# The "What" of XSL-FO

Somewhere out there, I guess, somebody still believes the paperless office is right around the corner. I don't know *where*, exactly, this person is—certainly nowhere within shouting distance of any of the offices I've ever been in.[1]

There's no question that reducing paper is a worthwhile goal. It really is. Unfortunately, the world is still a long way off from *recognizing* that goal. Even in the Internet age, we're not ready to shrug off the shackles of paper. And that's probably the main reason you need to care about the XSL Formatting Objects standard.

You may recall Chapter 2, similarly titled "The 'What' of XSLT." This chapter, like that one, will cover some of the basic concepts of XSL-FO, including some basic (but by and large unexplained) code. Subsequent chapters in *Just XSL* will delve much deeper into the specifics of the XSL-FO language.

1. Actually, I do know where this person is. He's moved in right next door to the woman who has placed the first order for a gleaming, brushed-aluminum sidewalk in the sky. If they got married and had kids, the kids would all use Buck Rogers–style jetpacks instead of skateboards.

# First, The "Why"

I covered some of this earlier. But now might be a good time to revisit the question of why an XML-based formatting standard is even worth, er, the paper it's printed on.

## The alternatives

If you want to format your documents and data for print, and you *don't* want to do so in a platform-dependent way, you've generally got any number of choices. Among the most common are PostScript and CSS. Each has its advantages and drawbacks.

(Just remember that "platform-(in)dependent" actually means much less than a visitor from another planet might think, given the amount of media attention the phrases receive. PostScript and CSS both "run" on a variety of platforms (in the sense of operating systems), so you might consider them platform-independent. On the other hand, the PostScript specification is owned by Adobe, whereas CSS runs in only a single kind of application—a Web browser—on any of its "platforms." Whether this makes these two formatting languages platform-independent in any real sense is debatable!)

**The operative word is "common"**

Numerous other (some exotic, some quite powerful) options are available. For instance, the LaTex document-preparation language is widely used among the academic community. Old-line Unix techies still favor the troff command-line facility for formatting print documents.

Other options aside, I believe PostScript (in one form or another) and CSS are probably the most widely used print-formatting languages.

## PostScript

PostScript is a text-based language developed by Adobe Systems for creating vector-based "images" (including text) on any output device that understands the language. Here's an example of PostScript code:

```
/Courier findfont                    % Get the font
```

```
24 scalefont                    % Scale it to 24-point size
setfont                         % Make it the current font
newpath                         % Start a new "path"
40 65 moveto                    % Set lower left corner (40,65)
(B Alert!) show                 % "Print" string "B Alert!"
```

Without knowing anything at all about the language, it's pretty easy to guess what's going on in this code fragment. For instance, the percent signs evidently mark the start of inline comments. (And, duh, the comments explain the rest!)

There's also some stuff that *isn't* quite so obvious. One example is the slash at the beginning of the first line, the function of which is to put something (a font, in this case) on the so-called "stack" for later use. (The font on the stack here is used by both the `scalefont` and `setfont` operators.)

PostScript is in wide use, across all kinds of platforms. *Lots* of devices support PostScript—printers, plotters, monitors, and so on. It has two main drawbacks for our purposes here: It's property of Adobe Systems (although Adobe is generally amenable to new implementations), and it's not at all markup-aware.

That second point may be fatal for many applications. Not only does PostScript not process marked-up documents, it's not even in markup form itself. Thus, you couldn't for example develop an XSLT stylesheet to manipulate a PostScript document (although, with a great deal of work, you could transform XML *to* PostScript).

**PostScript and PDF**

In the mid-1990s, Adobe announced its Portable Document Format (PDF) standard. This was intended to make PostScript "Web-friendly" by eliminating one other significant problem of PostScript files: They can be *enormous*.

PDFs are created from PostScript using a "distiller" process that boils down the text-only raw PostScript into a combination of text and binary data. You can embed PostScript fonts in a PDF, too, which greatly increases the chances that a document will print exactly as it was formatted.

More importantly for our purposes, you can use various software packages to generate PDF directly from XSL formatting objects. This eliminates various difficulties involved in handling raw PostScript code, and at the same time allows you to take advantage of features unique to

XSL-FO (primarily, the ability to generate the latter from raw XML using XSLT stylesheets).[2]

## CSS

I talked about CSS versus XSL-FO in Chapter 1. There, I said that CSS's chief drawback as a formatting language is that it's Web-bound. Even with cursory, latter-day nods in the direction of print publication needs, CSS is (and will probably remain) almost exclusively a Web technology.

On the other hand, of course, there's one of XSL-FO's chief drawbacks: It's oriented almost exclusively toward support of *print* applications. This doesn't make XSL-FO any better or worse, *per se*, than CSS. As always, you pick the right tool for the job (i.e., avoid the "when all you have is a hammer, everything looks like a nail" trap).

### Political unrest

A sometimes heated argument has been swirling in the air for a couple of years now, and this is a good time to bring it up. The argument goes like this: "Why do we need another formatting language blessed by the W3C? We've already got CSS; shouldn't we just focus on improving *that*, rather than developing a competing standard?"

It's really not an unreasonable question. Unfortunately, some of those asking the question are asking it rhetorically. They already know the right answer. Which is, in short, that there's nothing wrong with CSS that can't be fixed.

This argument has gone through several stages. At first, it was "All XSL, including both XSLT and XSL-FO, is redundant. You can use scripting languages for the former and CSS for the latter." In fairness, that form of the argument predominated before the one XSL spec

2. While on the subject of PDF, here's one important note: These mixed binary-and-text documents have a certain *legal* standing which is unlikely ever to be attained by text-only documents (including those coded in XSL-FO or other markup vocabularies). The reason is that a document's physical form—including such seemingly non-substantive characteristics as layout, headings, and so on—is considered "information" in a legal sense. A PDF document captures all that information and locks it down in electronic form exactly as it appears when printed.

split into the two separate ones. Since the split, and especially since XSLT has become such a successful technology on its own, you hear much less grumbling about that half of "XSL."

But XSL-FO still raises hackles. A few arguments are put forth most often for its undesirability.

The first is philosophic, and has to do with the notion that XML is all about semantics—about *meaning*—and not about presentation at all. In these terms, an XML vocabulary whose only "meaning" is presentation characteristics may seem a perverse, heretical abomination. I take the point here, in a way: Start with a document in a nice, richly meaningful source vocabulary such as MathML or DocBook or (egad) FlixML; to it, you apply an XSLT transformation whose function is to strip its contents of all meaning *except* presentation. Yeah, in a way that does seem bizarre. On the other hand, it seems to me that "how to present something" is a problem domain profoundly in need of its own structured vocabulary. When you want to send XML to a printer, what language *should* you speak? What's a printer supposed to do with an element named `invoice` or one named `EmpNum`? Saying, "No! No! XML may be used to construct vocabularies for every problem domain *but* presentation!" seems a tad arbitrary and selective (to say nothing of silly).

(This argument against XSL-FO really ignores one central point: that the "meaning" inherent in any XML document is meaning *ascribed to it by humans*. That is, the XSL-FO spec, or any other, chooses element and attribute names on the basis of their usefulness to human readers of documents marked up in that vocabulary. If the human readers see meaning in the markup, then it's meaningful. *There is no inherent meaning in an XML document.*)

Next is the argument that XSL-FO is too explicitly print-bound—in particular, that it's a giant step backward in terms of accessibility of content to vision-impaired users. This may well be true, at least in the sense that XSL-FO continues to place the same emphasis on "reading" that CSS does. XSL-FO, like CSS, specifies quite a few aural properties for spoken (text-to-speech) content, but the lion's share of attention is inarguably on text.

The third argument is the one I've seen presented most recently, and it goes something like this: "Oh no no no! We weren't saying that XSL-FO *as such* is a problem. We were saying that XSL-FO as a formatting vocabulary for content presentation *on the Web* is a problem." I can only scratch my head at this argument. Maybe I'm hanging out with the wrong people, but no one I know *wants* to use XSL-FO for presentation on the Web. We have CSS for that.

Harder to shoot holes in is the argument that, at one level, XSL-FO is no more than CSS properties recast in XML-based form. (If that were *all* it were, I'd have more sympathy for the "Who needs XSL-FO?" point of view.) Also, as you'll see soon enough, XSL-FO is unbelievably verbose. It's not too great a stretch of the English language to say that XSL-FO can be downright ugly. I've got some feelings about this last point, but I'll hold them for now. (Just for now, though!)

(Note that although XSL-FO can be considered ugly, the spec has been developed—as such things go—fairly quickly. As a general rule, the longer a spec incubates before its completion, the uglier its tenets become.)

## Key XSL-FO Concepts

Before getting into all the specific bits of the XSL formatting model, let alone the vocabulary itself, let's clear the air of some terms and concepts you'll need to have a grip on.

### What XSL-FO *is*

XSL-FO is an XML vocabulary that serves one purpose: unambiguously describing block layout, inline formatting, and other presentational characteristics that one bit of content has versus another.

**Where's the spec?**

The XSL-FO standard is available on the W3C Web site, at

`www.w3.org/TR/xsl/`

This is a W3C *Candidate Recommendation* (CR), which means it's not yet final but expected to become so. (The next stage in its "acceptance," if it follows the normal path for such documents, would be Proposed Recommendation, followed by plain-old Recommendation.) The CR "review period" supposedly ended a few weeks ago (as of this writing), which in theory means that the W3C Working Group responsible for the standard is mulling over any comments received before moving it along to the next stage. Since W3C Working Groups deliberate in private, there's no way of knowing for sure which way the wind will blow… just that it's *likely* to be in the direction of the CR.

As you know, the content of an XML document consists of text—letters, digits, punctuation, and whitespace—optionally including pointers to non-XML content (images and what-not) that lies outside the document itself.

So think about it, then: If you were designing a formatting vocabulary for presenting content such as this, what would that vocabulary have to know about and deal with?

- Any vocabulary (or other means) for presenting such content must account for common characteristics of the individual letter-forms themselves (font family, weight, "posture," and so on). If it's intended as a *non-trivial* formatting mechanism, it needs to address the less common characteristics as well, such as spacing between letters, sub- and super-scripting, and the like.
- It must also handle *groups* of letters—spans and paragraphs—in ways appropriate to the language and writing system in which the text will appear. (For example, how much space appears between lines of text? Does the text flow from left to right or right to left, top-to-bottom or vice-versa? What special classifications of text need to be addressed—headings, footnotes and endnotes, callouts, tables?)
- And it must provide facilities for positioning and sizing blocks of content (whether the content in question is text or something else, such as images, which an output device can handle). You'll want to be able to draw a box around such content if desired, for example, or position it to the left, right, top, or bottom of the output medium (irrespective of the alignment of text *inside* the block). You'll probably want the ability to "anchor" such a block of content to a particular piece of text—or, alternatively, to force the block to appear at a certain position regardless of the text around it. And you'll want to be able to use content (again, text and/or images) as a watermark—that is, repeated content floating in the background, behind any "real" content.

Beyond these requirements to style and present the actual content, though, a fully functional formatting language must be able to "understand," even *manipulate*, the format of the output medium in which the content will be presented.

In terms of print, this requirement includes obvious considerations such as paper size and margins. It also covers how to break up the physical page into subpages—regions or areas on the page—reserved for particular purposes, such as running page headers and footers. It addresses matters of text flow from one

page to the next, including widow and orphan[3] control, and whether the "content" area of each page text will appear in single- or multi-column format (and, if the latter, whether the columns need to appear "balanced").

Finally, a print-formatting vocabulary needs some way of expressing the structure of an entire publication—as a collection not just of pages but of *types* of pages. There's one page type for the title page, a different one for the table of contents, one for regular text, one for the index, and so on. (Within each of these *types* there might be, yes, one or more pages.) And note as well that the language in question must be able to spell out what *sequence of page types is acceptable*. Do the glossary pages precede or follow the appendix? Does the title precede or follow the dedication? And so on.

(Of course, if the formatting vocabulary is, like XSL-FO, meant to cover *aural* presentation of the content as well, there's a whole raft of additional issues: pacing, volume, inflection and intonation, pauses, orientation—sounds coming from the left or right, up or down, front or back [the so-called "Z-axis" or "Z-coordinate"]—and so on. I wish I could demonstrate some of these capabilities of XSL-FO; I guess you'll just have to wait for the audio-book edition of *Just XSL*. Or, for that matter, the VRML edition—VRML also provides fully spatialized sound—even if the likelihood of a VRML edition is even more remote than an audio-book!)

**XSL-FO's "verbosity"**

Given the number and variety of types of presentation XSL-FO is meant to support, it's no wonder that the spec's table of contents alone is 8 or 9 pages long, that the spec itself (depending on how it's formatted) is nearly 400 pages long, and that even a "simple" fragment of XSL-FO code can run to a couple of pages.

3. Just in case you haven't seen these terms before, a *widow* is the last line of a paragraph which, without widow control, appears by itself at the top of a page. Depending on a number of factors, this may be merely unattractive or actually problematic (as when the widow consists of an entire sentence, which may cause the reader to miss it altogether). With widow control, a page break is inserted by the software so that more than one line of the paragraph will be bumped to the new page. An orphan presents the opposite problem: The first line of a paragraph is stranded by itself at the bottom of a page. The software-controlled page break will be placed before the orphan, if orphan control is on.

You'll soon see that not only are XSL-FO's "sentences" quite long but so are many of its individual "words." Again, this makes a kind of sense; any full, formal, and unambiguous statement of something—from space-shuttle documentation to human-resource policies to legislative acts to document structure and format—always seems to take lots of big, complicated words to state it. Short words and sentences work best for generalizations.

Given all this, I'm not terribly bothered by XSL-FO's "verbosity"… aside from the fact that I'm in a position to profit from explaining it (cough).

## Namespaces and XSL-FO

If you understand the general concept of namespaces, namespace prefixes, and so on, all you need to know about them in an XSL-FO context is the namespace associated with elements and attributes in the XSL-FO vocabulary. The URI is http://www.w3.org/1999/XSL/Format, and (by convention) the prefix is `fo:`. Therefore, in a document containing these elements and attributes, you'd need the following namespace declaration:

```
xmlns:fo="http://www.w3.org/1999/XSL/Format"
```

Of course, this declaration needs to be in scope at the time an element or attribute with that prefix occurs. Typically, this means the namespace declaration will appear in an XSLT stylesheet's `xsl:stylesheet` element.

## No "hand-editing" of XSL-FO documents

Maybe that's a little extreme. There's nothing at all to prevent your building from scratch an entire XSL-FO document, if you're so (masochistically) inclined. After all, XSL-FO is just another XML vocabulary. As long as you've got the right attributes on the right elements, and all the elements and text conforming to the right context models, who cares whether the document was authored and edited by a human or generated by software?

The answer is, *you'll* care—once you've tried to create even a simple XSL-FO document yourself. The reasons are the language's verbosity *and* its problem domain, both of which I addressed above. Even if your XML authoring/editing software is quite happy auto-generating the lengthy element and attribute names for you, freeing you just to enter the content, the content is really devoid of "meaning." This is a tedious and confusing way to work.

Much simpler is to author your documents in a real XML vocabulary, one which truly describes and structures the documents' contents in a meaningful way, and then transform the documents to XSL-FO with XSLT stylesheets. The XSLT processor doesn't grow bored typing and re-typing lengthy element names, and it does so consistently, one transformation after another. Any sensible use of XSL-FO simply will not require hand-coding of XSL-FO "stylesheets." In fact, the only way you may ever need to see XSL-FO code is in the context of templates in an XSLT stylesheet.

**Practicing what I preach**

In this part of *Just XSL*, I'll be showing you many code fragments as if I'd hand-coded them myself. This is an illusion. Although I do need to show you, for instance, how one element fits inside another, the XSL-FO code fragments I'll demonstrate here are always the result trees from XSLT transformations. I may hand-tweak them for presentation purposes—making sure their indentation is reasonable, for example, and replacing lengthy sections of repeated or irrelevant code with ellipses. But creating XSL-FO code by hand is just crazy.

Oh, I know—there are always obsessive-compulsive types who will insist on building things by hand rather than using the equivalent of power tools. When this book was in production, I heard of someone who'd coded by hand, in VRML, an incredibly complex world called the Tralee Mars Colony. VRML, like XSL-FO, is not a language for the faint of heart even when power-assisted. When one hears of such people, one just has to shake his head in amazement.

## The `fo:root` element

The root element of an XSL-FO result tree or document must be `fo:root`. This element's syntax is very basic; it's used, minimally, as the container for the rest of the document and as a repository for namespace declarations. (There are quite a number of "inheritable" properties—inheritable in the same sense that a namespace declaration is—that can be assigned in the `fo:root`, as well; I'll cover them in Chapter 10.)

Within the `fo:root` element are any number of other elements. Roughly, these fall into two categories: elements that describe the types of layouts in a publication and elements that make up the actual content. These two categories of the `fo:root` element's descendants together reflect the XSL-FO *formatting model*.

# The XSL-FO Formatting Model

A content model in XML is an expression of how the various elements and text contents fit together when represented in a given XML vocabulary.

Similarly, XSL-FO can be said to have a *formatting model* describing the interrelationships of different kinds of formatting. The spec itself is replete with a dizzying constellation of graphical illustrations of this formatting model and the steps a formatting processor takes to implement it. I'm not going to try reproducing most of these illustrations here; after you've stared at each one for a few minutes, you'll probably come to suddenly think (as I did), "You mean *that's* all this thing illustrates?"

A few of the illustrations are essential, though, and I'll provide my own versions of them when the time comes.

## XSL-FO's view of a publication

In the XSL-FO world, a document—in its as-output form—somewhat resembles an Open eBook document, as I described at the end of Chapter 6: You spell out not only the individual chunks of content but also the sequence in which they appear. The overall structure of an XSL-FO publication is defined in the portion of the XSL-FO document called a *layout master set*; the content goes into a series of *page sequences*. Thus, the general structure of an XSL-FO document (result tree) is something like this:

```
<fo:root>
   [...layout master set...]
   [...page sequence(s)...]
</fo:root>
```

Within the layout master set are specifications for one or more *types of pages*, and there's a *sequence* in which those types of pages may appear. Each type of page is referred to as a *simple page master*; the order in which each may appear, as *page sequence masters*. So we can expand the general structure of an XSL-FO document to something like this:

```
<fo:root>
   [...layout master set:
     simple page master(s)...
     page sequence master(s)...]
   [...page sequence(s)...]
</fo:root>
```

Now, before going any further, note the use of the word "master" in all these terms. The XSL-FO spec uses the "master" designation to denote a class of objects. Thus, for instance, a simple page master does not describe a specific page; it describes a *class* or type of page. One or more specific pages will have the characteristics of that master, but their contents will differ.

The second point I wanted to raise here is that the XSL-FO vocabulary just *loves* hyphens. Most of these terms actually appear in the spec as hyphenated phrases: layout-master-set, simple-page-master, page-sequence-master. The significance of this is that these are generally the *names* of element types in the XSL-FO language; if you understand the nesting of the concepts, you also understand at least the rudiments of the language's content model. Thus, an XSL-FO result tree will include a structure something like this:

```
<fo:layout-master-set...>
   <fo:simple-page-master...>
   <fo:simple-page-master...>
   <fo:simple-page-master...>
   <fo:page-sequence-master...>
   <fo:page-sequence-master...>
   <fo:page-sequence-master...>
</fo:layout-master-set>
```

## Simple page masters

Each simple page master considers a page to consist—at least potentially—of five *regions*. Each region (as the term implies) is a rectangular portion of the page within which content of one kind or another can appear. The five regions are listed below:

- `region-body`: This is the main area of a page—the part where the bulk of the content will be placed. The other four regions surround it.
- `region-before`: In Western writing systems, this corresponds to a "page header" area (that is, a rectangular area that occurs *before* the body itself).
- `region-after`: In Western writing systems, this corresponds to a "page footer" area (that is, a rectangular area that occurs *after* the body itself).
- `region-start`: In Western writing systems, this corresponds to a "left margin" area (that is, a rectangular area that occurs before the *start* of individual lines).
- `region-end`: In Western writing systems, this corresponds to a "page header" area (that is, a rectangular area that occurs after the *end* of individual lines).

Notice that the terms for the "surrounding" regions don't explicitly refer to directions such as top, bottom, left, and right. They're named in ways whose actual meanings differ from one language or writing system to another. Throughout the rest of *Just XSL*, I'll fall back on the more common Western terms—"before" means "top," "after" means "bottom," "start" means "left," and "end" means "right"— but remember that the results as formatted in one part of the world may be quite different from those as formatted somewhere else.

Figure 9–1 depicts the relationships of these regions within a simple-page-master (again, assuming a Western orientation).

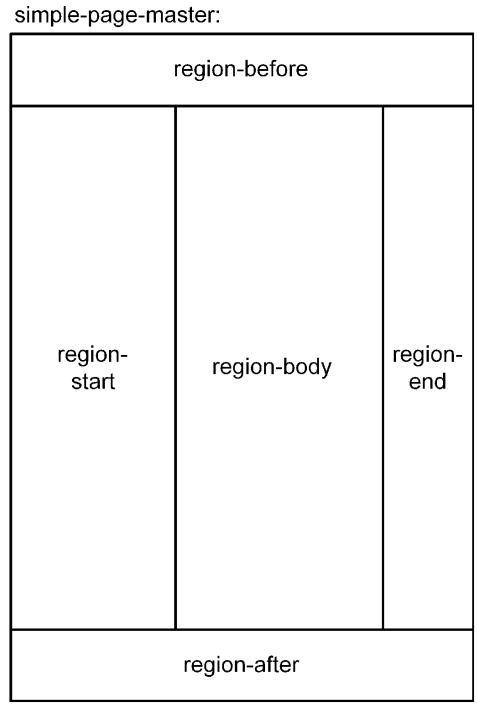


**Figure 9-1** Breakdown of an XSL-FO simple-page-master into its constituent regions. The simple-page-master itself isn't the entire physical page; it's the entire area which may contain content, the area *within the page margins*. In certain non-Western writing systems, the region-start and region-end, and/or the region-before and region-after, may switch positions.

As before, you wouldn't be far off the mark if you guessed from this that an XSL-FO result tree might contain something like the following:

```
<fo:simple-page-master...>
   <fo:region-body...>
   <fo:region-before...>
   <fo:region-after...>
   <fo:region-start...>
   <fo:region-end...>
</fo:simple-page-master>
```

**About that "simple"…**

The simple page master is only one kind of *page master*—the only kind, as it happens, described by the XSL-FO spec. Without offering much discussion of the alternative, the spec says only that future versions of the standard "will support more complex page layouts." This will undoubtedly be accompanied by a fresh crop of graphics showing boxes-within-boxes.

The `fo:simple-page-master` element takes one interesting (and optional) attribute: `media-usage`. This attribute controls the overall way in which the output medium or device (referred to as the "User Agent") presents the XSL-FO document's content. It can take one of four possible values:

- `auto` (the default): How the output is presented will behave as if one of the other three values were assigned, depending on circumstances the User Agent can assess.
- `paginate`: Both the height and width of the output medium are fixed. (This is the most common value for print publications, of course.) Note that the effect of fixing the height and width of the medium is that, every time that two-dimensional area gets filled up, another instance of it is presented for filling. It's paginated, in other words, and hence this value for `media-usage`.
- `bounded-in-one-dimension`: The most obvious example here is a browser window or other "scrollable" area. One dimension is nailed down, and the other simply grows or shrinks to accommodate it.
- `unbounded`: As the value implies, there are no limits to either height or width. This presumes that text will not wrap, and that there are no line-feed characters in the content to *force* the wrapping. Each chunk of text simply takes up as much "width" as it needs, and the number of chunks determines the "length."

- `repeatable-page-master-alternatives`: This kind of page sequence is most commonly exemplified by the left- and right-hand pages of a book or magazine. Page headers and footers commonly have different content depending on whether a given page is odd- or even-numbered, and even the page margins may be different to allow for the binding (the edge of the page closest to the staples, stitching, and/or glue holding the publication together). Like the `repeatable-page-master-reference`, this kind of page sequence just goes on and on (in this case, cycling through the alternative layouts) until all its content is exhausted.

Given these three kinds of page sequences, then, a typical page sequence master might be coded in XSL-FO like the following:

```
<fo:page-sequence-master...>
   <fo:single-page-master-reference...>
   <fo:repeatable-page-master-alternatives...>
   <fo:repeatable-page-master-reference...>
</fo:page-sequence-master>
```

This might indicate that this page sequence master describes (respectively) a book's title page, any number of pages laid out differently for left- and right-hand pages, and an index whose pages are all laid out identically, regardless of which side of a two-page spread they fall on.

## Page sequences

A *page sequence* is an instance of actual pages with content, not simply a *type* of page. Thus, this is where most of the content from your source tree will wind up, formatted in a way that is controlled by a page master and page sequence master working together.

Each `fo:page-sequence` element contains at least one and up to three (or more) children:

- `fo:title`: An optional element used to assign a title to a document, this serves the same purpose as an XHTML document's `title` element. Depending on the User Agent, this might end up being reproduced in a window's title bar, as a watermark or running footing, and so on.
- `fo:static-content`: Also optional, this element can occur as many times as you need it. Its purpose is to declare (as the name implies) content that does not change—or, rather, that does not change according to some

condition in the source tree. For instance, you might put the literal text "First Edition" in the region-after area of every page. Or you might put "Page *n*" on every page (where *n* is replaced by a special `fo:page-number` element); even though the page number varies from one page to the next, this kind of content always appears at the same place in a page sequence defined by this `fo:page-sequence` element.

- `fo:flow`: Here's where your actual content appears, and this is the only required child of each `fo:page-sequence` element. The content is not contained directly by the `fo:flow` element; this serves just as a "wrapper" for the contents of the page sequence as a whole.

## Bottom line

In Chapter 10, I'll cover the details of the XSL-FO elements I've mentioned so far. For now, though, you already know the rough framework of describing a publication's overall layout and structure using XSL-FO:

```
<fo:root media-usage="type">

  <!-- The one layout-master-set in each XSL-FO
    document describes the overall structure which the
    publication will have. -->
  <fo:layout-master-set>

    <!-- Every simple page master describes a different
      type of page in the publication. -->
    <fo:simple-page-master>
      <fo:region-body/>
      <fo:region-before/>
      <fo:region-after/>
      <fo:region-start/>
      <fo:region-end/>
    </fo:simple-page-master>

    <!-- Every page sequence master describes a
      different *sequence* of pages within the
      publication. -->
    <fo:page-sequence-master>
      <fo:single-page-master-reference...>
      <fo:repeatable-page-master-alternatives...>
      <fo:repeatable-page-master-reference...>
    </fo:page-sequence-master>
```

```
    </fo:layout-master-set>

    <!-- Publication's content is all poured into
      a series of fo:page-sequence elements. -->
    <fo:page-sequence>...</fo:page-sequence>
    <fo:page-sequence>...</fo:page-sequence>
    <fo:page-sequence>...</fo:page-sequence>
      [etc.]
</fo:root>
```

## Formatting objects and properties

The XSL-FO spec seems to waver on exactly what the term "formatting object" refers to. (This is surprising in that the term is built into the vocabulary's very name.) At one point, it says that all the elements in the result tree of an XML-to-XSL-FO transformation are formatting objects. At other points, it indicates that only *some* of these result-tree elements are formatting objects.

As a practical matter, I think it's safe to consider a formatting object to be any element which either is itself subject to formatting *or* provides that formatting. For example, the various regions described in the previous section are all formatting objects (FOs); they may assume various presentational characters and hence fall into the first category—objects to which formatting may be applied. There's also a color-profile FO, which is used (as you might guess) to declare the coloring scheme to be used in a document. This isn't itself—unlike one of the regions—something that is actually rendered in a document or something that "has content." It simply assigns formatting characteristics to other FOs, and hence is an example of the second FO category.

### Categories of FOs

Formatting objects can be placed in one of five general categories, as summarized in Table 9–1. The categories are assigned based on where in a page sequence (or, more precisely, where within an `fo:flow` element) you may use the FOs within that category.

A block FO is a contiguous area of content derived directly from the source tree. An out-of-line FO is, in some ways, similar to a block FO—like the latter, it's a single "thing"—but it operates independently of the flow of content. For instance, a callout (an excerpt from the content, placed above or to the side of a

**Table 9-1** Formatting-Object Categories

| Category | Description |
|---|---|
| block | Block FOs work similarly to XHTML's `div` element. They're single *things* (or clusters of things) which can be manipulated (formatted) as a unit. Among the block FOs are an element called `fo:block`, one called `fo:table`, and so on. |
| inline | Inline FOs are analogous to XHTML's `span` element, in that they assume some characteristic(s) apart from those of the containing block or other FO. There's an inline element named `fo:inline`, one named `fo:character`, one named `fo:page-number`, and so on. |
| neutral | These FOs may be used anywhere that text (#PCDATA), inline FOs, or block FOs may be used. Among the "neutral" FOs are one called `fo:wrapper` and one called `fo:retrieve-marker`. |
| out-of-line (1) | There's only one FO in this category, `fo:float`. It may be used anywhere that text, inline FOs, or block FOs may be used, *except* as a descendant of an out-of-line FO. |
| out-of-line (2) | Likewise, there's only one FO in this category, `fo:footnote`. It may be used anywhere that text or inline FOs may be used, *except* as a descendant of an out-of-line FO. |

paragraph in which the excerpt appears) might be considered an out-of-line FO. Footnotes are out-of-line FOs because they, too, appear outside the normal flow of content.

## Properties

*Properties* are characteristics (generally presentation-related) which formatting objects may have. There's a huge overlap between CSS's properties and XSL-FO's, and where the overlap exists the properties even share the same name. For instance, the XSL-FO equivalent of the CSS `font-size` property is also called `font-size`.

Of eight main numbered sections in the XSL-FO spec, the last, titled "Conformance," is a page or less long when printed out. Section 7 describes the XSL-FO properties, and begins before the half-way point—easily the biggest single portion of the standard. (If you threw out Section 7, in other words, the spec would be half its present length. Of course, in that case XSL-FO would be much more lightweight in more ways than one!)

There's no reasonable way to cover *all* FO properties in *Just XSL*. You can probably safely assume, though, that if there's some formatting characteristic imaginable, it's covered somewhere in the spec.

Properties are used in an XSL-FO document as *attributes* placed on the FO element(s) desired. For example, the `text-align` property can be assigned to the `fo:block` element, as in this example:

```
<fo:block text-align="right">Text</fo:block>
```

(You can probably guess that this element formats the word "Text" as right-aligned text within the block FO.)

**Properties, traits, attributes…**

One incredibly annoying feature of the XSL-FO spec—at least in its Candidate Recommendation form—is that it seems determined to invent new words where perfectly good ones already exist, in the general XML universe of discourse.

This business about properties being represented as what are more commonly called attributes is just one example. Another is the term "trait," which seems to appear in the text almost as often as the phrase "formatting object" but is defined nowhere. (Not that "formatting object" is, either.) If you read closely enough, though, it appears that the terms "property" and "trait" are interchangeable. For instance, `line-height` is sometimes referred to as "the `line-height` property" and sometimes as "the `line-height` trait."

If I were you, I wouldn't lose too much sleep over any of this. I'll try to stick to the word "property" to refer to some characteristic of an FO, and "attribute" to refer to what you put on an element representing the FO.

## Transforming to an XSL-FO Document

I've pretty much given you the basics of this, especially in the earlier section titled "Bottom line."

What I'm going to give you now is a bare bones overview of an XSLT stylesheet for transforming a FlixML document to an XSL-FO document. All this XSL-FO document will contain, at first, is the title of a reviewed movie. Then I'll show you how to view the results (given that we won't be able to look at it in a Web browser any longer!).

But first, here's a brief digression to the movie itself, and its FlixML review.

### B Alert!

**Johnny Guitar (1954, Republic Films)**

Flamboyantly mannish saloon owner Vienna (Joan Crawford) has never gotten along with dully mannish Emma. And when Vienna has a fling with The Dancin' Kid, whom Emma covets, Emma will stop at nothing to get revenge—including driving the Kid out of town.

Into this mix rides gunslinger Johnny "Johnny Guitar" Logan (Sterling Hayden). He and Vienna had a relationship years ago, and now she's brought him into her present to help defend her interests as she prepares to make a killing from the coming railroad boom. Unfortunately, she's at the bank, withdrawing money, when the Dancin' Kid's gang pulls one last heist—and Emma, convinced that Vienna was in on the robbery, wants her hanged….

*Johnny Guitar* has been tremendously popular in Europe. (Do a Web search on the film title and you'll find as many sites in French, Italian, Spanish, and German as in English.) Truffaut was especially crazy about it. In one scene from his *Mississippi Mermaid* (1969), Catherine Deneuve and Jean-Paul Belmondo emerge from a theater where they've just seen *Johnny Guitar*. Deneuve says, "You were right. It's not just a film about horses." Reportedly, director Ray was so taken with the Joan Crawford wardrobe in this film that he cribbed from it when costuming James Dean in his later *Rebel Without a Cause*.

## Johnny Guitar's FlixML review

```
<flixinfo author="John E. Simpson" copyright="2001"
   xml:lang="EN"
   xmlns:xlink="http://www.w3.org/1999/xlink/namespace/">
   <title role="main">Johnny Guitar</title>
   <genre>
     <primarygenre>Western</primarygenre>
   </genre>
   <releaseyear role="initial">1954</releaseyear>
   <language>English</language>
```

```
<studio>Republic Pictures Corporation</studio>
<cast id="castID">
  <leadcast>
    <female id="JCrawford">
      <castmember>Joan Crawford</castmember>
      <role>Vienna</role>
    </female>
    <male id="SHayden">
      <castmember>Sterling Hayden</castmember>
      <role>Johnny Guitar</role>
    </male>
    <female id="MMcCambridge">
      <castmember>Mercedes McCambridge</castmember>
      <role>Emma Small</role>
    </female>
    <male id="SBrady">
      <castmember>Scott Brady</castmember>
      <role>The Dancin' Kid</role>
    </male>
    <male id="WBond">
      <castmember>Ward Bond</castmember>
      <role>John McIvers</role>
    </male>
  </leadcast>
  <othercast>
    <male id="EBorgnine">
      <castmember>Ernest Borgnine</castmember>
      <role>Bart Lonergan</role>
    </male>
    <male id="JCarradine">
      <castmember>John Carradine</castmember>
      <role>Old Tom</role>
    </male>
  </othercast>
</cast>
<crew id="crewID">
  <director>Nicholas Ray</director>
  <screenwriter>Philip Yordan</screenwriter>
  <cinematog>Harry Stradling, Sr.</cinematog>
  <sound></sound>
  <editor>Richard L. Van Enger</editor>
  <score>Victor Young</score>
  <speceffects></speceffects>
  <proddesigner>Edward G. Boyle</proddesigner>
  <makeup></makeup>
  <costumer>Sheila O'Brien</costumer>
```

```
   </crew>
   <plotsummary id="plotID">Flamboyantly mannish saloon owner
Vienna (Joan Crawford) has never gotten along with dully mannish
Emma. And when Vienna has a fling with The Dancin' Kid, whom Emma
covets, Emma will stop at nothing to get revenge -- including
driving the Kid out of town.<parabreak/>Into this mix rides
gunslinger Johnny "Johnny Guitar" Logan (Sterling Hayden). He and
Vienna had a relationship years ago, and now she's brought him into
her present to help defend her interests as she prepares to make a
killing from the coming railroad boom.<parabreak/>Unfortunately,
she's at the bank, withdrawing money, when the Dancin' Kid's gang
pulls one last heist -- and Emma, convinced that Vienna was in on
the robbery, wants her hanged.</plotsummary>
   <reviews id="revwID">
     <flixmlreview>
       <goodreview>
         <reviewtext></reviewtext>
       </goodreview>
     </flixmlreview>
     <otherreview>
       <goodreview>
         <reviewlink
xlink:href="http://www.entertainmentnutz.com/movienutz/movie
reviews/johnny_guitar.htm">MovieNutz (Brian W.
Fairbanks)</reviewlink>
       </goodreview>
       <goodreview>
         <reviewlink
xlink:href="http://film.guardian.co.uk/Century_Of_Films/Story/
0,4135,37234,00.html">Guardian Unlimited (Derek
Malcolm)</reviewlink>
       </goodreview>
       <goodreview>
         <reviewlink
xlink:href="http://members.aol.com/michaemann/jgmain.html">
The "Johnny Guitar" Society</reviewlink>
       </goodreview>
       <goodreview>
         <reviewlink
xlink:href="http://www.pifmagazine.com/vol23/v_Johnny_Guitar.shtml"
>Remote Control (Nick Burton)</reviewlink>
       </goodreview>
     </otherreview>
   </reviews>
   <distributors id="distribID">
     <distributor>
```

```
      <distribname>Reel.com</distribname>
      <distribextlink>
        <distriblink
xlink:href="http://www.reel.com/movie.asp?MID=5001"/>
      </distribextlink>
    </distributor>
    <distributor>
      <distribname>Amazon.com</distribname>
      <distribextlink>
        <distriblink
xlink:href="http://www.amazon.com/exec/obidos/ASIN/6303391931"/>
      </distribextlink>
    </distributor>
    <distributor>
      <distribname>Yahoo! Video Shopping</distribname>
      <distribextlink>
        <distriblink
xlink:href="http://shopping.yahoo.com
shop?d=v&amp;id=1800083797&amp;clink=dmvi-ks/johnny_guitar"/>
      </distribextlink>
    </distributor>
    <distributor>
      <distribname>BlockBuster</distribname>
      <distribextlink>
        <distriblink
xlink:href="http://www.blockbuster.com/mv/
detail.jhtml?PRODID=117675&amp;CATID=1100"/>
      </distribextlink>
    </distributor>
    <distributor>
      <distribname>MovieGallery</distribname>
      <distribextlink>
        <distriblink
xlink:href="http://www.moviegallery.com/vhs_info.cgi?product
_id=REP2127.3"/>
      </distribextlink>
    </distributor>
  </distributors>
  <dialog>Johnny: How many men have you forgotten? Vienna: As many
women as you've remembered.</dialog>
  <remarks>Tremendously popular in Europe. In one scene from
Truffaut's "Mississippi Mermaid" (1969), Catherine Deneuve and
Jean-Paul Belmondo emerge from a theater where they've just seen
"Johnny Guitar." "You were right," says Deneuve, "It's not just a
film about horses."</remarks>
  <mpaarating id="rateID">NR</mpaarating>
```

```
   <bees b-ness="&BEE5URL;"/>
</flixinfo>
```

## Creating the basic result tree

There's not too much to add to what you already know. The idea—just as when transforming to XHTML or Open eBook, for example—is to begin with the root of the source tree and create a corresponding "root structure" in the result.

In this case, the result tree will be a print publication that at first simply re-creates a portion of the contents of a FlixML document. Eventually, this print publication will be one of those little "playbill"-type handouts you get when attending a play, and it might conceivably include reviews of more than one film (for a hypothetical "FlixML B-Film Festival"). But we'll work up to that.

### Laying out the page

For now, let's just start with the information shown in Table 9–2, defining the look of an extremely simple page. Dimensions are given in points, a unit of measurement often used by print designers; there are approximately 72 points in an inch.[4]

**Table 9-2** General Layout/Structure of a Simple Document Page

| Item | Dimension (in points) |
|---|---|
| Page size (total) | 200 W × 200 H |
| Page size (printable) | 100 W × 100 H |

Note that, for this simple example, we won't start out specifying any of the regions *around* the body, just the body region itself.

### From layout to result

With the dimensions provided in Table 9–2 we have the rudiments of a simple page master. So we can start to build the XSLT stylesheet like this:

4. Although you *can* use points to express both vertical and horizontal dimensions, technically they're for use only in specifying an object's height. As a unit of measurement for demonstrations, though, they're an ideal size, especially when dealing (as we will be) with Adobe's Acrobat Reader software.

```
<xsl:template match="flixinfo">
   <fo:root
     xmlns:fo="http://www.w3.org/1999/XSL/Format">
     <fo:layout-master-set>
       <fo:simple-page-master master-name="page-review"
         page-width="200pt" page-height="200pt"
         margin="50pt">
         <fo:region-body region-name="reg-body"/>
       </fo:simple-page-master>
       <fo:page-sequence-master
         master-name="seq-mast-cover">
         <fo:single-page-master-reference
           master-name="page-review"/>
       </fo:page-sequence-master>
     </fo:layout-master-set>
     <xsl:apply-templates select="title"/>
   </fo:root>
</xsl:template>
```

The layout master set constructed by this template rule has a single simple page master, which is assigned the name `page-review` (via its `master-name` attribute). The width and height are assigned as in Table 9–2; the margin is set to 50 points all around, which will make the "printable" area 100 points square. The `page-review` simple page master contains a single region—the body, which is assigned a name of `reg-body`.

Having defined the general layout of the page with the simple page master element, we need to indicate how pages which use this layout will be sequenced in the resulting document. This is accomplished with a page sequence master, named `seq-mast-cover`. It contains a single child, a `single-page-master-reference` whose `master-name` attribute does *not* name this reference, but rather points back to a page master which has this name—that is, in this case the `page-review` simple page master previously set up.

Note the way these elements and attributes are named. Yes, the names are confusingly (almost maddeningly) alike. However, their names are built up in a logical manner. For instance, as I mentioned, when an element name ends in "-master," that element will serve not to hold actual content but to describe a

```
<xsl:template match="title">
   <fo:page-sequence master-name="seq-mast-cover">
     <fo:flow flow-name="reg-body">
       <fo:block font-size="20pt"
         border-style="solid" border-width="1pt">
         <xsl:value-of select="."/>
       </fo:block>
     </fo:flow>
   </fo:page-sequence>
</xsl:template>
```

This template rule instantiates in the result tree a page sequence—that is, a flow object which actually contains some content. There's a `master-name` attribute, whose value asserts that the layout of this page sequence's contents is defined by the `page-sequence-master` element in the preceding template rule. (See the way the `master-name` attribute here ties back to the `master-name` attribute there?)

Within the page sequence, we're going to flow content into the body of the page. This portion of the page layout was previously defined by a region named `reg-body`, so that's the value we give to the `flow-name` attribute. Within the `fo:flow` element, we'll set up a block FO using 20-point type and surrounded by a narrow box. The contents of this block FO will be supplied by the value of the `title` element we're currently processing, per the `xsl:value-of` element in the template.

When applied to the *Johnny Guitar* FlixML review, this stylesheet creates the following result tree:

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
   <fo:layout-master-set>
     <fo:simple-page-master
       master-name="page-review"
       page-width="200pt" page-height="200pt"
       margin="50pt">
       <fo:region-body region-name="reg-body"/>
     </fo:simple-page-master>
     <fo:page-sequence-master
       master-name="seq-mast-cover">
       <fo:single-page-master-reference
         master-name="page-review"/>
     </fo:page-sequence-master>
   </fo:layout-master-set>
   <fo:page-sequence master-name="seq-mast-cover">
     <fo:flow flow-name="reg-body">
```

```
      <fo:block font-size="20pt" border-style="solid"
        border-width="1pt">Johnny Guitar</fo:block>
    </fo:flow>
  </fo:page-sequence>
</fo:root>
```

Again, notice—now that the surrounding XSLT stuff has been removed—how the attributes in later elements refer back to those identifying earlier elements.

## Viewing an XSL-FO Document

Now we come to a critical juncture: How do you actually look at one of these XSL-FO documents? (Or, heck, what can you do with one, period—look at it or otherwise?)

I'm not going to get into the details of using XSL-FO software at this point. But I do want to explain how I'm going to be showing you the effects of producing XSL-FO documents.

### Step 1: Generate the XSL-FO document

To "generate" an XSL-FO document you can, of course, create one by hand. As I've mentioned, though, that's not a recommended course of action, and it's certainly not the one I'm using.

I'll use the Saxon XSLT processor to produce my XSL-FO result trees. I won't always show you (as I did above) the XSLT which transforms a given FlixML document in a given way; but understand that when you see blocks of XSL-FO code here, I've almost *never* hand-authored it.

### Step 2: Convert the XSL-FO to PDF

Huh? Why go to all the trouble of generating XSL-FO result trees if you're just going to convert them to another Adobe format?

Granted, this seems at first glance to be kind of silly (if not downright perverse). There are a few reasons for it:

- Almost no native XSL-FO processors are available at the time I'm writing this. (And the one or two that are out there aren't particularly stable.) However, a number of solid XSL-FO-to-PDF converters *are* available.

- …and, of course, there's one extremely good PDF renderer widely available: Adobe's own free Acrobat Reader product. Note that Adobe's product for *creating* PDFs, Acrobat Exchange, is *not* free.
- Finally, I've got to convert the XSL-FO to *something* if I'm going to show you the effects of the vocabulary on output. Just as with the FlixML-as-XHTML MSIE screen shots in the first part of the book, screen shots of an Acrobat Reader window will make immediately obvious how a given result tree "works."

## What I'll use

The specific package I'll use for Step #2 is the enigmatically named Xep, a product of RenderX. I had a couple of choices, but Xep is easily the most advanced product in terms of compliance with the XSL-FO standard.

On the other hand, Xep has a couple of disadvantages. First, its full-featured version is *very* expensive. (Anyone used to spending a couple hundred dollars for software, if that, will have the breath knocked out of her by Xep's price tag.[5]) Second, although Xep *is* available in a time-limited evaluation version, this version is functionally crippled in some ways. For instance, it stamps every page with a watermark-like indication that the page was created using a RenderX product; and after page 11, every page is *blank*.

**Making do**

Neither of these limitations will be crippling for my purposes (I'll just use the evaluation version), but they may be for yours. If you're interested in purchasing the uncrippled version of Xep, contact RenderX's sales unit at sales@renderx.com. The evaluation version can be downloaded from the company's Web site, www.renderx.com.

In terms of sheer numbers of users, probably the most popular XSL-FO-to-PDF package is FOP, originally developed by James Tauber but

5. Not to imply that it's necessarily *over*-priced. The XSL-FO standard is undoubtedly enormous, and implementing a package that complies with it is an amazing achievement for RenderX. But it's not a product that will help drive XSL-FO's acceptance among the masses. (Among the masses of publishers with deep pockets, maybe.)

now supported by the Apache open-source project. Currently at version 0.16, FOP's greatest virtues are that it *is* open source and it's free. Unfortunately, both of those virtues combine to generate its biggest current weakness: It simply doesn't support enough of the current spec to be useful as a tool for *demonstrating* the current spec.

If you've got any application-development skills at all and would like to make a mark on the history of both the XML world and the open-source movement, I strongly urge you to get involved in bringing FOP up to snuff.

The XSL-FO result tree I reproduced earlier resides in a file called playbill.fo. (Other common filename extensions for XSL-FO documents are .fob and .xfo. However, if you use the .fo extension, Xep will automatically name the resulting file—in this case—playbill.pdf. Otherwise it simply tacks the .pdf extension onto the full XSL-FO document's filename.) When I run Xep (which is a command-line utility) against playbill.fo, it displays various status messages as it works its way through the XSL-FO code and then writes out a playbill.pdf file.

Figure 9–2 illustrates how playbill.pdf looks. I've turned Acrobat's "forms grid" on, and set it to display the grid in points (using a 10-point grid size) rather than inches. This makes it easier to understand the relationship between the PDF output and the XSL-FO that went into it. Note that the box surrounding the film's title starts at 50 points down, 50 points in from the left, and extends to the right margin—also 50 points in, per the margin settings for this simple page master. However, the vertical size of the box simply extends only to the bottom of the enclosed content, not to the bottom margin. This would allow further content to be added within the box, if so desired.

## Other Regions

In general, the content to be transferred from your source tree will go into the body region of your XSL-FO result tree. In the other regions, you'd typically insert static text that would repeat from one page to the next. (This is enforced by a requirement that a page sequence may have only one `fo:flow` child element, but several `fo:static-content` children.) Let's add a couple of simple examples to the simple result tree we've got so far.

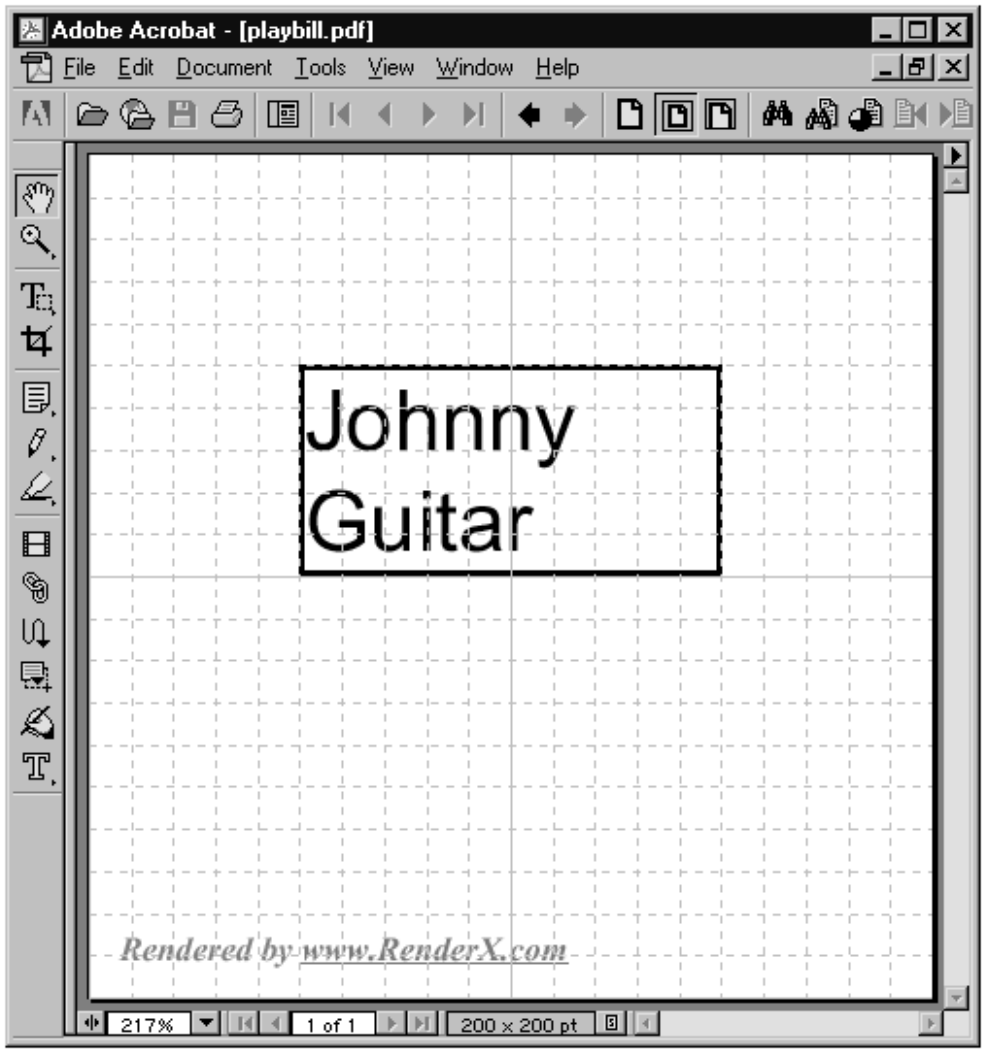


**Figure 9-2** A simple XSL-FO rendering of part of the *Johnny Guitar* FlixML review, converted to PDF by RenderX's Xep processor. (Note the RenderX plug inserted at the bottom left; in the registered commercial version of Xep, this tagline does not appear.) The page size displayed in the status bar (200 × 200 pt.) is consistent with the page-width and page-height settings established for this simple page master in the XSL-FO code.

## Adding a header: `region-before`

The first thing you've got to do before adding any actual text content to another (non-body) region in the result tree is establish the region's existence in the first place. This occurs in the simple page master's definition, immediately following the `fo:region-body` element.

The size of one of these "side regions," as they're called, is specified using an attribute named `extent`. For the before and after regions, this attribute defines their *height*; for start and end, their *width*.

So let's take a stab at declaring a before region. The result tree containing the simple page master will now look like this:

```
<fo:simple-page-master
   master-name="page-review"
   page-width="200pt" page-height="200pt"
   margin="50pt">
   <fo:region-body region-name="reg-body"/>
   <fo:region-before region-name="reg-before"
    extent="20pt"/>
</fo:simple-page-master>
```

The only change so far is the addition of the `fo:region-before` element. It specifies an extent of 20 points in size, which means that the before region will be 20 points high.

Now, to insert some static text in this before region, we also need to make a corresponding addition to the page sequence (which, again, is where the actual content appears):

```
<fo:page-sequence master-name="seq-mast-cover">
   <fo:static-content flow-name="reg-before">
    <fo:block font-size="10pt"
      background-color="silver">
      A FlixML Review
    </fo:block>
   </fo:static-content>
   <fo:flow flow-name="reg-body">
    <fo:block font-size="20pt"
      border-style="solid" border-width="1pt">
      Johnny Guitar
    </fo:block>
   </fo:flow>
</fo:page-sequence>
```

The addition is the `fo:static-content` element. Its contents will appear in every "reg-before" region created by this result tree—that is, in the `region-before` just added to the simple page master. Here, I've set the contents to the phrase "A FlixML Review," which will appear in 10-point type on a silver background.

**A final reminder**

I'm not going to belabor the issue further, but remember that what I'm showing you is the result tree of an XSLT transformation, from FlixML to XSL-FO. If this were a fragment of XSLT code, the words "Johnny Guitar" wouldn't be there—in their place would be the `xsl:value-of` element I showed you earlier.

All excited, we crank this through Xep and see Figure 9–3. Uh-oh. Looks like we've got something scrambled, hmm?

The "problem" isn't really a problem; it just requires a little more thought about what we've declared the document's layout to be at this point.

Remember first that the printable page area (defined by the `fo:simple-page-master` element's `height` and `width` attributes) was set to 200 by 200 points. The `region-before` has an extent of 20 points, and this seems to have rendered correctly. But there's no "automatic shoving-down" of the `region-body`; we've got to arrange that ourselves.

Typically, this is done by adding a `margin` attribute to the `fo:block` element enclosing the `region-body`'s contents. For instance,

```
<fo:flow flow-name="reg-body">
   <fo:block font-size="20pt"
     border-style="solid" border-width="1pt"
     margin-top="20pt">
     Johnny Guitar
   </fo:block>
</fo:flow>
```

Now (see Figure 9–4) the results are much more in line with expectations!

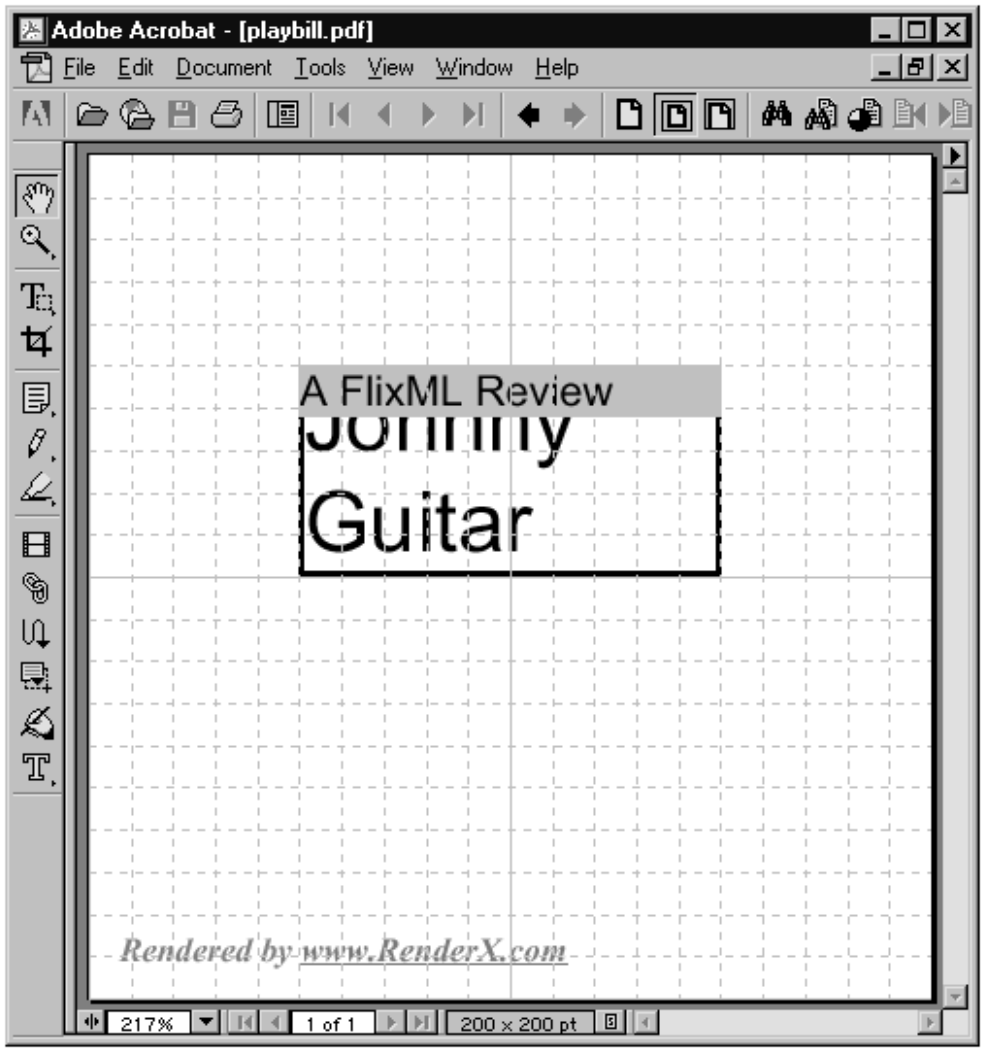


**Figure 9-3** The same XSL-FO-to-PDF document as shown in Figure 9–2, here “enhanced” by the addition of a before region. But this doesn’t look like it’s following any standard definition of “before,” does it? (Looks more like “on top of.”) See the text for an explanation.

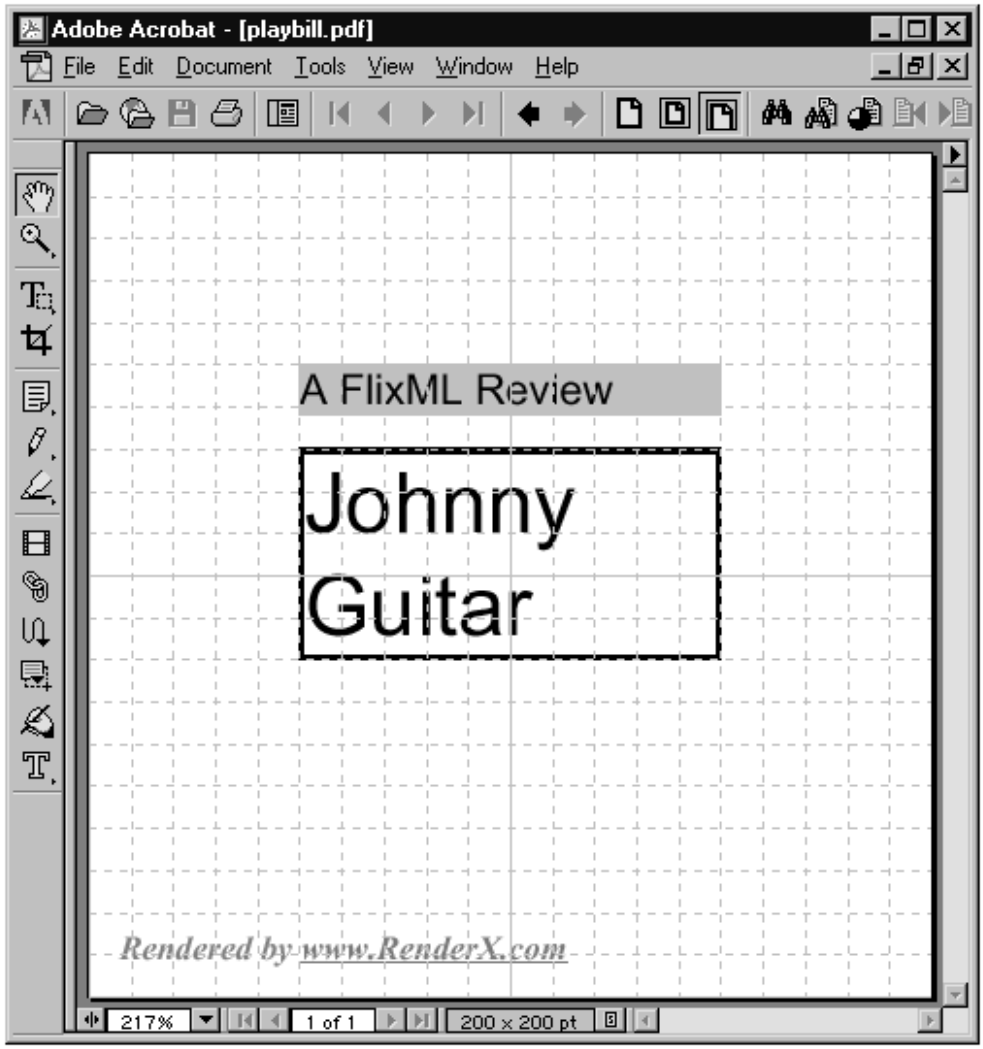


**Figure 9-4** The body region of Figure 9–3 fixed up, by assigning it a top margin at least as large as the before region's extent. Notice the silver background to the before region, which is not 20 points high (the size of the region's extent). Rather, it's *10* points high (the height of the `fo:block` element within which the text is enclosed).

Things look like they're starting to shape up in some reasonable fashion now, so let's add code for the other three regions: after, start, and end. First, add them to the simple page master itself:

```
<fo:simple-page-master master-name="page-review"
   page-width="200pt" page-height="200pt"
   margin="50pt">
   <fo:region-body region-name="reg-body"/>
   <fo:region-before region-name="reg-before"
     extent="20pt"/>
   <fo:region-after region-name="reg-after"
     extent="20pt"/>
   <fo:region-start region-name="reg-start"
     extent="5pt"/>
   <fo:region-end region-name="reg-end"
     extent="5pt"/>
</fo:simple-page-master>
```

Note that this sets the extents for both the start and end regions to 5 points. (And recall that for these two regions, the `extent` attributes establish *width*.)

There's no need to make any further changes to the general structure of the page (which is what the simple page master defines), so we can move on to actually putting some content in those regions. As with the before region, we'll use static content; the `fo:page-sequence` element (as set up in the XSLT stylesheet) now looks like this:

```
<fo:page-sequence master-name="seq-mast-cover">
   <fo:static-content flow-name="reg-before">
     <fo:block font-size="10pt"
       background-color="silver">A FlixML Review</fo:block>
   </fo:static-content>
   <fo:static-content flow-name="reg-after">
     <fo:block font-size="8pt"
       background-color="silver" text-align="center">
       Copyright <xsl:value-of select="../@copyright"/>
     </fo:block>
   </fo:static-content>
   <fo:static-content flow-name="reg-start">
     <fo:block font-size="8pt"
       background-color="silver"
       height="60pt">S</fo:block>
   </fo:static-content>
   <fo:static-content flow-name="reg-end">
     <fo:block font-size="8pt"
       background-color="silver"
```

```
      height="60pt">E</fo:block>
   </fo:static-content>
   <fo:flow flow-name="reg-body">
     <fo:block font-size="20pt"
       border-style="solid" border-width="1pt"
       margin-top="20pt"
       margin-left="5pt"
       margin-right="5pt">
       <xsl:value-of select="."/>
     </fo:block>
   </fo:flow>
</fo:page-sequence>
```

As you can see, the `region-after` will contain the copyright date of the FlixML review, and the `region-start` and `region-end` will contain the literal characters "S" and "E," respectively. Notice, too, that the latter two regions' heights have been set to an arbitrarily large 60 points. (In theory, this should extend silver bars to the left and right down the length of the body region.)

After cranking this revised stylesheet through an XSLT processor, and the resulting XSL-FO document through Xep, we see the results in Figure 9–5.

What happened to the silver bars at left and right?

What happened to them is the same thing that happened to the silver bars at the top and bottom: They take up only as much space as required by the `font-size` attribute of the `fo:block` element, regardless of the value of the corresponding `height` attributes.

This is one of your first lessons in a, well, I don't want to say *nasty* reality about using XSL-FO… let's just settle for a *surprising* reality. Which is this: There may be not just one element or attribute you need to look at, but *several*.

In this case, it helps to know that `fo:block` elements can have not only `font-size` and `height` attributes but also a `line-height` attribute. This specifies the "vertical space taken up by a line of type," no matter how large or small the actual `font-size`. Replacing the `height` attributes in the corresponding `fo:block` elements in the page sequence with `line-height` attributes (with a value of 60) instead, this portion of the result tree now looks like the following:

```
<fo:static-content flow-name="reg-start">
   <fo:block font-size="8pt"
     background-color="silver"
     line-height="60pt">S</fo:block>
</fo:static-content>
<fo:static-content flow-name="reg-end">
```

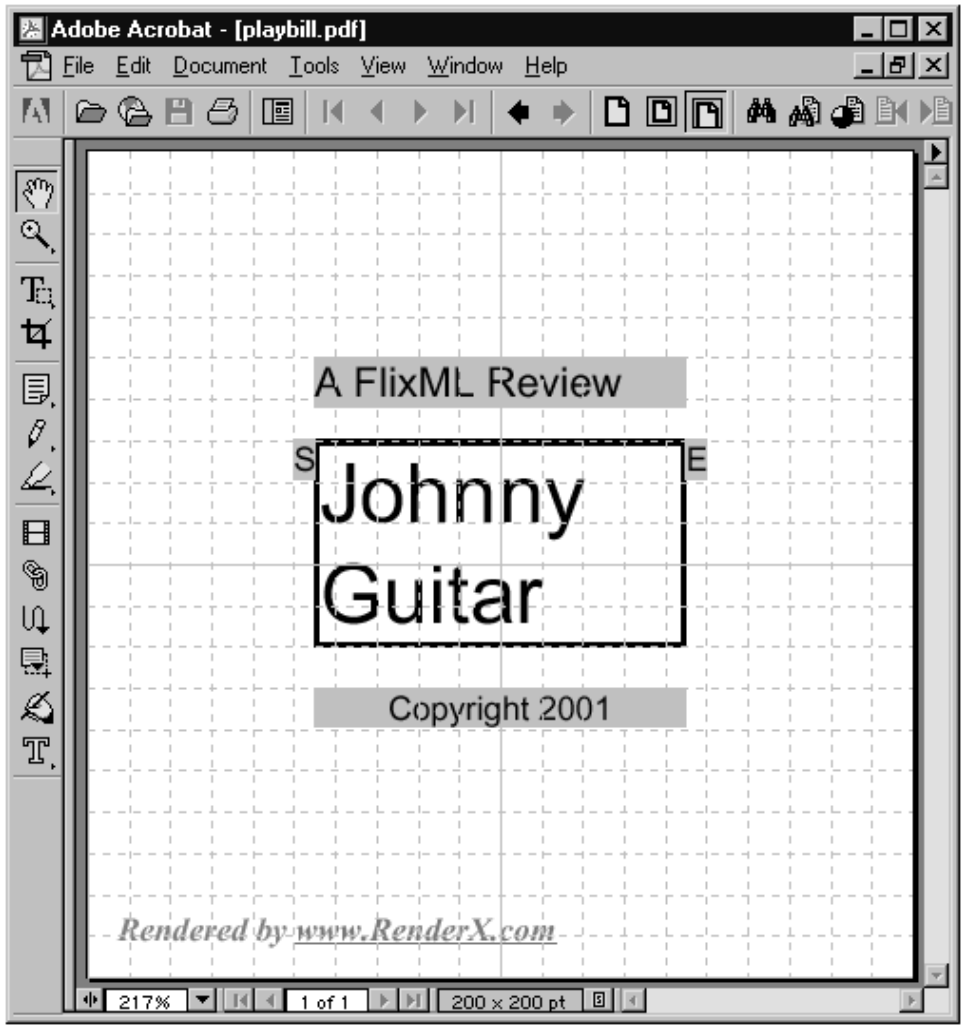


**Figure 9-5** Same output as in Figure 9–4, “enhanced” by the addition of start, end, and after regions (left, right, and bottom respectively). Look back at Figure 9–4, and observe not only that the new regions have been added but that the *existing* ones have “shrunk” to accommodate the new ones, by 5 points at the left and right. The body has also shrunk by 20 points to accommodate the region-after, but that’s not obvious here because the text in the body doesn’t fill the entire region.

```
    <fo:block font-size="8pt"
      background-color="silver"
      line-height="60pt">E</fo:block>
</fo:static-content>
```

And the outcome of the usual XML-through-XSL-FO-to-PDF cycle now produces a document like the one shown in Figure 9–6.

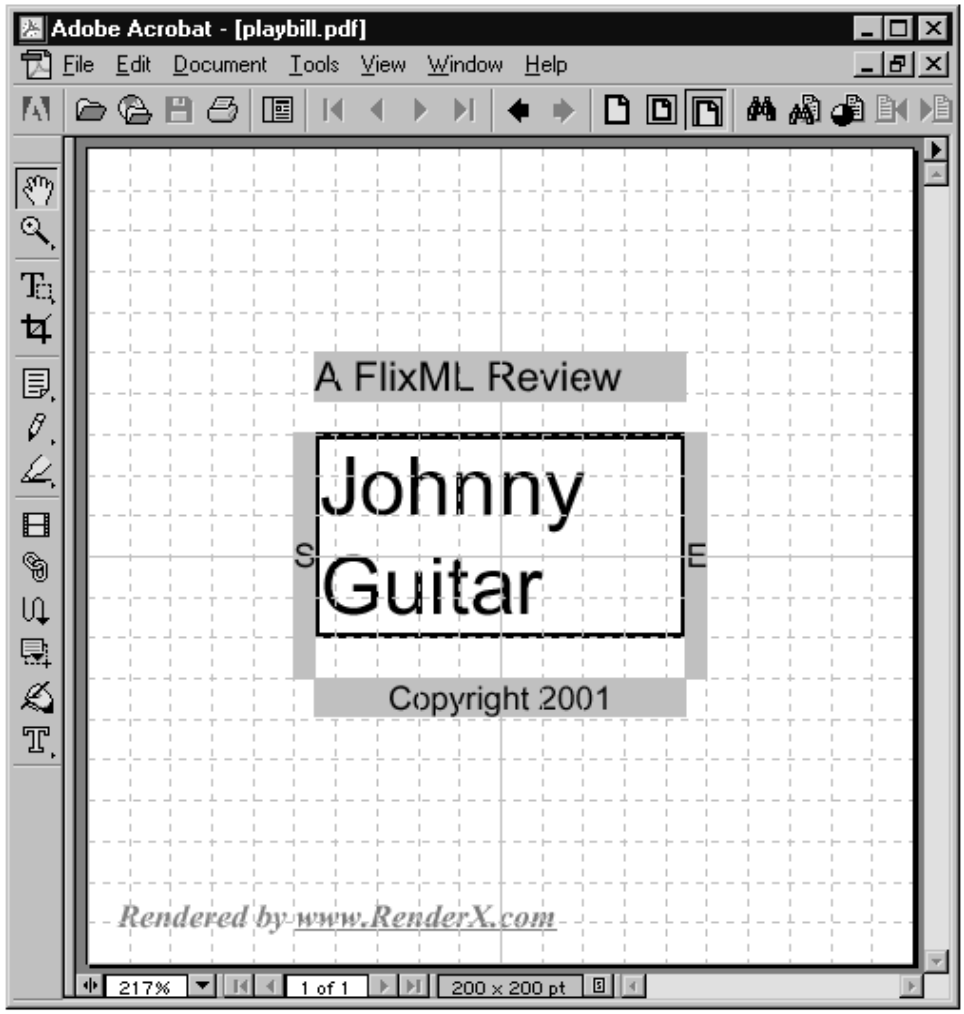


**Figure 9-6** Full-length silver backgrounds added to the start and end regions (left and right, respectively), using the `line-height` instead of the (simpler and more intuitive) `height` attributes to the corresponding `fo:block` elements.