w:t>
      </w:r>
    </w:p>
    <w:p>
      <w:pPr>
        <w:sectPr>
          <w:pgSz w:w="12240" w:h="15840"/>
          <w:pgMar w:top="1440" w:right="1800" w:bottom="1440"
                   w:left="1800" w:header="720" w:footer="720"
                   w:gutter="0"/>
          <w:cols w:space="720"/>
        </w:sectPr>
      </w:pPr>
      <w:r>
        <w:t>Second section, second paragraph</w:t>
      </w:r>
    </w:p>
  </wx:sect>
  <wx:sect>
    <w:p>
      <w:r>
        <w:t>Third section, first paragraph</w:t>
      </w:r>
    </w:p>
    <w:p>
      <w:r>
        <w:t>Third section, second paragraph</w:t>
      </w:r>
    </w:p>
    <w:sectPr>
      <w:pgSz w:w="12240" w:h="15840"/>
        <w:pgMar w:top="1440" w:right="1800" w:bottom="1440"
                 w:left="1800" w:header="720" w:footer="720"
                 w:gutter="0"/>
      <w:cols w:space="720"/>
      <w:docGrid w:line-pitch="360"/>
    </w:sectPr>
  </wx:sect>
</w:body>
```

Note that there are three `wx:sect` elements, one for each section, and that the paragraphs in each section are clearly grouped together. As mentioned before, we could remove the start and end tags of each `wx:sect` element, and Word would process the document no differently. Conversely, the meaning of the document as far as Word is concerned is completely unaltered by the addition of the `wx:sect` element. It only considers the `w:sectPr` elements to determine where the sections are. The same old rules apply: `w:sectPr` elements inside `w:pPr` elements represent section breaks, but the last `w:sectPr` element (provided it follows the last paragraph inside the `w:body` element) does not represent a break, but instead simply contains the properties of the last section.

An example using XPath can help demonstrate how the wx:sect element enables easier processing of WordprocessingML documents outside of Word. If we were to write an XPath expression to select all of the paragraphs in, say, the third section, this would be easy (assuming the appropriate namespace bindings):

```
/w:wordDocument/w:body/wx:sect[3]/w:p
```

However, without the aid of the wx:sect element, the task is still possible but not as straightforward and certainly not as intuitive:

```
/w:wordDocument/w:body/w:p[count(preceding::w:sectPr)=2]
```

Clearly, the wx:sect element, though it may have looked cryptic at first sight, is a helpful aid to processing WordprocessingML documents as output by Word.

Outline Levels and Sub-Sections

Word has a special paragraph property that we didn't mention earlier: the outline level. As might be guessed, the outline level property has an effect on the display of a paragraph in Word's "Outline" view. Example paragraph styles for which an outline level is defined include all of Word's built-in Heading styles. In fact, it's no accident that the Outline view supports nine levels and that there are precisely nine Heading styles. Figure 2-22 shows how all of the Heading styles are displayed in Outline view, along with some body text on each rung of the ladder. The body text has no outline level specified, as is the case with most normal paragraphs. All of the Heading paragraphs, however, have the outline level corresponding to their name. Heading 1 has Outline Level 1, Heading 2 has Outline Level 2, etc.

Clearly, the document in Figure 2-22 follows a hierarchical structure (if rather deep). Many people author such hierarchically organized documents in Word. Indeed, the Heading styles in conjunction with Outline view give them incentives for doing so. Unfortunately none of that hierarchical structure made it into WordprocessingML, which remains wedded to the flat-list-of-paragraphs paradigm. Sure, you can make a document *look* like it's hierarchically structured, but underneath the covers it's just a sequence of paragraphs with various formatting properties applied. But all is not lost. Once again, the wx namespace comes to the rescue, in what is arguably the most useful element of all the auxiliary hints: the wx:sub-section element.

Whenever Word saves a WordprocessingML document that has an outline level specified on any of its paragraphs, then at least a one-level depth tree of wx:sub-section elements will be present in the output. Specifically, any time Word comes across a paragraph with an outline level, it establishes a new sub-section context equal in depth of sub-sections to the outline level of the paragraph. For example, if the outline level is 3, then the paragraph will be contained within three nested wx:sub-section elements. This stays in effect for following paragraphs either until it reaches another paragraph with an outline level, or it comes to the end of the section

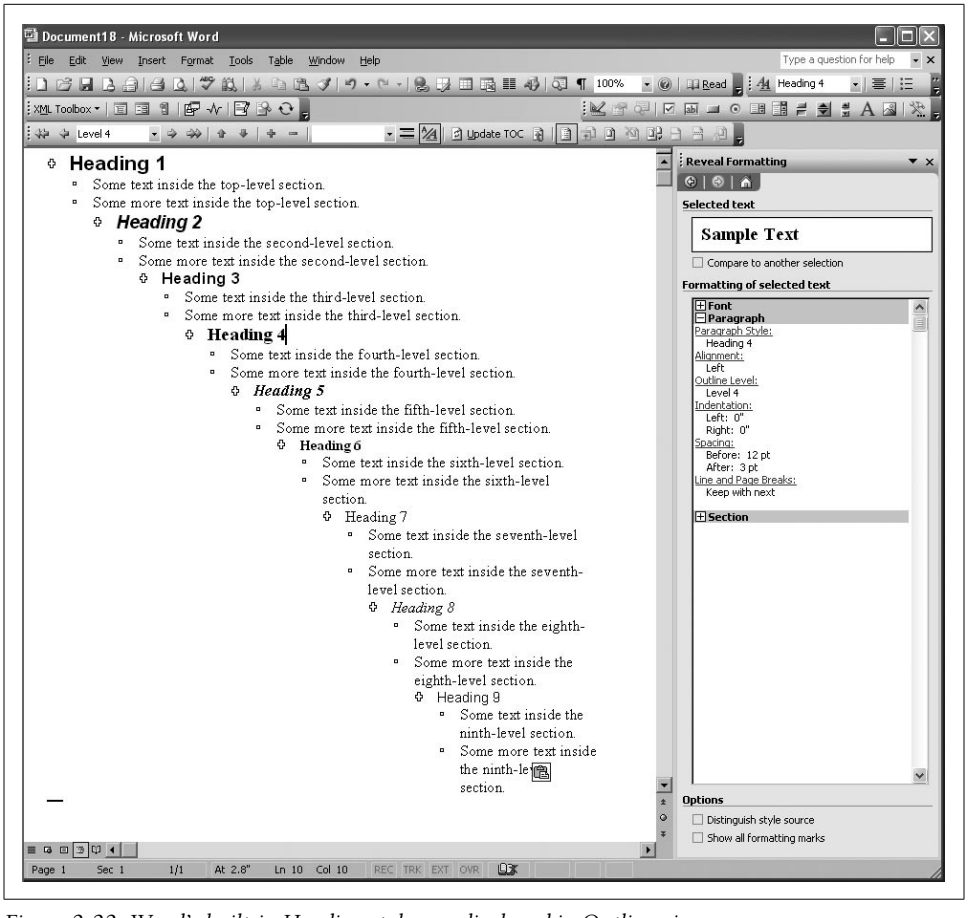


Figure 2-22. Word's built-in Heading styles, as displayed in Outline view

(in which case all of the `wx:sub-section` elements are closed). In the case of the document in Figure 2-22, it would output a structure similar to the following:

```
<wx:sub-section>
  Heading 1
  Body text
  Body text
  <wx:sub-section>
    Heading 2
    Body text
    Body text
    <wx:sub-section>
      Heading 3
      Body text
      Body text
      ...
    </wx:sub-section>
  </wx:sub-section>
</wx:sub-section>
```

You can achieve a similar effect with any custom paragraph style that you develop, simply by adding an outline level to the style definition. While using styles is probably the best way to achieve this effect, the use of styles isn't required. You can also apply the outline level property locally, as direct formatting on your paragraph. Example 2-10 finally demonstrates the syntax for the outline level property, as specified inside a paragraph's `w:pPr` element. This document contains a series of five paragraphs, two of which specify an outline level using the `w:outlineLvl` element, whose `w:val` attribute value must be between `0` and `8` (exposed as 1 through 9 in the Word UI).

Example 2-10. Setting outline levels locally

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml"
  xml:space="preserve">
  <w:body>
    <w:p>
      <w:pPr>
        <w:outlineLvl w:val="0"/>
      </w:pPr>
      <w:r><w:t>This is the top-level heading</w:t></w:r>
    </w:p>
    <w:p>
      <w:r><w:t>This is some text inside the top-level sub-
section.</w:t></w:r>
    </w:p>
    <w:p>
      <w:r><w:t>This is some more body text.</w:t></w:r>
    </w:p>
    <w:p>
      <w:pPr>
        <w:outlineLvl w:val="1"/>
      </w:pPr>
      <w:r><w:t>This is a second-level heading</w:t></w:r>
    </w:p>
    <w:p>
      <w:r><w:t>This is some body text under the second-level
heading.</w:t></w:r>
    </w:p>
  </w:body>
</w:wordDocument>
```

First, let's see what this document looks like when opened in Word. Figure 2-23 shows both the Normal view and the Outline view. The outline levels are completely invisible in the Normal view; the paragraphs look no different than any other plain, boring paragraph. Outline view is another story.

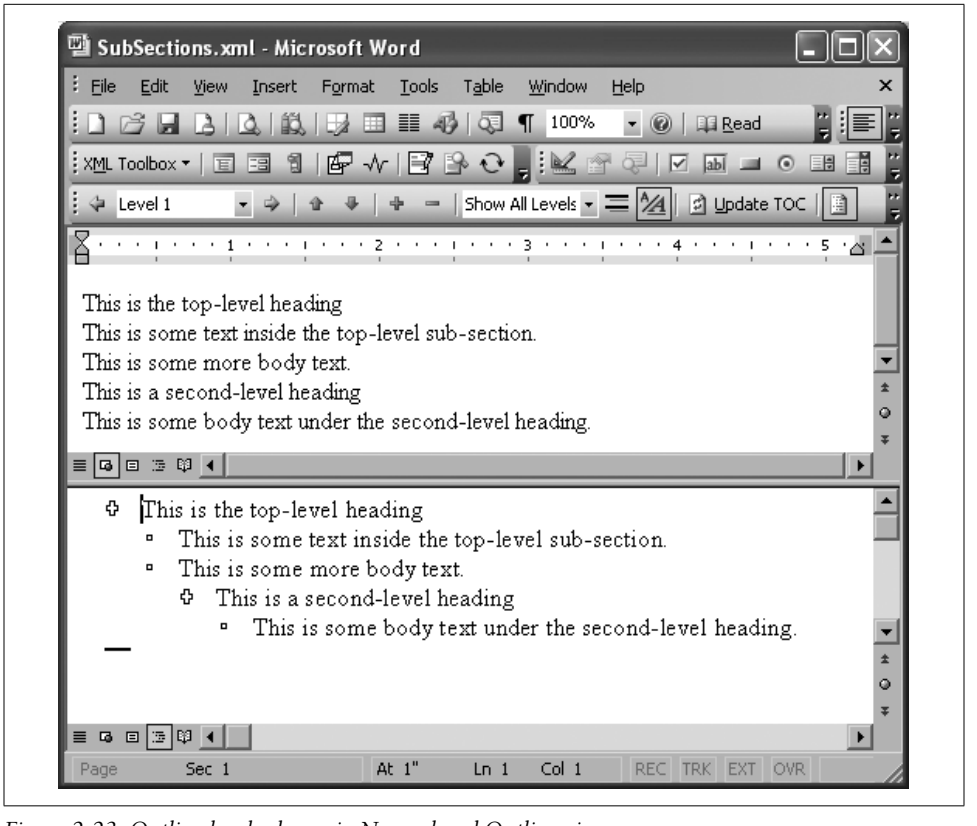


Figure 2-23. Outline levels shown in Normal and Outline views

Finally, we can see the `wx:sub-section` element in action by resaving the document as XML from within Word. Example 2-11 shows the body content excerpted from the WordprocessingML document as saved by Word.

Example 2-11. A document body with outline levels, when saved as XML in Word

```
<w:body>
  <wx:sect>
    <wx:sub-section>
      <w:p>
        <w:pPr>
          <w:outlineLvl w:val="0"/>
        </w:pPr>
        <w:r>
          <w:t>This is the top-level heading</w:t>
        </w:r>
      </w:p>
      <w:p>
        <w:r>
          <w:t>This is some text inside the top-level sub-section.</w:t>
        </w:r>
```

Example 2-11. A document body with outline levels, when saved as XML in Word (continued)

```
        </w:p>
        <w:p>
          <w:r>
            <w:t>This is some more body text.</w:t>
          </w:r>
        </w:p>
        <wx:sub-section>
          <w:p>
            <w:pPr>
              <w:outlineLvl w:val="1"/>
            </w:pPr>
            <w:r>
              <w:t>This is a second-level heading</w:t>
            </w:r>
          </w:p>
          <w:p>
            <w:r>
              <w:t>This is some body text under the second-level heading.</w:t>
            </w:r>
          </w:p>
          <w:sectPr>
            <w:pgSz w:w="12240" w:h="15840"/>
            <w:pgMar w:top="1440" w:right="1800" w:bottom="1440"
                     w:left="1800" w:header="720" w:footer="720"
                     w:gutter="0"/>
            <w:cols w:space="720"/>
            <w:docGrid w:line-pitch="360"/>
          </w:sectPr>
        </wx:sub-section>
      </wx:sub-section>
    </wx:sect>
  </w:body>
```

Example 2-11 demonstrates that Word interprets the outline levels to automatically structure the resulting WordprocessingML into sub-sections, using `wx:sub-section` elements, which are highlighted. Again, outline levels are most useful when they are associated with particular paragraph styles, rather than assigned directly to individual paragraphs (which, in the Word UI, can only be done in Outline View). Provided that the user applies styles in the order that they are intended, e.g., Heading 1 followed by Heading 2, etc., then the WordprocessingML that Word generates will be structured into sub-sections that reflect the true hierarchical structure of the document, rather than merely a flat sequence of paragraphs.

List Item Formatting Hints

Anything Word wants to provide in the way of making lists easier to process is certainly welcome. As we saw earlier in this chapter, lists in WordprocessingML are rather complicated to process. Generally, you can recognize the presence of a list

item by the presence of a w:listPr element inside a paragraph's w:pPr element. While that's a start, if you want to find out anything about how the list item is formatted, including even whether it's a "numbered" or "bulleted" list, you have to traverse a number of intra-document links. How many depends on whether and to what extent paragraph or list styles are involved.

As a matter of fact, Word does rather consistently save us this trouble by outputting the wx:t element inside a paragraph's w:listPr element. The wx:t element has three attributes: wx:val, wx:wTabBefore, and wx:wTabAfter. The wx:val attribute specifies the actual text used for the number or bullet point of this particular list item. The wx:wTabBefore is measured in twips and specifies the width of the tab preceding the line number. This usually corresponds to the indentation of the list item from the page's left margin. The wx:wTabAfter, on the other hand, calculates the distance, in twips, between the end of the text of the line number and the beginning of the editable area. It takes into consideration the font size and length of the line number itself. For example, consider the second list item of the simple list in Figure 2-24.

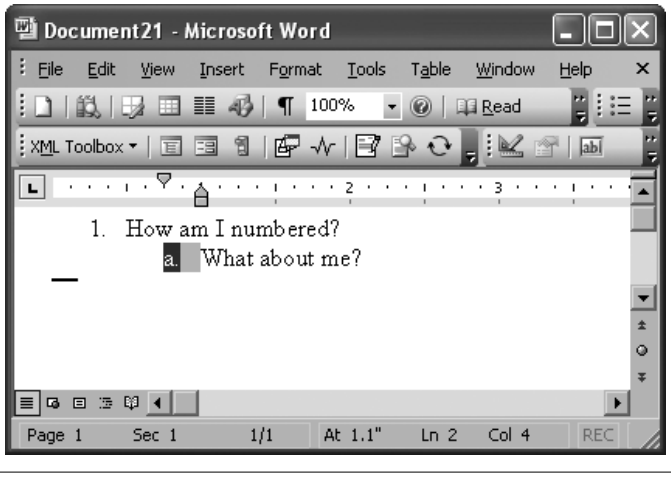


Figure 2-24. A simple list item

The hint as it resultantly appears in this paragraph's w:listPr element (inside its w:pPr element) is as follows:

```
<wx:t wx:val="a." wx:wTabBefore="1080" wx:wTabAfter="195" />
```

The wx:val attribute clearly relates that the line number text is "a." The wx:wTabBefore corresponds to the actual left indent of this paragraph, namely .75 inches, or 1080 twips. And the wx:wTabAfter attribute represents the distance between the "a." text and the contents of the list item—in other words, the gray, highlighted area following "a." in Figure 2-24.

More on Styles

Having come this far in the chapter, you should already know a few key aspects of how styles work in Word and WordprocessingML:

- A style is a grouping of property settings that can be applied as a unit.
- There are four kinds of styles: paragraph, character, table, and list.
- Styles are defined using `w:style` elements inside a WordprocessingML document's `w:styles` element.
- Paragraphs, runs, and tables can be directly associated with a style of the appropriate kind through the `w:pStyle`, `w:rStyle`, and `w:tblStyle` elements, respectively.

You should also know the basic syntax of the `w:style` element, and four aspects in particular:

- The `w:type` attribute, indicating the type of style defined here (`paragraph`, `character`, `table`, or `list`)
- The `w:default` attribute, indicating whether this style is the default style for its type
- The `w:styleId` attribute for intra-document references to this style
- The `w:name` element, indicating the style's primary name as exposed in the Word UI

In this section, we'll look at a few more aspects of how styles are defined, how default styles work (or don't), how to derive styles, and how style conflicts are resolved.

A Document's Styles

All styles that are used within a document must also be defined in the document. This effectively means that you can't leverage Word's built-in styles outside of Word; i.e., you can't simply refer to them by name. When a document uses a built-in Word style, Word makes a *copy of* the built-in style, rather than merely a reference to it. From that point forward, the style is part of the document and begins to exist independently of the built-in style from whence it came. To see a definitive list of the styles that are contained in your document, through the Word UI, select Tools → Templates and Add-Ins... and then click the Organizer... button. The styles listed on the left should correspond one-to-one with the `w:style` definitions in the WordprocessingML serialization of your document.

Default Styles

WordprocessingML's default style mechanism (using the w:default attribute) works well for paragraph and table styles. If you have w:p and w:tbl elements in your document that do not explicitly associate themselves with a style (with w:pStyle or w:tblStyle elements, respectively), then you can create sweeping formatting changes by simply changing the default style to a different paragraph or table style inside the w:styles element. You do this by setting the w:default attribute to on:

```
<w:style w:type="paragraph" w:default="on" w:styleId="MyParagraphStyle">
  <w:name w:val="My Paragraph Style"/>
  ...
</w:style>
```

On the other hand, the default style mechanism does *not* work for character styles and lists. If you try to specify a custom default character style, for example, Word will ignore it and will simply set the "Default Paragraph Font" character style as the default. For example, the w:default attribute shown here has no effect on Word's behavior:

```
<w:style w:type="character" w:default="on" w:styleId="MyCharacterStyle">
  <w:name w:val="My Character Style"/>
  ...
</w:style>
```

Effectively, this means that runs can only be associated with a character style *explicitly*—through the w:rStyle element, like this:

```
<w:r>
  <w:rPr>
    <w:rStyle w:val="MyCharacterStyle"/>
  </w:rPr>
  <w:t>This text is associated with a custom character style.</w:t>
</w:r>
```

Also, while you can freely customize the "Normal" paragraph style properties in your document, Word will discard any changes you attempt to make to the "Default Paragraph Font." Thus, there is no defaulting mechanism for associating runs with a particular character style (other than "Default Paragraph Font," which amounts to "no style"). In some respects, this is disconcerting, as it doesn't seem to match up with what WordprocessingML's syntax implicitly advertises. On the other hand, it reduces the possible combinations, thereby making the overall application of styles somewhat easier to think about.

The w:default attribute is essentially "syntax sugar," making it easy to create WordprocessingML documents without having to explicitly associate all of a document's paragraphs with a particular style (using a bunch of w:pStyle elements). Since the w:default attribute is merely syntax sugar and not part of Word's internal data structures, Word does not preserve your default style choices when it opens your document. Instead, Word always sets w:default="on" to the "Normal" style definition

when it outputs WordprocessingML, regardless of which paragraph style was the default in the WordprocessingML document it originally opened. This doesn't affect your document's formatting; it just means that the resulting WordprocessingML markup will be a little more verbose if most of your paragraphs don't use the "Normal" style. In that case, your paragraph style will be explicitly referenced via `w:pStyle` elements, rather than implicitly via the default style association:

```
<w:p>
  <w:pPr>
    <w:pStyle w:val="MyParagraphStyle"/>
  </w:pPr>
  <w:r>
    <w:t>This paragraph is explicitly associated with a para style.</w:t>
  </w:r>
</w:p>
```

Default Font Size for Paragraph Styles

There are two kinds of default font sizes in Word:

- 12 points, the font size of Word's built-in "Normal" style that gets automatically inserted into your document if you don't explicitly define it using a `w:style` element
- 10 points, the font size of a paragraph style definition (`w:style` element) that does not explicitly specify a font size using the `w:sz` element

We have already seen how the first default font size comes about. If you do not explicitly define the "Normal" paragraph style in a document, then Word automatically inserts its built-in "Normal" style, whose font size is 12 points (24 half-points). This scenario is exactly what we saw in Examples 2-1 and 2-2.

However, when you *do* define a paragraph style but do not explicitly specify the font size (using the `w:sz` element), then the font size of your paragraph style defaults to 10 points (20 half-points). For this reason, if you do define the "Normal" style in your document but without specifying a font size, then you will get a different result than if you didn't define the style at all. Specifically, the font size of your document's text will be 10 points, rather than 12 points. Example 2-12 shows a document that differs from Example 2-1 only in that it contains an empty definition for the "Normal" paragraph style (as identified by the `w:name` element).

Example 2-12. Defining the "Normal" style without specifying a font size

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml">

  <w:styles>
    <w:style w:type="paragraph" w:default="on">
```

Example 2-12. Defining the "Normal" style without specifying a font size (continued)

```
      <w:name w:val="Normal"/>
    </w:style>
  </w:styles>

  <w:body>
    <w:p>
      <w:r>
        <w:t>Hello, World!</w:t>
      </w:r>
    </w:p>
  </w:body>
</w:wordDocument>
```

When Word opens this document, the text "Hello, World!" is displayed in 10-point, rather than 12-point, Times New Roman. This is because you defined the style in your document, but did not include a `w:sz` element (inside a `w:rPr` element):

```
<w:style w:type="paragraph" w:default="on">
  <w:name w:val="Normal"/>
</w:style>
```

Word interprets such a paragraph style definition (regardless of whether it's the "Normal" style or some other paragraph style) as having a font size of 10 points. The above definition is equivalent to this one, where the font size of 20 half-points is explicitly specified:

```
<w:style w:type="paragraph" w:default="on">
  <w:name w:val="Normal"/>
  <w:rPr>
    <w:sz w:val="20"/>
  </w:rPr>
</w:style>
```

The only case where a paragraph style's font size could be different than 10 points without explicitly specifying a font size is when the style is derived from another paragraph style that has a different font size. As long as both the `w:basedOn` and `w:sz` elements are absent, then you can be sure that the paragraph style's font size is 10 points. But if there *is* a `w:basedOn` element and no `w:sz` element, then you would have to look at the base style to determine what the font size is.

So, what is the default font size for a WordprocessingML document? The answer is: it depends on what you mean by "default font size." If you're talking about the font size of Word's built-in "Normal" style, the answer is 12 points. If you're talking about the default font size of paragraph style definitions, the answer is 10 points.

Derived Styles

> In MS Word, editing styles is like drilling for oil in the Mariana Trench: by the time you finish the descent through the menus, you're down so deep that you can get the bends trying to remember what you started to do.
>
> —*http://www.linuxjournal.com/article.php?sid=7120*

One of the most powerful aspects of styles is the ability to base one style on another (in WordprocessingML, using the `w:basedOn` element), overriding individual properties as necessary. We'll see a couple examples of derived styles later in "A Pop Quiz," but the basic syntax looks like this:

```
<w:style w:type="paragraph" w:styleId="MyDerivedStyle">
  <w:name w:val="My Derived Style"/>
  <w:basedOn w:val="MyBaseStyle"/>
  <!-- formatting information -->
</w:style>
```

Using style derivation, you can base all of your paragraph styles, for example, on a base "Normal" style. Then, if you want to make a global change to all of your styles, such as font size, you need only make the change in one place—in the base style. This, of course, assumes that none of your derived styles override the base style's font size setting. Unfortunately, the Word UI doesn't give any visual clues as to when a particular property of a derived style is merely inherited from the base style or whether it is hard-wired to the style itself. This can make for some bewildering behavior.

For example, say your document has a base style called "Normal," from which a number of different styles have been derived, all of which merely inherit the font size property from "Normal." Whenever you update the font size of the "Normal" style, all of the derived styles' font sizes will be updated accordingly. So far, so good. But suppose you now want to derive another style, called "Code," that you know upfront should *always* be set to a font size of 9 points, regardless of any changes to the base "Normal" style's font size. This is the tricky part. When you first create the "Code" style and select a font size of 9 points, whether that size will end up being hard-wired to the "Code" style (which is what you want) or whether the "Code" style will merely inherit the font size from "Normal" (not what you want) completely depends on what the font size of "Normal" happens to be at the time you create the style. That's because Word gives you no way of telling it to hard-wire the font size to this style. Instead, it makes an assumption based on the current state of the base style. It assumes, in this case, that if the "Normal" font size is 9 points and you select 9 points when creating the "Code" style, you must want "Code" to always be the same size as "Normal." The only way to get around this is to temporarily change the "Normal" style's font size to something other than 9 points, and then create the new style, changing it back after you're done.

The introduction of WordprocessingML can largely alleviate this problem. By saving as XML, you get a readable (assuming you've pretty-printed), as well as editable, dump of all of your document's style definitions, removing once and for all any doubt about which of a style's properties are inherited and which are hard-wired to the style.

Resolving Conflicts

A given piece of text's formatting information can come from several different places, which raises the question of how conflicts are handled. Even after resolving a document's derived-style inheritance tree, there are still plenty of potential ambiguities, since you still have direct formatting, paragraph styles, and character styles to consider. Understanding how these all interact is fundamental to an understanding of WordprocessingML. In this section, we'll look at how potential conflicts are resolved—first for paragraph properties and then for font properties.

Paragraph property conflicts

A given paragraph can have paragraph properties applied to it in two ways:

- Through the associated paragraph style
- Through direct formatting

There is a simple rule for resolving conflicts between these two ways of applying paragraph properties: *direct formatting always wins*. For example, you can be sure that the following paragraph will be centered, without ever having to look at the `MyParagraphStyle` definition:

```
<w:p>
  <w:pPr>
    <w:pStyle w:val="MyParagraphStyle"/>
    <w:jc w:val="center"/>
  </w:pPr>
  <w:r>
    <w:t>This text is centered, regardless of what the associated paragraph
style says.</w:t>
  </w:r>
</w:p>
```

The `w:jc` element in the above snippet is an example of direct paragraph formatting. It is a paragraph property that is applied locally to this specific paragraph, as opposed to being part of a style definition. Any time you see a property setting applied within a local `w:pPr` element, you can be sure that it will take precedence over any conflicting settings in the associated paragraph style.

Font property conflicts

While paragraph properties can only be applied in two ways, font properties can be applied to a given piece of text in *three* different ways:

- Through the associated paragraph style
- Through the associated character style
- Through direct formatting

For font properties, as with paragraph properties, *direct formatting always wins*. For example, you can be sure that the run of text in the snippet below is italic and not bold without even looking at the `MyParagraphStyle` or `MyCharacterStyle` definitions:

```
<w:p>
  <w:pPr>
    <w:pStyle w:val="MyParagraphStyle"/>
  </w:pPr>
  <w:r>
    <w:rPr>
      <w:rStyle w:val="MyCharacterStyle"/>
      <w:i/>
      <w:b w:val="off"/>
    </w:rPr>
    <w:t>This text is italic and not bold, regardless of what the associated
paragraph and character styles say.</w:t>
  </w:r>
</w:p>
```

The `w:i` and `w:b` elements in the above snippet are examples of direct font formatting. They are font properties applied locally to this specific run, as opposed to being part of a style definition. Any time you see a property setting applied within a local `w:rPr` element, you can be sure that it will take precedence over any conflicting settings in the associated paragraph or character styles.

While the rule that "direct formatting always wins" is sufficient to resolve all potential paragraph property conflicts, it does *not* resolve all potential font property conflicts. Resolving font properties is a more complex problem, because—unlike paragraph properties—font properties can be defined in both the character style *and* the paragraph style. What happens when font property settings conflict between a run's associated paragraph and character styles?

To help answer this question, let's consider the different kinds of font properties that can be applied. Word's font properties can be classified into two categories:

- On/off properties
- Everything else (multi-valued properties)

Examples of on/off properties are bold (`w:b`), italic (`w:i`), all caps (`w:caps`), and strikethrough (`w:strike`). Examples of the other, multi-valued properties include

underline (`w:u`), font (`w:rFonts`), font size (`w:sz`), and font color (`w:color`). For multi-valued properties, the rule is simple: *the character style takes precedence*.

For the on/off properties, the rule isn't about which style has precedence; the paragraph and character styles are considered equally. Instead, the rule is about how their settings are *merged*. Here's the rule: *a given property is turned on only when it is turned on in one style but not the other*.

To help make this more explicit, Table 2-1 shows all four possible combinations for a particular on/off property and the effective result of each.

Table 2-1. How on/off font properties are merged between a paragraph and character style

Paragraph style	Character style	Result
Off	Off	Off
Off	On	On
On	Off	On
On	On	Off

Table 2-1 is essentially a truth table. The first two columns contain the inputs and the third column contains the XOR ("exclusive or") result. If you imagine representing a style's on/off property settings as a binary number (a series of 0s and 1s), then to compute the final result, you would apply an XOR bitmask to the two binary numbers, i.e., to the paragraph and character styles. That is in fact what Word does.

Let's bring this back down to earth with an example. At one time or another, you may have noticed Word's behavior when you applied an italicized character style to text within an italicized paragraph. Rather than keeping the text italic, this action had the opposite effect: the resulting text was *not* italicized. You may have thought that Word was just being clever about interpreting your intentions. After all, if you wanted to emphasize a particular word in a paragraph that is already emphasized as a whole, how else would Word do it? In reality, Word was just following the above rule. Since the italic property was turned on in both the paragraph and the character styles, they effectively cancelled each other out, and the result was not italicized. Example 2-13 illustrates exactly this scenario.

Example 2-13. Turning italics off using a character style

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml">

  <w:styles>
    <w:style w:type="paragraph" w:styleId="EmphasizedParagraph">
      <w:name w:val="Emphasized Paragraph"/>
      <w:rPr>
        <w:i/>
```

Example 2-13. Turning italics off using a character style (continued)

```
        <w:b/>
      </w:rPr>
    </w:style>
    <w:style w:type="character" w:styleId="Emphasis">
      <w:name w:val="Emphasis"/>
      <w:rPr>
        <w:i/>
        <w:b w:val="off"/>
      </w:rPr>
    </w:style>
  </w:styles>
  <w:body>
    <w:p>
      <w:pPr>
        <w:pStyle w:val="EmphasizedParagraph"/>
      </w:pPr>
      <w:r>
        <w:t>Most of this paragraph is italicized, but </w:t>
      </w:r>
      <w:r>
        <w:rPr>
          <w:rStyle w:val="Emphasis"/>
        </w:rPr>
        <w:t>this part is not.</w:t>
      </w:r>
    </w:p>
  </w:body>
</w:wordDocument>
```

Figure 2-25 shows what this document looks like when opened in Word. The last part of the paragraph is not italicized. The "Reveal Formatting" task pane shows that the "Emphasis" style contributes the "Not Italic" effect. In any other (non-italicized) paragraph, the "Emphasis" style would have exactly the opposite effect.

The other thing to note about this example is that the entire paragraph is rendered bold, even though the "Emphasis" character style explicitly tries to turn bold off:

```
<w:b w:val="off"/>
```

This behavior is consistent with the rule that if *either* (but not both) of the paragraph and character styles turns a property on, then that property will effectively be turned on. The only times that explicitly turning a property off will have an overriding effect are either when you are inheriting from another style (using the `w:basedOn` element) or when you are applying direct formatting (using a local `w:rPr` element). In those cases, to turn a property off, you explicitly turn it off. In contrast, if you want to use a character style to turn a property off, you have to do the counter-intuitive thing: you turn the property *on*.

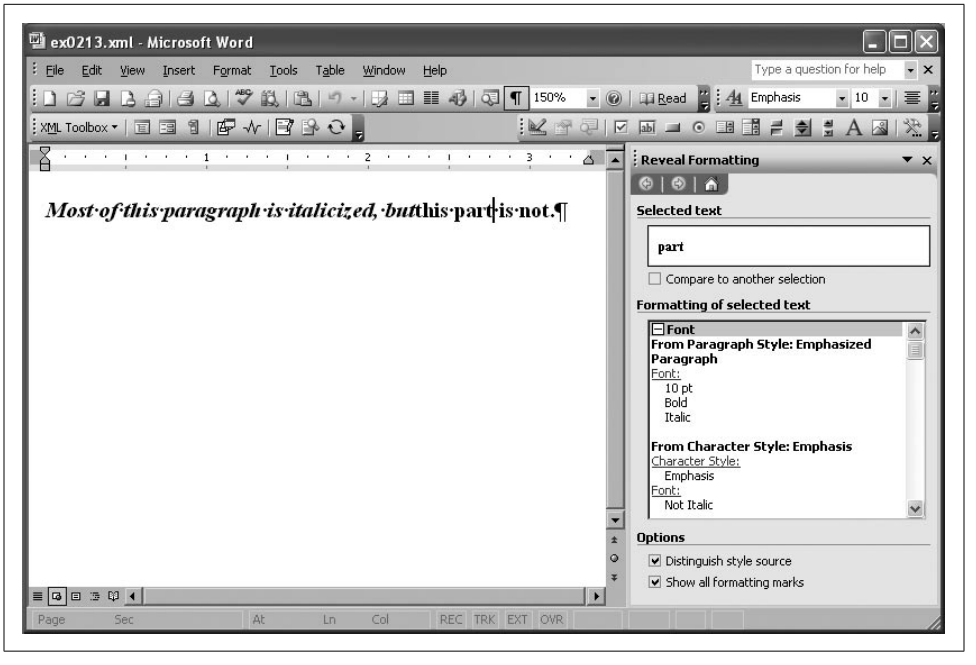


Figure 2-25. How Word renders Example 2-13

For most on/off font properties, explicitly turning them off in a character style has no effect. However, there are a few exceptions to this rule, including the `w:dstrike` (double strikethrough), `w:noProof` (ignore spelling/grammar errors for this run), and `w:rtl` (right-to-left reading order) elements. Though each of these are on/off properties, they are interpreted more like their multi-valued counterparts, i.e., they have an overriding effect. The character style takes precedence over the paragraph style setting. For example, if a run's paragraph style turns double strikethrough on, but its character style definition includes `<w:dstrike w:val="off"/>`, then it will be rendered *without* the double strikethrough.

A Pop Quiz

Now it's time for a pop quiz. Considering what you now know about default styles, derived styles, direct formatting, and how paragraph and character styles interact, try to figure out what formatting the runs in Example 2-14 have. There are two runs of text, separated by a soft line break. For each run, ask yourself: Is it bold? Is it italic? Is it both?

Example 2-14. What formatting do I have?

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml"
  xml:space="preserve">

  <w:styles>
    <w:style w:styleId="BaseParagraphStyle" w:type="paragraph">
      <w:name w:val="Base Paragraph Style"/>
      <w:rPr>
        <w:b/>
        <w:i/>
      </w:rPr>
    </w:style>
    <w:style w:styleId="DerivedParagraphStyle" w:type="paragraph"
             w:default="on">
      <w:name w:val="Derived Paragraph Style"/>
      <w:basedOn w:val="BaseParagraphStyle"/>
      <w:rPr>
        <w:i w:val="off"/>
      </w:rPr>
    </w:style>
    <w:style w:styleId="BaseCharacterStyle" w:type="character">
      <w:name w:val="Base Character Style"/>
      <w:rPr>
        <w:i/>
      </w:rPr>
    </w:style>
    <w:style w:styleId="DerivedCharacterStyle" w:type="character">
      <w:name w:val="Derived Character Style"/>
      <w:basedOn w:val="BaseCharacterStyle"/>
      <w:rPr>
        <w:b/>
      </w:rPr>
    </w:style>
  </w:styles>

  <w:body>
    <w:p>
      <w:r>
        <w:rPr>
          <w:rStyle w:val="DerivedCharacterStyle"/>
          <w:i w:val="off"/>
        </w:rPr>
        <w:t>What formatting do I have?</w:t>
      </w:r>
      <w:r>
        <w:rPr>
          <w:rStyle w:val="DerivedCharacterStyle"/>
        </w:rPr>
        <w:br/>
        <w:t>And what formatting do I have?</w:t>
      </w:r>
```

Example 2-14. What formatting do I have? (continued)

```
    </w:p>
  </w:body>

</w:wordDocument>
```

Okay, let's figure it out. The first thing we can do is determine what styles are used in the document. The document's one paragraph doesn't explicitly associate itself with a paragraph style; it has no `w:pStyle` element. Therefore, it adopts whatever the document's default paragraph style is. Looking at the document's style definitions, we see that the "Derived Paragraph Style" definition is the default one:

```
<w:style w:styleId="DerivedParagraphStyle" w:type="paragraph"
         w:default="on">
  <w:name w:val="Derived Paragraph Style"/>
```

Inside the document's paragraph are two runs, both of which are associated with the "Derived Character Style" definition, using the `w:rStyle` element:

```
<w:rStyle w:val="DerivedCharacterStyle"/>
```

The next thing we need to do is resolve the style derivations to determine exactly what formatting properties are applied by each derived style. The "Base Paragraph Style" turns bold and italic on:

```
<w:b/>
<w:i/>
```

But the "Derived Paragraph Style" turns italic off:

```
<w:i w:val="off"/>
```

Therefore, our document's default paragraph style consists of one font property setting: bold.

The "Base Character Style" turns italic on, and the "Derived Character Style" turns bold on. Nothing is overridden. Therefore, the character style associated with our document's two runs has two font property settings: bold and italic.

Next, we look to the body of the document itself. The first run explicitly turns italic off, so we know that the first run will not be italicized, as direct formatting always has the final word:

```
<w:r>
  <w:rPr>
    <w:rStyle w:val="DerivedCharacterStyle"/>
    <w:i w:val="off"/>
  </w:rPr>
  <w:t>What formatting do I have?</w:t>
</w:r>
```

The next question is whether this run is bold or not. Since, as we've seen, both the fully resolved paragraph style and the fully resolved character style turn bold on, that means bold will effectively be turned *off*. This is in keeping with the rule that a

property is on only if one *but not both* styles turns it on. Thus, the first run is rendered in neither bold nor italic type.

The second run is the same as the first, except that italic is not explicitly turned off via direct formatting. In fact, there is no direct formatting:

```
<w:r>
  <w:rPr>
    <w:rStyle w:val="DerivedCharacterStyle"/>
  </w:rPr>
  <w:br/>
  <w:t>And what formatting do I have?</w:t>
</w:r>
```

We've already seen that the paragraph and character styles' bold settings cancel each other out, so the remaining question is whether this run is italicized or not. Since the character style turns italic on but the paragraph style does not, that means that italic will indeed be turned on, because it is turned on in one but not both of the paragraph and character styles. Figure 2-26 shows the result of opening this document in Word (with paragraph marks turned on).

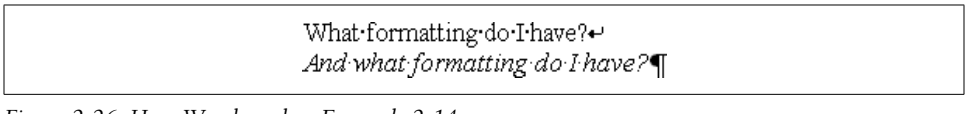


Figure 2-26. How Word renders Example 2-14

Dummy Styles

A common advantage of using styles in Word is that they can help to enforce consistency of presentation throughout a document. However, for an XML-oriented user, styles may at first seem to provide yet an additional advantage, especially when they are defined in a template: a way to separate presentation from content in Word. In a limited way they do, because within a document, the style definitions and the content are in distinct places, and changes to a document's style are propagated to all instances of that style throughout the document. However, styles defined externally in a template, rather than remaining separate from a document, are copied into the document when the template is first attached. (This ensures that a document will display uniformly on different machines without requiring all users to have access to the originally attached template.) When a template is attached, all of its styles are copied into the document, and the template's role is essentially over. The document does retain a loose association with the template (as represented by the `w:attachedTemplate` element), but for all practical purposes the template is no longer needed—*unless* you elect to set the document's "Automatically update document styles" option to true, as shown in Figure 2-27, in the "Templates and Add-ins" dialog box.

WordprocessingML represents this setting through the presence of an empty `w:linkStyles` element inside the `w:docPr` element (short for `<w:linkStyles w:val="on"/>`

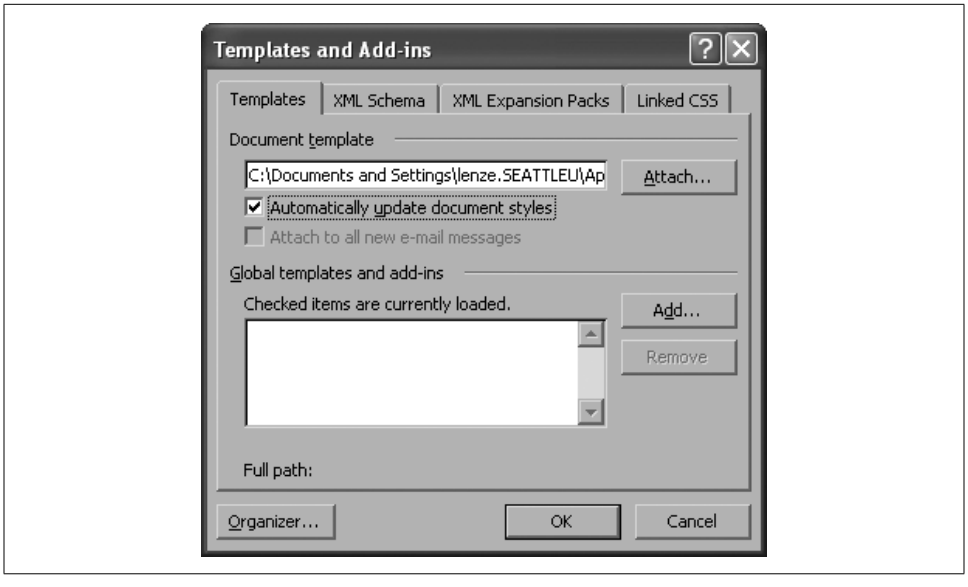


Figure 2-27. The "Automatically update document styles" checkbox

because on is the default attribute value for `w:val`). When `w:linkStyles` is present, the `w:attachedTemplate` reference gains new meaning. The next time Word opens the document, it immediately copies all the style definitions within that template into the document once again, replacing any style definition that has the same name as a style defined in the template. As long as this option is set, Word will continue to update the styles in the document, whenever the document is opened.

There is a practical implication for the XML developer writing XSLT stylesheets to, say, generate Word document reports. Provided that the user who opens the target Word document has access to its attached template, then styles in the template can effectively be referenced without duplicating the entire style definition.

As long as the `w:linkStyles` option is set, you can rely on Word to supply all the style definitions for you as soon as it opens the document. This greatly simplifies programs (such as XSLT stylesheets) that generate WordprocessingML documents that use styles already defined in a template.

Remember that to use any style within a document, it always must be declared in the top-level `w:styles` element. You can't just refer to a style from inside the `w:body` element, even if it's a built-in style. If you try to use a style without declaring it, the style reference will be ignored and discarded. So you must declare the style, giving it an arbitrary internal ID (using the `w:styleId` attribute) for reference from within the document body. (The `w:styleId` attribute's value can be any string.) Then, to have Word replace a dummy style definition for you, you must additionally ensure all three of the following:

- The w:linkStyles element is present inside the w:docPr element
- The value of the w:name element's w:val attribute is the same as the name of a style declared in the attached template
- The attached template is available to the user who initially opens the document

Example 2-15 shows a minimal WordprocessingML document created by hand that uses the Code,x style defined in the O'Reilly Word template. Rather than defining the entire style in all its verboseness, along with the ripe potential for error that would entail, this WordprocessingML document simply declares the style, using a dummy definition that includes nothing other than the w:name element, which identifies it as the Code,x style. The only paragraph of the document then is assigned that style using the w:pStyle element inside the w:pPr element. Thanks to the presence of the w:linkStyles element, the complete style definition for Code,x is inserted automatically (along with all of the template's other styles), as soon as Word opens the document.

Example 2-15. Replacing dummy style definitions via w:linkStyles

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml">
  <w:styles>
    <w:style w:styleId="Code">
      <w:name w:val="Code,x"/>
    </w:style>
  </w:styles>

  <w:docPr>
    <w:attachedTemplate w:val="C:\Documents and Settings\lenze.SEATTLEU\Application Data\
Microsoft\Templates\ora.dot"/>
    <w:linkStyles/>
  </w:docPr>

  <w:body>
    <w:p>
      <w:pPr>
        <w:pStyle w:val="Code"/>
      </w:pPr>
      <w:r>
        <w:t>This is a code example.</w:t>
      </w:r>
    </w:p>
  </w:body>
</w:wordDocument>
```

Word will always output complete style definitions in the WordprocessingML it creates. Accordingly, this technique shouldn't be thought of as enabling the separation of presentation and content, but rather as a one-time macro of sorts for getting Word to put all the styles in your document for you. Indeed, this describes the basic role that template attachment plays in the first place.

Linked Styles

The w:link element, when present in a paragraph style definition, represents a link to a character style. Conversely, when present in a character style definition, the w:link element represents a link to a paragraph style. Only paragraph and character styles can be linked to each other. The key characteristic of a paragraph-character style link is that the two styles are exposed in the primary Word UI as a single style, using the name of the paragraph style. Also, changes to the character properties of one style are automatically propagated to the other. Word automatically creates a linked character style when a user applies a paragraph style to only a portion of a paragraph, rather than to a paragraph as a whole. The alternative would be to throw an error, chastising the user for trying to use a paragraph style on anything but a complete paragraph. That being potentially bad business, Word instead gracefully falls back and automatically creates a new character style by copying all of the paragraph style's character properties into the newly created style. Thus a linked character style is born.

Figure 2-28 shows the creation of a linked character style named "Heading 1 Char." Word automatically creates the style, because the user has tried to apply the "Heading 1" style to only a portion of a paragraph (the word "partial"). At the top of the screen, the style is still listed simply as "Heading 1," though the Reveal Formatting task pane and the Style dialog box both reveal the distinction between "Heading 1" and "Heading 1 Char."

The style definitions in the resulting WordprocessingML are shown below, with the w:link elements highlighted:

```
<w:style w:type="paragraph" w:styleId="Heading1">
  <w:name w:val="heading 1"/>
  <wx:uiName wx:val="Heading 1"/>
  <w:basedOn w:val="Normal"/>
  <w:next w:val="Normal"/>
  <w:link w:val="Heading1Char"/>
  <w:rsid w:val="00B33163"/>
  <w:pPr>
    <w:pStyle w:val="Heading1"/>
    <w:keepNext/>
    <w:spacing w:before="240" w:after="60"/>
    <w:outlineLvl w:val="0"/>
  </w:pPr>
  <w:rPr>
    <w:rFonts w:ascii="Arial" w:h-ansi="Arial" w:cs="Arial"/>
    <wx:font wx:val="Arial"/>
    <w:b/>
    <w:b-cs/>
    <w:kern w:val="32"/>
    <w:sz w:val="32"/>
    <w:sz-cs w:val="32"/>
  </w:rPr>
</w:style>
<w:style w:type="character" w:styleId="Heading1Char">
  <w:name w:val="Heading 1 Char"/>
```

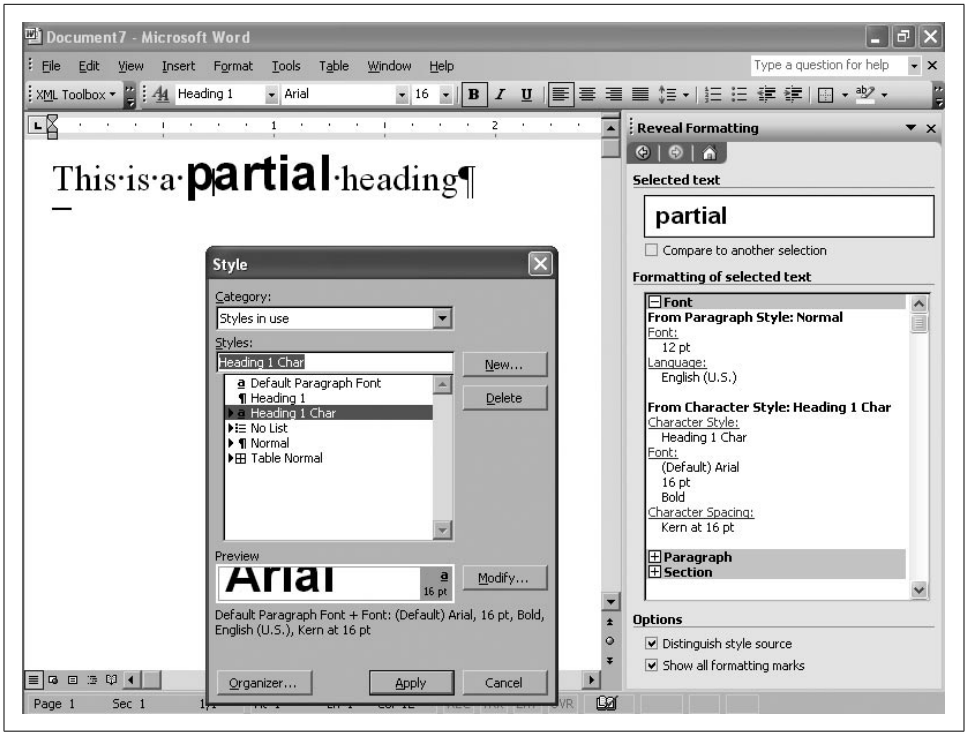


Figure 2-28. An automatically created linked character style, "Heading 1 Char"

```
    <w:basedOn w:val="DefaultParagraphFont"/>
    <w:link w:val="Heading1"/>
    <w:rsid w:val="00B33163"/>
    <w:rPr>
      <w:rFonts w:ascii="Arial" w:h-ansi="Arial" w:cs="Arial"/>
      <w:b/>
      <w:b-cs/>
      <w:kern w:val="32"/>
      <w:sz w:val="32"/>
      <w:sz-cs w:val="32"/>
      <w:lang w:val="EN-US" w:fareast="EN-US" w:bidi="AR-SA"/>
    </w:rPr>
  </w:style>
```

As you can see, all of the run properties from the "Heading 1" style are copied into the new "Heading 1 Char" style. The `w:link` elements retain the association between the two styles by reference to the `w:styleId` attribute of the other style. Word maintains the link between the styles and honors it by propagating any character property changes in one style to the other. It's possible to create a "synthetic" WordprocessingML document outside of Word that links two styles that do not share the same character properties. However, as soon as you try to change one of the styles within Word, all of the character properties of each get merged together and are synchronized from that point forward.