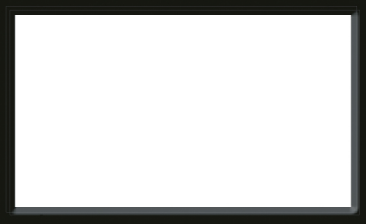
HTML FIXES FOR KINDLE

AARON SHEPARD



HTML Fixes for Kindle

Please Note!

This is an emulation of the original book as it appears on Kindle. For this reason, it purposely breaks some conventions of printed books.

HTML FIXES FOR KINDLE

Advanced Self Publishing for Kindle Books,
or Tips on Tweaking Your App's HTML
So Your Ebooks Look Their Best

By Aaron Shepard

Shepard Publications
Bellingham, Washington

POD Version 2.9
(Ebook Version 2.8)

Aaron Shepard is a foremost proponent of the *new* business of profitable self publishing, which he has practiced and helped develop since 1998. Unlike most authorities on self publishing, he makes the bulk of his living from his self-published books—not from consulting, speaking, freelance writing, or selling publishing services. He lives in Bellingham, Washington, with his wife and fellow author, Anne L. Watson.

The Kindle Publishing Series
From Word to Kindle ~ Pictures on Kindle ~ HTML Fixes for Kindle

The Print Publishing Series
Aiming at Amazon ~ POD for Profit ~ Perfect Pages

The Kidwriting Series
The Business of Writing for Children ~ Adventures in Writing for Children

For updates and more resources, visit Aaron's Publishing Page at www.newselfpublishing.com

Getting Started

If you're publishing on Kindle, chances are you want your book to look its very best. But that often requires a task not performed by most Kindle authors—namely, refining the HTML you submit.

Many authors follow Amazon's recommended approach to create their Kindle book: Compose in Microsoft Word, export to HTML—the language of ebooks and the Web—then submit to Amazon KDP. It's an approach I discuss in detail in my earlier book *From Word to Kindle*.

Done correctly, formatting in Word or another word processor can bring you maybe 80% of the way to a well-formatted ebook. But if you want your book to look as good as it can, there's that other 20% to worry about. Now, perfection is not really possible on the Kindle, given the quirks, bugs, and limitations of the platform itself. But you can come closer—if you're willing to tinker with HTML.

Does that idea scare you? It doesn't need to. You *don't* have to read that language (though it certainly won't hurt if you do want to learn). All you need is to be able to recognize small bits of code I'll point out to you, and shift them or replace them with other bits. And all these changes can be done with find-and-replace operations. In fact, if you set up a master macro as I recommend, the whole job can be finished in seconds.

Here are some of the things you can accomplish through changes in HTML.

• Adjust bookmarks so headings retain proper formatting when jumped to.

• Remove settings that stop the user from choosing their own.

• Keep fonts from appearing much too small or much too large when the book is opened.

• Make sure indents and other spacing stays relative to larger and smaller font sizes.

• Avoid line breaks that leave short words dangling at the ends of lines or paragraphs.

• Make up for features lost in translation from your word processor, like nonbreaking hyphens.

• Stop "ghost hyphens" from appearing in the middle of words.

• Turn off automatic hyphenation in headings, poetry, or software code.

• Keep pages of text from disappearing for some users.

• Block the Kindle from applying its own defaults in place of your settings.

Since Word is the most common tool for generating HTML for Kindle, I'll focus mostly on Word's exported code, making this book a perfect companion to my earlier one. Instructions are based on desktop Word versions from 2003/2004 to 2010/2011.

The general principles I provide, though, should still help if you're using a different word processor, or a specialized ebook tool like Scrivener, Vellum, or Jutoh, or even if you're writing HTML directly. In fact, I'll

provide some tips on Kindle formatting with HTML that you won't find even in books dedicated to that approach.

For updates and related materials, please visit my Publishing Page at

www.newselfpublishing.com

While there, be sure to sign up for my email bulletin, so you'll know when I have anything new, including revisions of this book. And though I'm not able to provide technical support or consulting, I'm always glad to receive comments, as well as suggestions on how my books might be improved.

WARNINGS!

Everything in this book has been tested, but not all conditions and variations can be foreseen. When trying out suggested operations, always test first on a file copy you can afford to lose!

Some Kindles, when breaking a line in the middle of a Web address or HTML code sample, may insert a hyphen where one doesn't belong. If you're not sure whether a hyphen you see in this book should be there, change your font size to move or remove the line break and see if the hyphen remains.

Even if a Kindle lets you copy and paste code samples and Web addresses directly from this book, don't do it! They may include extra, invisible formatting characters to help them display better on screen.

1
Working with HTML

HTML and Kindle

This is not the beginning of the book. If it opened here automatically, please page backward for important information.

HTML is the language of Web pages, and Kindle books are basically Web pages divided into screen-size pieces. Though Amazon lets you submit your book as a Word document, PDF, or plain text, Amazon must convert these formats into HTML before creating your Kindle book. It only makes sense, then, to do that conversion yourself, so you can control the results.

There are actually two Kindle book formats. The older one is called Mobipocket—MOBI, for short—after the company that created it and that was then acquired by Amazon. This is a rudimentary, outdated format, using only a small subset of HTML and therefore allowing only very limited formatting.

Starting in 2011, Amazon began moving to Kindle Format 8, or KF8. This format is more modern, accommodating much more of HTML and its formatting capabilities. It was used first on newer kinds of Kindle like the Fire and Paperwhite, and Amazon has now migrated it to some older kinds as well. But this is probably not even possible for some of the oldest, which are still in use. And oddly enough, in 2015, Amazon still uses MOBI for its Look Inside feature.

When the Kindle converter processes your book, it converts it to both MOBI and KF8 formats. Both formats are included in Amazon's preview files. But when a published book is sent to an individual Kindle, Amazon includes only the format that will work best.

There are some kinds of books—called *fixed-format*, including mostly comics and children's picture books—that usually don't work in Mobipocket format at all. Amazon restricts their sale to certain Kindles. Other books may include certain formatting that will show up on KF8-capable Kindles but be missing on others.

The important point here is that, even though KF8 offers advanced formatting, you cannot take advantage of it if you want your book to be readable on *all* Kindles—at least, not without devising fallbacks or complicated workarounds. Despite the MOBI format being primitive and outmoded, we're stuck with it for some time to come.

This book, then, like *From Word to Kindle*, takes a minimalist approach: I stick to HTML that will work on *any* Kindle, or at least do no harm—even if that sets a sharp limit on what can be done.

Update!

Starting in 2015, Amazon has been slowly introducing a third-generation format for those newer Kindles that can support it: KFX (with the *X* pronounced “Ten”), including its most important visible component, “Enhanced Typesetting.”

Unlike KF8 or even Mobipocket, KFX is based not on HTML directly but on a translation of it into a proprietary format. Controlling your HTML, then, can improve your results as always, but fewer of your instructions may be accepted without change.

For more information, please see my Publishing Page, and especially my Publishing Blog.

www.newselfpublishing.com/blog

HTML Export

Working with HTML begins with making sure to export it correctly. This is especially important when working with Word, because the default setting will export an incongruous blend of HTML and proprietary instructions that are useful only to Word in reimporting the document. So, let me review the procedure I described in *From Word to Kindle*. (And keep in mind that in Word, unlike in some other apps, you use the "Save As" command to export to HTML—so here I use "save as" and "export" synonymously.)

The exact steps will vary according to your version of Word, but you'll be saving as "HTML" or "Web Page" or the like. In the dialog box you get, you then want to specify the variety that will include the *least* code. This may be designated as "Web Page, Filtered" or by an option like "Save only display info into HTML."

If in doubt about which option to choose, save in alternate ways, then choose the one that creates the *smallest* file. Also, when saving the best way, you should *not* see any auxiliary files with the .xml extension.

•

Word offers a number of "Web Options" related to HTML export. The defaults for most of these options are fine—while other options seem to have no effect anyway!

If you'd like to know, though, the default text *encoding* for HTML from Word for Windows is "Western

European (Windows)," also known as Windows-1252. This is an encoding very close to "Latin-1," officially known as ISO 8859-1. For the Mac, the default is "Unicode (UTF-8)." (Some Windows apps refer to this as "UTF-8 without BOM." That's to distinguish it from UTF-8 such as written by Microsoft products on Windows, which still follow the obsolete practice of starting the file *with* a byte order mark.)

For the Kindle, either of these encodings is just fine. The Kindle's own native encoding is now UTF-8 (without BOM)—but the Kindle converter is more than ready to translate from one to the other.

By the way, checking the option "Always save Web pages in the default encoding" will mean that Word *ignores* any encoding choice of yours that differs from the default!

•

If you've exported the right way, any excess code still in Word's HTML isn't likely to amount to much, so it's OK to ignore it. Even so, you may find it worthwhile to minimize the Styles info that Word converts and places at the top of the file.

The most important way to do that is so simple and effective that I recommend it in all cases: Turn off the Editing option "Keep track of formatting." This will prevent Word from creating a new style every time you apply direct formatting! That not only trims the code but may also simplify your code modifications.

The location of this option varies in different versions of Word. In Word 2010 for Windows, for

example, find it at File > Options > Advanced > Editing Options. In Word 2011 for Mac, it's at Word > Preferences > Authoring and Proofing Tools > Edit.

The other way to minimize Styles info is to delete styles embedded in the document but not in use. Of course, before you delete a style, you want to make sure it really hasn't been applied to any text. If in doubt, you can search for the style with Word's Find and Replace. Select "Style" from the Format menu in the dialog box, then choose the style from the list of all available. Make sure the "Find what" box is empty.

Occasionally, you may see that Word is exporting a style that you can't find in the app's Style lists, even when viewing "All styles." If so, try looking in the Organizer.

HTML Editing

It's *possible* to modify HTML either in Word or in a general text editor like Notepad (Windows) or TextEdit (Mac). But that can be tricky or else not very efficient. You'll find the job easier and more pleasant if you use a tool designed for the job: a code editor.

Windows has many code editors available, both for free and for pay. The most popular is the free Notepad++, available here:

www.notepad-plus-plus.org

The Mac too has a choice of code editors, including what some consider to be the best HTML editor on any platform: BBEdit from Bare Bones Software. (BBEdit has been my own HTML editor since around 1996.) Find it here:

www.barebones.com

Don't confuse a code editor with a consumer app for building Web sites. Most of those apps take control of your code instead of letting you do what you want. That's not likely to work.

•

There are a number of advantages to editing your HTML with a code editor instead of a word processor or simple text editor. For one thing, you don't have to worry about the invisible characters that word processors insert in

text when used normally—characters that can wreak havoc in other apps.

A code editor will not only refrain from adding such characters, it will typically discard them from any text you paste in from elsewhere. And if any do get through and cause odd behavior, the code editor should have a command to seek out and remove them. BBEdit, for example, lets you "Zap Gremlins."

A code editor also helps handle the issue of "smart" or "curly" quotes vs. straight. Smart quotes are your friend! You nearly always want these in your *text*, because they are part of the distinctive look and feel of a book.

"word"

it's

On the other hand, you *don't* want them in your HTML code! To avoid error, all quotes in your code must be straight!

style='text-align:center;'

In both word processors and code editors, smart quotes can be turned on or off. But by default, they're *on* in a word processor, and *off* in a code editor—and to avoid confusion, it's best to leave them that way. Using

one app just for text and another app just for code helps keep things straight.

Of course, there can always be times when you want to insert a straight quote while in your word processor, or a smart quote while in your code editor. In those cases, you can change your option briefly, or use a keyboard equivalent for your system, or use Character Viewer (on a Mac), or even just cut and paste from the other app.

•

Another advantage of a code editor is that its search capabilities will generally be more sophisticated than those of a word processor or text editor. This will generally come in the form of a search language called *regular expressions*—often shortened to *regex* or *regexp* and other times nicknamed *grep*, after a command in one of the first programs that featured it. Though Word lets you use some of this language if you select "Use wildcards" in its Find and Replace dialog box, it's a watered-down version, and few other word processors will come close to even that.

Because it uses a specialized language, grep searching must be specially turned on when you need it—and turned off when you don't! Generally, your code editor's search dialog box will give you a way to select this. BBEdit, for example, has a simple "Grep" checkbox. In Notepad++, you select the "Regular expression" search mode—and for the grep searches in this book, leave *off* the option ". matches newline."

The language of grep search is extensive and powerful, and details of it can vary from one app to the next. The grep operations in this book, though, use basic, common features, so they're likely to work almost anywhere. Still, when using grep, it's *always* important to first test the operation carefully and only on a file you can easily replace. A small error in grep can wreck an entire document in less than a second!

Though you don't need to know the grep language to run the operations, knowing a bit of it may help you to understand their basic workings, to guard against errors in copying, and even to customize them. So, here are some basics.

• Standard grep is case-sensitive. So, "a" would normally match a lower-case letter *a*, while you would need "A" to match the capital. But this is *not* true in some code editors—including BBEdit and Notepad++—which make grep searches case-*in*sensitive unless you choose otherwise. Still, if you construct operations in case-sensitive grep, they will generally work either way—so that's what I've done here.

• Characters within square brackets represent a *choice* of character. For instance, "[ibu]" will match either *i*, *b*, or *u*. In case-sensitive grep, "[aA]" will let you match either a lower-case or capital *a*.

• A hyphen between characters in square brackets represents a range. So, "[a-zA-Z]" would match any letter from *a* to *z*, either lower-case or capital, while "[0-9]" would match any digit.

• A simple period will match *any* character. So, "b.d" will match *bad*, *bed*, *bid*, *b3d*, or even *b/d*. The one

exception is that it normally won't match a line break—but your code editor may have an option for it to match that too. This, for example, is what Notepad++ means by the option ". matches newline." If you do see such an option, make sure it's *off* for the operations in this book!

• A caret (^) in front of either a character or a character set within brackets means *not*. So, "^i" would match any character except *i*; "[^ibu]" would match anything except *i*, *b*, or *u*; and "[^a-zA-Z0-9]" would match anything but a letter or digit.

• Characters or bracketed sets may be followed by a *quantifier* that specifies the number of matches desired. For instance, a plus sign means "one or more," so that "[0-9]+" would match any number of digits in a row. An asterisk means "zero or more," so that "[0-9]*" would match any number of digits or none at all! A question mark means "zero or one."

• By default, grep searches are "greedy." Instead of looking for the *least* number of characters that will match a pattern, grep normally looks for the *most*, stopping only at a line break. This is one reason grep can be treacherous! One way to prevent a greedy search is to specify a character the match should *not* include. Another is to use a *non-greedy quantifier*—a normal quantifier followed by a question mark.

• Since the grep language gives special meaning to some punctuation marks, you have to set off those marks specially when you instead mean them as regular text. You do this by preceding them with a backward slash (\). For example, "\?" would mean a *real* question mark instead of a quantifier.

• On the other hand, a backward slash in front of a *letter* means it's instead meant as a special character. For instance, "\t" matches a tab character. Letters used this way *are* case-sensitive!

• To include part of a matched Find in the Replace, put parentheses around that part of the Find pattern, then put its position number in the Replace with a backward slash in front. As an example, look at the following.

Find: ([a-zA-Z]+)[0-9]([a-zA-Z]+)

Replace: \1\2

This would look for one or more letters on each side of a digit and then replace everything with just the letters.

And now you know grep!

HTML Processing

Every single one of the fixes I'll suggest for your HTML file can be handled by one or more find-and-replace operations. If you use a number of them, you'll find it best to combine them in a single *macro*—a custom command that combines multiple steps. Macro features are often provided by code editors, or you may be able to use a standalone macro recorder.

The kind of "master macro" I'm recommending may take a while to set up, but it takes next to no time to apply later on. For example, I currently have a "Fix Kindle" macro in BBEdit that performs over a hundred find-and-replace operations, processing the HTML of a medium-size novel in a couple of seconds.

Yes, even starting with HTML I've exported from Word, I can wind up with fairly clean and Kindle-optimized code in no time at all! With this kind of speed, I'm seldom tempted to edit text in the HTML file, instead of back in the original document, where I'm supposed to. It's plenty easy to edit the original, run off a new HTML file, and fix it up. (Sorry, I can't share my entire macro with you, because it's customized for my own work!)

•

In some apps, you can assemble macros in a special editor. BBEdit, for example, has its "Text Factories," which is what I use for my Fix Kindle macro. After opening a new Text Factory, you can add, move, and

remove “actions,” defining each one’s operation from a pull-down menu and employing appropriate options. (The operation most commonly needed is “Replace All.”) You can even drag in actions from another Text Factory! After saving, the Text Factory can be applied to your active BBEdit document—or to just a selected portion of it—from the Text menu.

In another kind of macro feature, you simply “record” the steps as you perform them and then replay them later. BBEdit offers this too, as an alternative to Text Factories, by way of the Mac’s AppleScript Editor—though I have not found this feature particularly helpful.

Now, obviously, it might be a challenge to flawlessly record a macro with dozens of steps. Luckily, you don’t need to. Once you record a macro with a single find-and-replace operation, you can usually then edit the macro to add others.

For example, the following line of code from an AppleScript is for a BBEdit find-and-replace operation that replaces “Shepherd” with “Shepard” throughout a document.

```
replace "Shepherd" using "Shepard" searching in text 1
    of text document 1 options {search mode:literal,
    starting at top:false, wrap around:true,
    backwards:false, case sensitive:false,
    match words:false, extend selection:false}
```

Opening the script and seeing this line in the Mac’s AppleScript Editor, you could simply copy and paste it

as many times as you wanted, plugging in new Find and Replace terms each time, and even changing other settings. In fact, the code you copy and paste doesn't even have to come from the same macro. You could create a number of them independently and then merge them.

•

When devising your macro, there are several safeguards you must employ to avoid potential gotchas.

• Code editors typically search forward from the current cursor position. If you "Replace All," this may still act only from the cursor to the end of the file. To make sure the entire file is processed, you can usually turn on an option to "Wrap around"—meaning the operation goes to the end and then continues from the beginning. If you can't select such an option, make sure either that wrapping around is the default or that your cursor starts at the head of the file.

• Avoid writing your replacements in a way that causes errors if you happen to run the macro on a file more than once. For instance, if you replace "1" with "11," running the macro twice will give you "1111"! To prevent this, you might set the find operation to "Match whole words only." Or you might include one or more characters of the HTML code that you know comes before and/or after, even if it's just a punctuation mark. Or you could add a follow-up operation to replace "1111" with "11."

• Make sure the text you're trying to find hasn't already been changed by an earlier operation. For

example, if you've already changed all exclamation points to question marks, you'll get zero results on a search for "Error!"

• Don't replace terms that are also found in text. For example, don't replace the font name "Georgia" with "Arial" if you're writing about the Peach State. Here, too, try to include in your Find some HTML code from before or after. Or else modify the phrase in your source document in some way that will avoid a match—like replacing a regular space with a nonbreaking one.

I admit it: I've committed that last kind of error in some versions of my own books, including this one. Very embarrassing!

HTML Basics

It's much easier to read a language than it is to speak it. I'm not going to ask you to compose HTML, and really not even to understand it. But if you can recognize some of its basic structures, this will help you work with the code and avoid introducing errors.

So, now that you've exported your HTML and acquired a code editor to process it, let's take a look inside.

•

Basic HTML is a collection of tags enclosed by angle brackets (< >). Most of the tags come in pairs—an opening tag and a closing one—that enclose text elements. So, for example, a paragraph of text might be enclosed by paragraph tags as shown here, with the closing tag distinguished by a slash.

```
<p>Very short paragraph.</p>
```

Other enclosed units you might see include headings (<h1>, <h2>, and so on), list items (<li>), and *divisions*, or "divs" ( ). What's a div? Basically a catch-all for any blocks of text not defined by other tags.

While some tags enclose and define basic units of text, others can enclose portions of text *within* those units and specify formatting. Simple formatting tags include those for italics (<i>), bolding (<b>), and

underlining (<u>). The span tag () can be adapted for various purposes.

Not all tags are paired or enclose text. For example, tags for line breaks (
) and images (<img>) have no closing tags. Tags for *anchors*—the HTML equivalent of bookmarks—are paired and *may* enclose text but don't have to.

Opening and standalone tags can include one or more *attributes* with associated *values*. For instance, we might see the following in a paragraph tag, signifying that the paragraph should be justified:

```
<p align="justify">
```

As you can see from this example, parts of HTML are pretty easy to figure out, once you take a look.

HTML can also include comments that are meant only for reference and are ignored by the browser or ereader. Though they can take up one line or many, they are always set off the same way at beginning and end:

```
<!-- This is an HTML comment. -->
```

You can edit comments as you like, add new ones, or delete old ones without any effect on your ebook. In regard to deletion, the one exception is when comment markers are used to hide modern code from obsolete browsers—an old trick that Word still uses. You don't want to delete the code between the markers! But the difference in such comments should be easy to spot.

HTML comes in a number of varieties. Besides being newer or older, versions are distinguished by levels of *strictness*—conformance to the ideals of HTML's developers. (That doesn't mean, though, that any version doesn't have its rules!)

Word, for example, exports HTML in a version called HTML 4.01 Transitional—an older version, and the most lenient one still in common use. On the other end of the spectrum, Kindle books and other ebooks today use a very strict kind of HTML called XHTML. That means that the Kindle converter, as part of its process, translates Word's HTML into XHTML. Be glad you don't have to do that yourself!

•

In a word processor or page layout app, you can insert two basic kinds of line breaks: a paragraph mark to end a paragraph and start a new one, and a manual line break to force a new line *within* a paragraph, list, verse, or such.

When you export to HTML, these line breaks are converted to *tags*: <p> for paragraph mark and
 for manual line break. And those are what mainly determine forced line breaking in your Web page or ebook.

But the HTML code you edit also contains its *own* line breaks, apart from those tags. The line break *character*—often called *newline* or *end-of-line* (EOL)—is inserted with the Return key, just as you would add a paragraph mark in a word processor; and also like the paragraph mark, it's invisible unless you choose to show it. Its job

is to split the code into separate lines for the sake of convenience in editing, so you're not dealing with one massive block of text.

When an app exports HTML, it normally inserts these line break characters in logical places, like between units enclosed by paragraph tags, and before or after line break tags. This helps keep the code organized. Extra line break characters may be added *within* units to keep lines from getting too long.

The important thing to remember about these line break *characters* is that they do *not* produce line breaks in the Web page or ebook. Only tags can do that. Instead, line break characters are read in the Web page or ebook as simple spaces. And if one appears in code beside a *real* space, the two spaces will be read together as one. So, the line break character can be sprinkled liberally within HTML code without affecting your ebook at all.

In this book, we need to discuss both line break *tags* and line break *characters*—so we'll have to be careful to keep them straight. (We'll also discuss *automatic* line breaking—commonly called *word wrap*—and how to encourage or discourage breaking at specific spots in your text.)

•

Nowadays, HTML as a coding language rarely stands alone. It is almost always accompanied by a companion language, CSS, which stands for "Cascading Style Sheets." CSS is used to specify formatting, with capabilities that HTML by itself was never designed for. On the Kindle, support for CSS is very limited in the

older Mobipocket format but is a major feature of the newer Kindle Format 8.

CSS consists of instructions or *declarations* that apply to specific text elements enclosed by the HTML tags. One place these declarations can appear is within an opening tag itself. For example, you might see a paragraph tag like this:

```
<p style="text-align: justify; margin-bottom: 12pt;">
```

As you can see, the declarations are together formatted as the value of an HTML attribute called "style," in that way integrating with the HTML. The declarations themselves consist of *properties* and their associated *values*. The declarations here tell the browser or ereader that the paragraph should be justified, with 12 points of blank space coming after.

Placing CSS in the tag is something like applying formatting directly to a paragraph in a word processor. But as in a word processor, you can also apply multiple formatting instructions you've grouped under a joint name. In a word processor, we'd talk about assigning a *style.* In CSS, we call it a *class.*

For example, the following might appear in the HTML document's *head*—a section appearing before your text, which includes information and instructions relating to the document. This example sets up a paragraph class named "plain," using the same properties we previously specified within the tag. Again, the CSS is integrated with the HTML, this time by enclosing the code within HTML style tags.

```
<style type="text/css">
p.plain {text-align: justify; margin-bottom: 12pt;}
</style>
```

With this code in the document head, all we need to do to assign the class to a paragraph is to place a class attribute in the opening paragraph tag.

```
<p class="plain">
```

Much cleaner!

CSS declarations placed in the document head are said to be *embedded*, while declarations placed within tags are *inline*. Generally, when a word processor exports HTML, both inline and embedded CSS will be included. Your source document's style definitions will translate to embedded CSS, while direct formatting will translate to inline.

There is one more kind of placement for CSS, and that's *external*. CSS declarations are often collected in a separate document called a *stylesheet*. In the original HTML document, you would still see class assignments and inline CSS for direct formatting. But in place of embedded CSS, you would see a line like the following, linking the document to the stylesheet file:

```
<link rel="stylesheet" type="text/css"
href="stylesheet.css">
```

In your exported HTML, you're unlikely to see this arrangement unless you're working with dedicated ebook software. Word processors and text editors in general won't write separate stylesheets. But you *will* see them if you "unpack" Kindle previews for inspection—a procedure I'll discuss later.

Like HTML, CSS allows comments used only for reference. Here, though, the markers are different.

```
/* This is a CSS comment. */
```

With these comments too, you're free to edit, add, or delete as you like.

Despite CSS being a language of its own, it's so intertwined with HTML that we generally use that term to refer to them together. So, in this book, I'll refer to CSS separately only when that's *all* I'm talking about.

•

Different versions of HTML call for slightly different ways to write code, while even a single version can allow latitude in how it's written. Often there's a "best" way to write it, but other ways are tolerated. And sometimes format details are simply left to the coder's preference.

When constructing find-and-replace operations, you need to be conscious of such possible variations so you can accurately follow the style of the HTML you're working on. Otherwise, your searches may miss matches, or you may introduce inconsistencies or even errors.

In the find-and-replace examples in this book, I try to show you exactly the way the code will appear in Word's exported HTML. But if your code comes from another app, there will likely be differences.

Here are some variations to watch out for.

• Best practice in HTML is to keep all tags, attributes, and values in lower case (unless a value includes text that may be displayed). Still, at least some versions of HTML allow caps anywhere.

<p> *or* <P>

• As I said, some HTML tags are not paired to be opened and closed. But in XHTML—the stricter variant of HTML—the idea of a tag *not* being closed is intolerable. So, all tags that are normally standalone are either paired with closing tags or, more commonly, are closed *within* the same tag, by ending it with a slash. Like so:

 or

Note also that the slash may or may not be preceded by a space!

• In word processors, we're used to inserting and viewing a line break at the *end* of a line. But the HTML line break tag (
) actually starts a *new* line in the document—and at the *beginning* of the corresponding line of code is where it logically should be positioned. Still, HTML tolerates either placement—and in fact, any placement at all.

• In *very* old styles of HTML, you might see opening paragraph tags without closing ones—and these might be placed either at the beginning of each paragraph or at the end! Not a good practice, but still acceptable in some HTML versions.

• HTML attributes are best enclosed by double quotes—straight, of course. But single quotes are also accepted, and in most cases, even *no* quotes.

<p align="justify"> *or*

<p align='justify'> *or*

<p align=justify>

Word, for example, prefers to use no quotes, with single quotes as a second choice.

• In an HTML document, not all characters are allowed within your text. This can be either because the characters are reserved for HTML code or because they're not supported by the HTML document's assigned *encoding*—the way its text characters are generated from raw computer data.

To get around this, HTML can represent a number of punctuation marks and special characters as *entities*, derived from abbreviations or assigned numbers. So, you may see substitutes like these:

< *or* < (<, less-than sign)

" *or* " (", straight double quotation mark)

  or   (nonbreaking space)

é *or* é (é, small e with acute accent)

```
&copy; or &#169; (©, copyright mark)
```

Note that entities always start with an ampersand (&) and end in a semicolon (;), while numbers are always preceded by a number sign (#).

Actually, with the UTF-8 encoding most commonly used in HTML today, you're not likely to find characters the encoding doesn't allow. But some HTML—including the default export from Word for Windows—is still written in one of the older, narrower encodings. And some HTML in UTF-8—including the default export from Word for the Mac—uses entities anyway as a carryover from the past.

• With CSS, there are two basic styles of writing the code. One is a "loose" style, inserting spaces after punctuation, as in normal text.

```
style="margin-top: 0; font-family: Caecilia;"
```

The other is a "tight" style, with spaces left out.

```
style="margin-top:0;font-family:Caecilia;"
```

Also, in some CSS, you'll find a semicolon at the end of each declaration, including the last one in a series, as shown above. In other CSS, you'll see the semicolon used only *between* declarations—*not* after the final one—as shown below.

```
style="margin-top: 0; font-family: Caecilia"
```

• Most HTML attributes have default values. When no different value is desired, an attribute can be either included in the code or just left out. (As we'll see, though, leaving it out can cause a problem when the default in HTML doesn't match the one on Kindle!)

•

You don't need to become an HTML expert to refine the code for your Kindle book—but the more you know of it, the easier you may find tracking down problems and adjusting things to your liking. If you'd like to further explore HTML, I suggest getting a copy of *HTML5 and CSS3*, by Elizabeth Castro and Bruce Hyslop.

HTML Checking

When you're fixing HTML, one thing you want to make sure is that you don't introduce errors. Even a tiny mistake in spelling or punctuation in your code can make your entire document useless. And some such errors can be nearly impossible to find just by looking.

To check for errors, you need an HTML *syntax checker.* If you're lucky, your code editor will have one built in, as BBEdit does. If it doesn't, it might still be able to host a third-party plug-in. Farther afield, your Web browser may have syntax checking you can enable, offered as an advanced tool for HTML developers. And if all else fails, you can upload your code to a site like the W3C's Markup Validation Service.

validator.w3.org

Of course, the syntax checker may find errors other than ones you make, as your exported HTML may contain some to begin with. Generally these aren't crucial, and in fact, the Kindle converter seems designed to overlook common errors, at least in Word's HTML. Still, you'll want to correct any existing errors to keep them from obscuring or compounding your own.

•

The syntax checker works by matching your HTML against the rules for specific HTML versions. That

means it must first know which HTML version you're using!

That version information is supposed to appear at the very beginning of your HTML document in what's called a *doctype declaration*, or just *DOCTYPE* (all caps), for short. The one for HTML5, the current version of HTML, is quite simple, looking like the following. (Note that we're not sticking to lower case, this time.)

```
<!DOCTYPE html>
```

Older DOCTYPEs, though, are much longer and more complex. Here's the one for HTML 4.01 Transitional, the version exported by Word and used by many older Web sites.

```
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

The problem is, a doctype declaration doesn't always make it into the document. Word, for example, is one app that omits it from exported HTML. Ironically, this can make a syntax checker spew out hundreds or thousands of false alarms as it checks by the wrong set of rules!

Though some checkers let you select the version for checking from a menu, a DOCTYPE makes the selection automatic. So, if it's missing, one of your first steps in processing your HTML should be to insert it.

But what if you don't know which version of HTML your document uses? You'll have to try different ones, then see which of them your syntax checker seems to approve by reporting the fewest errors.

If your checker doesn't have a menu for version choice, testing for a version means first inserting its DOCTYPE in the document. Your code editor may have a command that does this for you. If not, you can type it in or else cut and paste from an authoritative source. Place it at the very start of your document *before anything else*.

Here are two sample lists of older available DOCTYPEs:

www.w3.org/QA/2002/04/valid-dtd-list.html

www.htmlhelp.com/tools/validator/doctype.html

Chances are, though, that your mystery version is HTML 4.01 Transitional, the version exported by Word. That's because it's still the most common type of HTML—and because anyone using a stricter version would be unlikely to leave out the declaration! So, try that one first.

•

Of course, after figuring out which DOCTYPE you need, you won't want to insert it manually each time you export HTML. So, this would be one of the first processing steps in your master macro.

As I said, the DOCTYPE appears at the very beginning of the document, which places it just before the opening HTML document tag (<html>). Adding it there would be easy with a simple find-and-replace. Here's what it would look like with an HTML5 DOCTYPE.

Find: <html

Replace: <!DOCTYPE html><html

The only problem is, we aim to write macros that can safely be repeated. And every time you repeat *this* one, it will add another copy of the DOCTYPE! To prevent that, you can *precede* this find-and-replace with another one that looks for the exact DOCTYPE you're about to insert—including any embedded line breaks or tabs—and removes it by replacing with nothing. That frees you to insert the DOCTYPE again.

The alternative is to substitute a grep operation that will work whether or not you've already inserted the DOCTYPE. For example, the following Find will select the HTML tag plus any DOCTYPE inserted previously, as long as the DOCTYPE does *not* include any line break. (That's a period in front, not a flyspeck.) The Replace would not change.

Find: .*<html

•

Considering the reputation of Word's HTML export, as well as how long ago it was introduced, it produces

surprisingly few outright errors in code. In fact, aside from the omission of the DOCTYPE for HTML 4.01 Transitional, there's only one error you're likely to see from an all-text Word document: Contrary to what's required by this older version of HTML, there's an attribute missing from the style tag found in the HTML document head. You can fix it with this:

Find: <style>

Replace: <style type="text/css">

You might also want to remove the attributes Word adds to the opening body tag—the one that marks the beginning of document text. Though they're proper for this version of HTML and no problem for Kindle, you shouldn't need them, and some may generate errors if you later convert your HTML or MOBI file to EPUB. They're easy enough to remove with a little grep.

Find: <body[^>]*>

Replace: <body>

Word can make other minor errors in code—especially in its image tags—but I'll discuss those as we go along.

HTML Cleanup

One function of a word processor that we usually take for granted is *word wrap*, which automatically breaks a line and begins a new one whenever text reaches the right margin. Without that function, text lines would simply run off the right side of your screen, stopping only when they got to a paragraph mark.

In a code editor, though, word wrap might be turned off by default, and in a primitive text editor, it might not even be offered! So, when a word processor exports to HTML, it might—for your convenience—add line break characters not only for organization but also wherever needed to keep lines from getting too long. The characters might be inserted after spaces, or even in place of them.

As I explained before, these line break *characters*—unlike the line break *tag* (
)—are there only to make code editing easier. On a Kindle reader, each such character appears as a simple space, and if it's next to a *real* space, the two are merged into one. So, the insertion of line break *characters* into HTML code makes no difference in how the book is displayed.

But when it comes to find-and-replace in your code editor, it's a different story. A line break character coming before or in place of a space can throw off your search by making you miss a match! So, before you start fixing HTML in your code editor, you have to make sure there are no line break characters in positions where they can cause trouble.

•

Even before you start removing line breaks, make sure word wrap is turned on in your code editor, so you don't wind up with text extending far off your screen. In BBEdit, for example, do that with View > Text Display > Soft Wrap Text. (Don't turn on Hard Wrap, which will insert even more line break characters!) In Notepad++, it's View > Word Wrap.

You should also make sure you can *see* whatever line break characters are there. If your code editor does not *always* show you such characters, it will at least offer the option. In BBEdit, you choose it with View > Text Display > Show Invisibles. For Notepad++, it's View > Show Symbol > Show End of Line (or Show All Characters).

Line break characters themselves have no visual properties, so what appears on screen will depend on how the code editor *represents* them—and also on what's actually represented. In BBEdit, for example, you'll see a *not sign* (¬) to stand for the character that's *really* used: *line feed* (LF), the standard on OS X and Unix. (Earlier versions of BBEdit instead used the *carriage return*, the standard line break character in Mac OS versions prior to OS X.)

On Windows, the standard line break is *carriage return plus line feed* (CR+LF, or just CRLF). That's what you'll see, for example, if you export HTML from Word for Windows and open it in Notepad++. There it will show up as two small icons with the CR and LF initials. As you might figure, this line break is actually a *pair* of

characters—but since they're normally handled together, we'll keep speaking of them as a single one.

In fact, I may get even lazier, sometimes speaking of *line breaks* when I mean *line break characters*. To keep things straight, though, I'll always say line break *tag* when that's what I mean instead.

•

A code editor may have a command that removes unneeded line break characters automatically. In BBEdit, for example, the Remove Line Breaks command replaces all such characters with spaces—except where two line breaks together produce a blank line. This exception preserves separation between paragraphs or other code blocks.

But even with such built-in exceptions, broad commands of this type may also remove line breaks needed for code readability, especially in the HTML document head. They can even introduce errors—such as replacing a line break character with a space even if it comes *after* a line break tag (
). That replacement will make the new line start with a space! (The error wouldn't be entirely the code editor's fault, though. Logically, the line break character should go *before* that tag, not after.)

You can fix some such problems afterward with find-and-replace—or avoid many in the first place by restricting the operation to a selected section or sections. But you can do even better by substituting your own set of find-and-replace operations.

Here's the procedure:

1. Figure out how you need to represent a line break in your code editor's find-and-replace dialog. For BBEdit, the search code is always "\n", for a line feed. For Notepad++, it's normally "\r\n", for a carriage return plus line feed—and to use that code, you need to turn on Extended search mode!

2. Choose a placeholder made of characters in a pattern that would otherwise never appear—"zxy", "@#$", or such.

3. For double line breaks—used to create a blank line—replace *each* break with your placeholder. With BBEdit, it might look like this:

Find: \n\n

Replace: zxyzxy

For Notepad++ (in Extended search mode, as I noted):

Find: \r\n\r\n

Replace: zxyzxy

4. Make the same replacement for any specific kinds of *single* line breaks you'd like to later restore in code. These would be line breaks that fall before or after certain symbols or tags. (But not after commas, colons, or semicolons!)

For instance, in HTML from Word, you will likely want to restore line breaks after the line break tag (
). The replacement in BBEdit would look like . . .

No, wait. While we're at it, let's shift them into reverse positions, as they should be.

Find:
\n

Replace: zxy

Or for Notepad++:

Find:
\r\n

Replace: zxy

Then of course, we need to look out for line breaks in this *new* position as well, in case the file gets reprocessed. For BBEdit:

Find: \n

Replace: zxy

And for Notepad++:

Find: \r\n

Replace: zxy

Other line breaks you might want to restore later in Word's HTML are:

Before tab character ("\t" in BBEdit or Notepad++)

Before @

After <!--

After */

After }

Between > and <

5. OK, here's the big one: Replace each remaining line break with a space.

6. In case the previous step left any *double* spaces, replace each pair with a *single* space.

At this point, you should have *no* line break characters left in the file. With word wrap on, all you should see is a solid block of text. (If you see only a single line, then word wrap is off!)

7. Finally, replace each of your placeholders with a single line break. For our BBEdit example:

Find: zxy

Replace: \n

For Notepad++:

Find: zxy

Replace: \r\n

•

Actually, in the instructions just given, I cheated a little by putting off discussing one of the trickier aspects of removing line breaks in code. This has to do with the two different styles of writing CSS code, which I described earlier: "loose," with spaces after punctuation, as in the first example below; and "tight," without spaces, as in the second.

```
style="margin-top: 0; font-family: Caecilia;"

style="margin-top:0;font-family:Caecilia;"
```

Again, each style is proper CSS, but the issue here is consistency for the sake of search. In Word's HTML, for example, CSS is written tightly—but even *without* the spaces found in loose CSS, Word may insert a line break character after a CSS colon or semicolon. The result might look something like the following, with the pilcrow (paragraph mark) standing in for your code editor's representation of a line break character.

```
style="margin-top:0;¶
font-family:Caecilia;"
```

If you replace *that* line break with a space, you're placing it where a tight style wouldn't have one. That creates a variation—a combination of tight and loose styles—that a search could miss.

```
style="margin-top:0; font-family:Caecilia;"
```

So, if you find yourself dealing with tight CSS like Word's, any line breaks after colons or semicolons in that code will have to be handled separately. You'll have to remove the breaks *without* adding spaces—and you'll have to do it *before* replacing other line breaks.

Of course, if you never use colons or semicolons in your text, you can handle the ones in CSS with a couple of simple find-and-replace operations. But if you *do* use

them in text, the line breaks following those marks *would* need to be replaced by spaces.

To treat the punctuation marks one way in CSS and another in text, we'll have to turn to grep. The following find-and-replace operation will remove all line breaks after colons and semicolons in tight CSS without adding spaces. At the same time, it will ignore those same punctuation marks in your text. (This assumes your code editor lets you search for line breaks with grep!)

First, how it would look in BBEdit:

Find: ([=;])\n?(["']?[a-z\-]+:)\n?([^/])

Replace: \1\2\3

And now the Find for Notepad++, substituting the default Windows line break characters. The Replace is the same.

Find: ([=;])\r?\n?(["']?[a-z\-]+:)\r?\n?([^/])

In case it's not clear, the two characters in the second pair of square brackets in both Finds are a single quote (') and a double quote ("), both straight (not "curly"), with nothing between the two.

This operation looks for CSS property names—which consist of single words or hyphenated phrases in lower case—in the midst of punctuation patterns found only in CSS. Then it removes any line breaks in the area. But don't apply it to *loose* CSS with line breaks, or you could wind up with the same problem of inconsistency in reverse!

•

Beyond cleaning up line breaks, you should standardize spaces in relation to closing tags. This too will help in later operations.

First search for spaces placed *before* closing tags for font styling—closing tags like for italics (</i>), bold (), underline (</u>), and span (). Then move the tags in *front* of the spaces. This makes sure the tags always abut characters they're actually modifying, minus the spaces that follow.

After that—not before!—search for spaces placed before closing tags for text elements—closing tags like for paragraph (</p>), list item (), div ( ), and heading (</h1>, </h2>, and so on). Those spaces should just be removed.

•

Though Word, when handled properly, doesn't interject nearly as much extraneous code as many claim, it does add more than necessary to the beginning of its HTML. While not enough to really be a problem, it can be irritating when you're examining your file and trying to scroll to the beginning of your text.

The biggest culprit is a collection of font data that serves no purpose in your Kindle book but that in some files can be extensive. You'll find it between these two comment lines:

```
/* Font Definitions */
/* Style Definitions */
```

You can delete it all manually, but you can also do it automatically with grep. Use the following Find to select it all, then replace with nothing—in other words, replace with the Replace box empty. That's a regular space before and after the phrase "Font Definitions."

Find: /* Font Definitions */[^/]*

Some earlier Word versions—such as Word 2004 for Mac—have another annoying and generally useless habit: After a closing tag for text formatting—for instance, for italics or a change in typeface—they insert a pair of span tags specifying a return to "normal."

Though these tags may never interfere with your operations, you might prefer to clear them out. If you see any, you can use this grep on them:

Find: <span style='[^>]+:normal'>([^<]*)

Replace: \1

But I did say *generally* useless, because there's one case in which you'll need to leave these spans in place. Within an extended passage in italics, it's common to denote emphasis on some words by *removing* italics. In Word's HTML, this is rendered as a "normal" span. If you take it out, you'll lose the emphasis.

Earlier Word versions might also have inserted large amounts of numbered "OLE_LINK" bookmarks in your source document. These markers were meant to support functions that were seldom used and are entirely irrelevant to Kindle books. Besides causing clutter in

your HTML, they are often exported incorrectly, creating errors—so you should definitely get rid of them. Just delete them all in Word's Insert Bookmarks dialog.

Note that the document itself doesn't have to have come from an older Word version to contain these links. Any document can be infected from part of it having been copied and pasted from an older one.

•

Though these cleanup methods should make your HTML ready for most other find-and-replace operations, I can't guarantee complete safety. You'll have to keep an eye on your own code and your own operations to spot special cases. Be prepared to tinker to get everything perfect.

HTML Testing

While working out fixes for your HTML, you'll probably want to test your results more often than the typical Kindle author would. So, forget the online previewer at Amazon KDP and download Amazon's Kindle Previewer for the desktop. You can find it here:

www.amazon.com/kindleformat/kindlepreviewer

Not only is the desktop previewer faster than the online one, but it can also convert your HTML to Kindle format right there on your computer, by way of the Kindle converter it also installs. All you need do is open or drag in your HTML file, and the Previewer will handle the rest.

The conversion provided by the Kindle Previewer is not quite as good as the one you get at Amazon KDP. Specifically, it will be missing your cover, as well as Go To menu items for the book's "beginning" and table of contents. Still, joint converting and previewing in the Previewer lets you run quick and mostly accurate tests of formatting without the need to visit Amazon KDP and wait for processing.

Another advantage of converting in the Previewer is that you get access to a log of "Compilation Details," including warnings and error messages. These can be valuable in pinpointing errors in your file or in just studying how the converter operates. Ideally, the *only* actual warning you want to see there is "Cover not

specified." (And that's when you plan to upload your cover to Amazon KDP separately.)

Of course, for best testing, you'll want to take the file that Previewer has converted and view it on actual Kindles. You can start with the free desktop Kindles for Windows or Mac, available here:

www.amazon.com/kindleapps

Beyond that, hardware Kindles are now cheap enough that you may find it worthwhile to collect a few just for testing. For my own tests while first writing this book, I had one Kindle from each major family—a latest-generation basic Kindle, a Kindle Paperwhite, and a 7-inch Kindle Fire HD—plus an iPad with the Kindle app.

Files from the Kindle Previewer can quickly be sent to any or all of these Kindles with Amazon's Send to Kindle app. You can get yours here:

www.amazon.com/sendtokindle

Note that preview files sent this way will be treated by Kindles as Documents, not Books.

•

One challenge in polishing your HTML is that it's *not* your HTML that will wind up in the Kindle book—it's Amazon's *translation* of your HTML! And as Amazon continues to develop its Kindle tools, it changes the way it makes that translation.

If you're comfortable enough with HTML, there's one tool that's particularly helpful in facing this challenge: KindleUnpack. It takes apart a preview book package and lets you look at its component files. You can download it from the first post in this forum thread:

www.mobileread.com/forums/showthread.php?t=61986

KindleUnpack requires a version of the Python software language to be installed on your computer. Macs already have it. For Windows, you can download free versions from either of these locations, among others:

www.python.org/download

www.activestate.com/activepython/downloads

The latest version of either Python 2 or Python 3 should work with KindleUnpack.

What you'll see after unpacking includes the source files you fed the converter plus files and folders for two different versions of your Kindle book—the Kindle's old Mobipocket format (here called "mobi7") and the newer Kindle Format 8 (here called "mobi8"). Each Kindle version will have its own HTML file or files for the book contents, as well as its own set of image files.

You can now see exactly how the Kindle converter handles the HTML you feed it. And you can go back and see how changes in your HTML generate different results in Amazon's.

•

Even if you get everything right in your preview, that doesn't mean it will be right in the finished book. Amazon KDP staff, for example, have a bad habit of moving a book's navigation anchors before releasing it.

What's more, Amazon reserves the right to reprocess your files anytime after publication without letting you know—and they actually do that. This is so your books can take advantage of improvements to the Kindle platform, but it ignores the possibility of unexpected results.

Sadly, there's no HTML to stop Amazon from messing with your books!

•

Previewing for Kindle is a large topic in itself. To read much more about it, see my article at

www.newselfpublishing.com/ProofingKindle.html

HTML and Apps

Though most people still use word processors like Word to compose their books for Kindle, there are now a number of dedicated ebook apps and other apps that feature ebook authoring. These include Scrivener, Jutoh, Vellum, and Amazon's own Kindle Comic Creator and Kindle Kids' Book Creator. Instead of giving you HTML files, these apps provide files in Kindle book format.

But that doesn't mean the apps actually don't produce HTML files. What it does mean is that they send those working files directly to the Kindle converter that must also be installed on your computer. So, there are HTML files, you just normally don't see them.

It's possible, though, to tinker with the HTML from these apps just as you would with HTML from a word processor. Some apps, like Jutoh, allow you to save their working files for inspection and editing. But if your app doesn't, you can still access them.

First you need to break open the final Kindle book file with KindleUnpack, the utility I discussed in my section on HTML testing. You will then see a file named kindlegensrc.zip (for "Kindlegen Source"). After unzipping this file—usually with a double click—you'll find the files your app sent to the converter, and you can edit them like any others.

After editing the working files, you can rezip these files alone and submit them to Amazon KDP. Amazon doesn't need anything else from the Kindle book you generated.

2
HTML Fixes

Fixes for Fonts

In *From Word to Kindle*, I discussed Georgia as the best font to specify in the source document for a Kindle book. But specifying any primary font at all can cause problems. For example, it can prevent the Kindle user from changing the font as desired. On some Kindles, it can even cause a switch to a sans-serif font. So, the better course is to *not specify at all*.

Unfortunately, there's no way *not* to in a word processor or page layout app—so to avoid specifying, you have to *remove* some code from the exported HTML. But that's easy to do. For example, assuming your font is Georgia, the find-and-replace in Word's HTML could look like the following. (The first Find example is Windows, the second is Mac.)

> **Find:** font-family:"Georgia","serif" *or*
>
> **Find:** font-family:Georgia
>
> **Replace:** font-family:

In other words, you can just get rid of the font name or names, along with any double quotes immediately surrounding them and any comma between. The empty-value declaration that remains will be the same as *no* declaration. This is simpler than trying to remove the declaration entirely, since the surrounding punctuation can vary.

•

With Word, there's one font you might have to give special attention, even if you haven't specified it in your document: Times New Roman. That's because it's Word's underlying default font, and Word may specify it in HTML wherever you haven't asked for a different one. This might in some cases happen, for instance, if you don't specify a font for the Hyperlink and FollowedHyperlink character styles.

For this too, the best plan is to remove the specification, just as you would for your primary font. Here are the Finds, for Windows and Mac respectively.

Find: font-family:"Times New Roman","serif" *or*

Find: font-family:"Times New Roman"

•

Though it's best not to specify your primary font, you may well want to specify a secondary one for contrast or special use. As I said in *From Word to Kindle*, the best choice for this is generally the sans-serif font Helvetica, both for its characteristics and for its wide availability on Kindles. Arial will work in most of the same cases, since it's nearly identical and since many Kindles can swap these two fonts as needed. To use one or the other, you could simply leave the specification for it in Word's HTML export.

But to increase your chances of getting a font you want, you could add alternate choices to your HTML. So, instead of leaving a specification of Helvetica or Arial, you would replace it by listing Helvetica *and* Arial *and* the font family "sans-serif," in that descending order of preference. In the HTML of Word for Mac, for example, it would look like this:

Find: font-family:Helvetica

Replace: font-family:Helvetica,Arial,sans-serif

Word for Windows, on the other hand, already includes "sans-serif" as an alternate, so the find-and-replace would look like this.

Find: font-family:"Helvetica","sans-serif"

Replace: font-family:"Helvetica","Arial","sans-serif"

In the same way, you can increase the odds of getting a monospace font by listing together Courier, Courier New, and "monospace." Or a narrower font by listing Times, Times New Roman, and "serif."

Note that font names like Courier New and Times New Roman—multi-word names with spaces between—*must* be enclosed by quotes, regardless of whether you're enclosing other font names. And those quotes must be single, if the declaration is part of a string surrounded by double quotes.

•

Word processors and page layout apps normally specify sizes and distances in *absolute* units like inches or centimeters or *points*—a point being 1/72 inch. And that's what gets exported to HTML. When expressed this way, older Kindles need you to stick to the common font sizes of 10, 12, 14, 18, and 24, with 12 as normal body type. Any in-between size gets rounded.

But the point size of Kindle text characters can vary according to the device and user settings. So, nowadays, it's much better to give the Kindle *relative* measurements.

In typography, relative measurements are often expressed in terms of the *em.* For print, an em equals the point size of the font in use. So, if the font size is 12-point, an em equals 12 points, or one-sixth inch. If the font size is 18-point, the em is 18 points, or a quarter inch. (The em got its name from the width of a capital letter *M,* which traditionally had a width equaling the point size.)

In ebook typography, though, the em is defined a bit differently. It is instead the size of normal body type, as set by either the device user or the book itself. So, if normal body type is currently set to 12-point, then 18-point type is 1.5 ems.

With this modified definition, all differences in font size can be specified in ems instead of points. And that's what the Kindle favors.

Font size specified in points will still work fine on many Kindles, which will simply scale them as needed to appear normal. But some newer Kindles will stick to the actual point sizes while equating points to *pixels.* As a result, the type will be *tiny* until the Kindle user adjusts it. Other Kindles—at least in their buggy emulations in Amazon's previewers—may show the same type as oversize!

For best handling, then, convert the point-based font sizes in your CSS to ems. Here's what the conversion would look like for 14-point type in Word's HTML. (Note the decimal point and following zero that Word includes in the point size.)

Find: font-size:14.0pt

Replace: font-size:1.2em

The conversions for all of the Kindle's favored point sizes would be:

10 points ~ .8 em

12 points ~ 1 em

14 points ~ 1.2 em

18 points ~ 1.5 em

24 points ~ 2 em

Another way to express relative measurements for the Kindle is percentages. For font size, this would again

be based on the size of your normal body text. For 14-point type in Word's HTML:

Find: font-size:14.0pt

Replace: font-size:120%

And all the conversions together:

10 points ~ 80%

12 points ~ 100%

14 points ~ 120%

18 points ~ 150%

24 points ~ 200%

One tricky thing about relative measurements is that they're *cumulative.* If you apply two different measurements to the same text, the final measurement is the two of them multiplied against each other.

For example, say that you set a paragraph's text to 18-point by way of a relative measurement of 1.5 em or 150%. And then say you want to set a single *word* within that paragraph to 24-point. If you try specifying that as 2 em or 200%, as you normally would, you wind up with a final measurement of 3 em (1.5 × 2) or 300% (150% × 200%), for a point size of 36—half again as large as you want! To avoid surprises, then, limit yourself to one font size per paragraph.

Note that if you change *any* font sizes in your Kindle book to relative measurements, you must change them

all. Otherwise they'll be grossly out of proportion to each other on some Kindles.

•

An odd complaint you sometimes read in Kindle book reviews is that some or all of a book's text is missing, leaving blank pages. You might figure that the customer's file was corrupt on download, but that's not the case. What has happened is that text color in the book was specified as black, while the customer had chosen to view text on a black background. Black text on black simply vanishes.

In HTML, black is the default color for text—so unless a different color is needed, it's usually not specified at all. Word, for instance, will not specify text color in its exported HTML as long as font color in the source document is set to the default "Automatic." This lack of specification allows the Kindle to match text color to viewing mode—for example, white text on a black background.

But sometimes font color in a source document is accidentally or purposely set specifically to black; or text that is set that way is pasted in from another source; or an app might *always* specify text color in its exported HTML. You might not notice this, or you might just not see any reason to change it. On your computer screen or in print, the text looks exactly the same. And when you're testing on your Kindle, you might never think to try a black background. (Tip: You should.)

It's a good idea, then, to add a step to your processing that checks for the black specification and removes it. First you have to see how your exporting app *would* specify it, then simply remove the color designation. Here's how it would look in Word.

Find: color:black

Replace: color:

Leaving the specification empty is the same as not specifying.

If you're using a different app, find the relevant code by first setting font color to black in your source document, then searching for "black" in your exported HTML.

Fixes for Paragraphs

A word processor or page layout app may omit a setting from its exported HTML when the value is zero or another default. That's perfectly acceptable in HTML. The problem is that the Kindle may then assume a different value entirely.

If, for example, you set a zero first-line paragraph indent in Word, the exported HTML won't include that instruction, because it's already the HTML default. So, the Kindle won't get the message and will then use its *own* default—which for older Kindles is to indent!

In *From Word to Kindle*, the workaround I offer for this is to set a first-line indent in Word of .01 inch or centimeter. This forces Word to write the instruction to HTML, while the indent is small enough not to be noticed. Though that solution works fine, a quick find-and-replace can change the HTML to what it really *should* be. For the measurement in inches:

Find: text-indent:.7pt;

Replace: text-indent:0;

Word uses a similar trick of its own to eliminate space after a paragraph, setting it to .0001 point. You can easily clean that up, too:

Find: margin-bottom:.0001pt;

Replace: margin-bottom:0;

•

A similar problem may arise when you want left alignment for your text. Here again, the instruction may be left out of exported HTML, because it is already the default. But without the instruction, *any* Kindle will *justify*—add space within lines to produce a smooth right margin.

When Kindles justify without hyphenating, *this will make any text look bad*. And even when Kindles *do* hyphenate, as the newest can do, the text won't look great. That's because they use a primitive form of justification that can only expand spaces between words, never contract.

From Word to Kindle provides a trick that blocks justification, but blocking is much easier to handle in HTML. In your source document, make sure you justify any text you want left-aligned—in other words, make it the *opposite* of how you want it to wind up. That guarantees that an instruction will be written to HTML. Then change the instruction in code with a find-and-replace, such as this one for HTML exported from Word:

Find: text-align:justify

Replace: text-align:left

Having provided this fix, I must also warn you that Amazon does not like forced left alignment in Kindle books—at least, not for basic text. In fact, Amazon is very attached to the amateurish look of its justification.

So, in the event that a customer complains or that KDP staff have occasion to examine your book, you may well receive a "Quality Notice" asking you to degrade your Kindle book to Amazon's standards. And they will check back with you periodically until the issue is "resolved." So far, though, I have been able to withstand these depredations simply by explaining that I prefer to leave my book as is.

But even if you decide—willingly or unwillingly—to justify your text, there are still elements you *won't* want to justify, like headings and lines of verse. Though these aren't justified unless broken into two or more lines, almost *any* line might be broken when read with a large enough font or on a phone. So, it's best to block justification for at least certain paragraph classes.

How to do this selectively? The simplest way is usually to turn off justification for the whole document as described above, and then restore it for specific classes. You can do the latter by adding a declaration to the *end* of the style section in the document header. Coming after the other class-defining declarations, it will take precedence over them.

Let's see how this would look in Word. Word writes the ending of the header style section with a closing comment marker and a closing style tag, like this:

```
-->
</style>
```

Comment markers within style tags are ignored in modern HTML, so you could add your new style declaration either before or after it. But to design a macro that will be safe to repeat, search for the closing comment marker and the closing style tag *together*, then insert your declaration *between*. Here's how it would look if you were turning on justification for the two paragraph classes Basic and BasicIndented.

```
-->
p.Basic, p.BasicIndented {text-align: justify;}
</style>
```

Of course, you could instead decide to do the opposite: Leave on justification for the whole document but turn it off for selected classes. Choose whichever is simplest for you.

•

As I said, most Kindles will happily scale fonts for the user when the sizes are based on points. So, you'd expect the same treatment for format spaces like indents, space before paragraphs, and space after, when based on points or inches or centimeters. No such luck. When a Kindle user scales the font, *none* of those spaces are adjusted. What's more, on a high-resolution Kindle, the spaces appear about half size!

I've already suggested basing your font sizes on the em as a relative measurement. The em is even *more*

useful in defining format spaces, as it allows them to scale with the font size.

Keep in mind, though, that in ebook typography, the em is based on the size of normal body text. So, an em of indent or of space before or after would be the same size no matter where it appears. For example, if your normal body text is 12-point, an em of space will measure 12 points whether added to a normal paragraph or to a heading. (That is, unless you apply one measurement over another, making them cumulative.)

A typographic rule of thumb is that a first-line paragraph indent should be 1 em. So, say you've specified the indent in your source document as one-quarter inch. The replacement in Word's HTML would be:

Find: text-indent:.25in

Replace: text-indent:1em

You could likewise substitute em measurements for absolute ones for the following CSS properties (which should be all lower-case in your HTML). The rule of thumb would be 1 em for every 12 points.

• Margin-left, for a whole-paragraph left indent (as for a block quote).

• Margin-right, for a whole-paragraph right indent (which is honored only on newer Kindles).

• Margin-top, for space before the paragraph (which, actually, you should avoid when possible, because some Kindles may ignore it).

• Margin-bottom, for space after.

Using ems has one important limitation, though: For format spaces—unlike for fonts—*older Kindles round all em measurements to the nearest whole em*. For consistency among Kindles, then, you can specify spacing of 1 em or 2 ems, but not 1.5. That makes it rather a blunt instrument.

•

While there are some format spaces you want automatically scaled with the font, there are others that should instead be relative to the whole screen display. For this you would use percentages. Unlike when used for fonts, percentages for format spaces relate to the surrounding unit—in most cases, the display itself.

For example, you might want a paragraph positioned halfway across the display, as for the closing of a letter. (Don't expect older Kindles to display this example correctly.)

Love,
Lynne

The replacement in Word's HTML might be

Find: margin-left:2.25in

Replace: margin-left:50%

A percentage of this type for left margin won't work properly on older Kindles, but only because they don't properly handle large left margins in *any* form. So, you don't lose anything by trying.

As another example, you might want to start each chapter's text about a third of the way down the display from the chapter heading. In that case, in your source document, you could set the space after the heading to a rough number of points, then replace that number in code. For Word's HTML, it might look like so:

Find: margin-bottom:120.0pt

Replace: margin-bottom:33%

•

For your Kindle book, it's best *not* to specify line spacing—the space between lines in a paragraph. *Not* specifying will allow readers of at least some Kindles to adjust the spacing. Also—whether by intention or in error—it can give you a *much* better page count for payments in Amazon's Kindle Unlimited program.

To omit the setting from Word's exported HTML, you could simply choose Single spacing in your source document. With that setting, though, the line spacing displayed in Word will be much tighter than the Kindle's default.

So, to emulate the Kindle while editing, you might like to instead set your line spacing to "At Least" or "Exactly" a certain number of points, then *remove* the

setting from Word's exported HTML. That requires finding two different phrases and replacing with nothing. For line spacing of 14 points:

Find: line-height:14.0pt;

Find: mso-line-height-alt:14.0pt;

If you really do need to set line spacing for the Kindle book itself, it's best that this too be set with a relative measurement. Word actually allows you to do this in the source document with a "Multiple" setting for line spacing. But you could instead set absolute spacing and replace it later in HTML with a percentage relative to font size. It might look like this:

Find: line-height:14.0pt;
Replace: line-height:120%;

•

As time has gone on, Amazon has grown more aggressive in imposing its idea of suitable paragraph formatting. For example, the practice of setting off paragraphs both with a first-line indent *and* with space between is redundant—a typographic abomination we've inherited from the defaults of Word. Yet most Kindles seek to enforce it, even when our HTML specifies differently.

To thwart this, you must resort to guerilla action and confuse your opponent. Basically, you have to remove anything the Kindle might use to identify potential victim paragraphs.

If you're working in Word, that means avoiding Normal style—Word's default. Of course, you can simply apply a style of a different name in your Word document. But if that disrupts your workflow, you can make the change later in code. In Word's exported HTML, just replace all instances of "MsoNormal" with any other class name, with the provisos that it contain no space and does not include "Normal" within it. As an example, I substitute the name "Basic".

Find: MsoNormal

Replace: Basic

If your HTML contains class names like MsoNormalCxSpMiddle, this probably means you did *not* turn off Word's Editing option "Keep track of formatting" as I recommended earlier. But if you keep these, be sure to replace "MsoNormal" as *part* of the names, as well as where it stands alone.

Aside from a possible class name change, you may want to cloak your paragraphs so they don't appear as paragraphs at all! That's done by replacing all paragraph tags in your HTML with division tags. (HTML handles "divs" exactly like paragraphs, except that less default formatting is applied.)

Here are the two replacements you'll need to make:

Find: <p
Replace: <div

Find: </p>
Replace:

At least, that's all you have to do in HTML exported from Word, because Word's code has a quirk that this time works in your favor. When Word defines a class in CSS, it specifies that the class can be associated with either paragraphs or divs. So, after changing a paragraph to a div in Word's HTML, you'll find that its assigned class still works for it.

Another app, though, might be less generous, requiring you to adjust your CSS *selectors* as well. If a class is only specified to work for paragraphs, you can make it work for divs *instead* with a replacement like this:

Find: p.stylename
Replace: div.stylename

Fixes for Headings

Just as the Kindle will try to hijack the formatting of your normal paragraphs, it may do the same to headings. For one thing, some Kindles will make them bold, even when you say not to!

This too can be worked around in your source document, by defining and applying alternate styles in preference to the standard styles for headings. But it's often more convenient to handle the change in HTML.

This is a bit trickier than with paragraphs, though, because of the difference in how headings are exported. Instead of winding up with a paragraph tag (`<p>`) associated with a class, headings are set off by heading tags (`<h1>`, `<h2>`, `<h3>`, and so on) with formatting of their own. So, they're harder to disguise to the Kindle.

For Word, you would start in your source document by defining alternative heading styles *as though* you'd apply them there—even though you won't. For example, for a top-level heading in Word, you could define an alternative style named Level 1, specifying that it is *based on* the Heading 1 style. With no change from the base style definition, the two would produce identical formatting. This alternative would then be included in Word's exported HTML among the class definitions for paragraphs.

You would then replace the standard heading tags in your HTML with paragraph tags associated with that class. It would look as follows. (Note that the space in

the original style name has been removed in the translation to class name.)

Find: <h1

Replace: <p class="Level1"

Find: </h1>

Replace: </p>

By the way, I tried “Head 1” as a style name in Word, and the Kindle was smart enough to figure out I was trying to fool it!

Fixes for Line Breaking

Word processors and page layout apps both feature special formatting characters that let you control line breaking and hyphenation. Though these characters are allowed in HTML text as well, they are not always recognized or honored by individual Web browsers and ebook readers.

It's important to know which of these characters you can use on the Kindle, which of them you can't, and which can be replaced with equivalent HTML code. It's also important to understand how they can be used to meet the unique challenges of flowing text. You can then prevent much of the awkward line breaking that is a bane of Kindle text display and that most Kindle publishers don't even know can be avoided.

•

One formatting character that is both common and respected by Kindle is the *nonbreaking space*—also called *no-break space* or *hard space*.

This character appears as a regular space but also holds together the two adjacent words or numbers, keeping them always on the same line when you don't want them split apart. It's also used to hold together a typographically correct ellipsis (. . .), which consists of three periods with spaces between, plus one in front. And it's commonly used in HTML to create a string of spaces, since browsers and ereaders won't collapse them

all into one, as they do with normal spaces—but Amazon now asks you *not* to string them together like that.

Though this character can almost always be added in your source document, you may still need to edit it in HTML—and sometimes it's more practical to add it there in the first place. For either task, you'll need to find out how the character is or should be represented. In Word's exported HTML, for example, the nonbreaking space appears as an entity name, while another app might export it as an entity number.

  or

Or you might see instead the native Unicode character as displayed by your code editor. BBEdit, for example, shows it as a gray bullet (•). You can just copy and paste this character into your Find or Replace, or insert it with Option-Space, a standard keyboard shortcut on the Mac.

It really doesn't matter which of these forms is used in your HTML, as long as it's consistent for the sake of your operations. In fact, you can switch forms as convenient. My own HTML processing begins with the nonbreaking space as an entity name, as exported by Word, since that seems most convenient for editing. Near the end, though, I convert them all to the Unicode character, making my HTML text more readable for troubleshooting. But in case I happen to reprocess,

there's a safety step near the *beginning* to convert them all to the entity name again.

•

In Kindle books, the use of nonbreaking spaces can go beyond the common job of protecting phrases and on to avoiding other kinds of awkward line breaking. In this way, they can help overcome one of the chief faults of flowing text in ebooks—the randomness of line breaks.

One example of awkward line breaking is a short, one-syllable word appearing on a line by itself at the end of a paragraph—sometimes called an *orphan*. This is something a book designer definitely wants to avoid, while Kindle text is full of them. In fact, it is one of the Kindle's main uglifying features.

The solution is to add a nonbreaking space in front of any such word that *might* end up in that position. That would pull down the last word of the line above if needed to lengthen the final line. In *From Word to Kindle*, I suggested adding nonbreaking spaces in your source text, and that's really the most reliable method. But with grep, you can do almost as well in your HTML.

The following operation will place a nonbreaking space in front of almost any word or number made up of four characters or fewer at the end of a paragraph. You can add, remove, or substitute punctuation marks as needed. The apparently empty set of square brackets in front has a regular space inside, and there's another toward the end, after the caret (^). Note that you must first have moved or removed spaces in front of closing

tags, as prescribed in my section on cleanup. As with any grep, test carefully when first using!

Find: [ ]([a-zA-Z0-9][a-zA-Z0-9]?[a-zA-Z0-9]?[a-zA-Z0-9]?[\.\?!'":\)]*<[^ &]*/p>)

Replace: \ \1

You can change the maximum number of characters by adding or removing instances of "[a-zA-Z0-9]?". More characters will give you fewer orphans but more unevenness in the preceding line.

The operation above can handle a closing formatting tag but not an opening one. To deal with an opening tag for italics, bold, or underlining, add the following Find, using the same Replace as above.

Find: [ ](<[ibu]>[a-zA-Z0-9][a-zA-Z0-9]?[a-zA-Z0-9]?[a-zA-Z0-9]?[\.\?!'":\)]*</[ibu]>[\.\?!'":\)]*<[^ &]*/p>)

Though both of these orphan operations allow punctuation before the final tag, they don't require it, so their use is not limited to regular paragraphs. With the "p" at the end, you can make them work on anything you care to fit inside <p> tags—including headings, if you've chosen to disguise them.

Other substitutes for "p" will make them work on additional defined elements. Substituting "li" will apply orphan control to list items. Substituting "h1" will

handle top-level (undisguised) headings. To handle *all* levels of heading, substitute "h[1-6]".

One drawback of the operations above is that they take no account of the length of the preceding word. If the last word is four characters and the preceding word is eight, this creates quite a gap at the end of the shortened line. This becomes more important if your text is justified.

As an alternative, you could use something like this, which avoids an orphan of exactly one character as long as the preceding word is no longer than nine.

Find: ([ ]["'\(]*[a-zA-Z0-9]?[a-zA-Z0-9]?[a-zA-Z0-9]?[a-zA-Z0-9]?[a-zA-Z0-9]?[a-zA-Z0-9]?[a-zA-Z0-9]?[a-zA-Z0-9]?[a-zA-Z0-9])[ ]([a-zA-Z0-9][\.\?!'":\)]*<[^ &]*/p>)

Replace: \1\ \2

With modifications, you would set up a series of such operations, each one dealing with a different character length combination. For example, starting with the operation above, a workable series might be 1/9, 2/7, 3/5, 4/3, and 5/1. You could also make the operations handle more cases with some judicious additions—for instance, inserting an optional apostrophe before the last character of words with three or more; or adding a hyphen and dashes to the first pair of brackets.

•

Nonbreaking hyphens are special characters often used to block automatic line breaking between hyphenated words or in the middle of other hyphenated terms. Even though it's normally fine to include them in HTML text, they don't work on all Kindles, and on some can even trigger a font change for the entire book! So, you need to keep them off the Kindle.

But removing them from your source document may be less than convenient if the document will be used also for another ebook format or a print version. The solution is to leave nonbreaking hyphens in your source but later replace them with regular hyphens in HTML.

As with nonbreaking spaces, though, exactly how you find them will depend on how they appear in your HTML document. Word for Windows, like many apps, will export it as an entity number. (There's no entity *name* for this character.)

```
&#8209;
```

Word for the Mac exports the Unicode character itself to its HTML, and you can copy and paste it from there. (You *cannot* copy it from a *source* document in Word, because the character used there is only a substitute.) You can also get the Unicode character from the Mac's Character Viewer by searching for "non-breaking hyphen" (with the first word hyphenated). However you acquire it, the character will *look* like a

regular hyphen, so you'll have to keep track of the difference.

Losing use of the nonbreaking hyphen is a typographic nuisance, to say the least. But you can more or less make up for it—at least on newer Kindles. For example, for this book and my others on Kindle publishing, I want to avoid automatic line breaking in the middle of "UTF-8." This can be done with a find-and-replace like the following, which shows the HTML entity in the Find (between "UTF" and "8"), and a regular hyphen in the Replace.

Find: UTF‑8

Replace: UTF-8

The enclosing span tags, with their white-space attribute and nowrap value, take the place of the nonbreaking hyphen, with the same job of blocking automatic line breaking. I like to call the tags and their enclosure a *nonbreaking span.* Older Kindles, which do *not* honor this, will safely ignore it.

This kind of simple replacement works fine to protect a limited number of terms or phrases. But to protect a variety of them, you might want to turn to grep. The following is how I would write a grep find-and-replace to make *all* needed substitutions for nonbreaking hyphens in HTML from Word for Windows. For the default encoding of Word for the *Mac*,

in place of this Find's HTML entity (‑), you would place the actual Unicode character.

Find: ([a-zA-Z0-9]*)‑([a-zA-Z0-9]+)

Replace: \1-\2

This operation works even when a phrase has more than one nonbreaking hyphen—as, for example, in a hyphenated ISBN. In that case, though, it will generate excess code that does no harm but can be trimmed if you like. Just find the following and replace with nothing.

Find:

Finally, you should replace any *remaining* nonbreaking hyphens with regular ones—just in case the grep search missed any.

The examples above assume you've inserted nonbreaking hyphens in your source document as needed and only want to replace them in HTML. That's certainly the most reliable approach. But if they were never in your source document to begin with, you can write limited find-and-replace operations for your HTML like the one above for "UTF-8"—only this time with a regular hyphen in the Find instead of a nonbreaking one.

•

Besides the nonbreaking hyphen, the Kindle has trouble with its cousin, the *soft hyphen*—or Optional Hyphen, as it's called in Word. This character tells the reading app where a word can or should be hyphenated when it falls at the end of a line. You would commonly insert one in a word processor or page layout app when automatic hyphenation is dividing a word incorrectly, awkwardly, or not at all. Soft hyphens might also be inserted by an OCR app as it converts scanned pages to editable text, replacing all normal hyphens it finds at the ends of lines.

Unless your word processor is specially set to display these characters, a soft hyphen should normally be hidden, to appear *only* when the word is broken at the end of a line—in which case it displays as a regular hyphen. And that's how *some* Kindles handle it. But others show it as a regular hyphen even when the word is in the *middle* of the line. Because of this incorrect behavior, any soft hyphens should be removed from your Kindle book.

This is simple enough to do in your source document—in Word, for example, you just choose "Optional Hyphen" from the Special menu in Find and Replace, then replace with nothing. But you can also do it in HTML—and that's a good idea even just as a safeguard, in case any soft hyphens have slipped in without your knowledge.

Here again, you'll have to determine how to find the character. In HTML from Word, both for Windows and Mac, the soft hyphen is inserted as the Unicode character itself. So, copy one from your exported HTML—*not* from your source document in Word, which uses a substitute—and paste it into your Find box. Or get one by searching for "soft hyphen" in the Mac's Character Viewer, or in the Windows Character Map with "Advanced view" selected. (In a BBEdit Text Factory, if you paste the character into Find text, the app substitutes a special code that works just as well.)

Another app might export the soft hyphen as an HTML entity instead.

 or

•

Another special formatting character that *does* work on the Kindle—sort of—is the zero width non-joiner. This is something like a soft hyphen without the hyphen. You can use it to indicate preferred break points in Web addresses, code, and other long strings of characters that are likely to be divided between lines but should *not* be hyphenated.

Without this, the Kindle may break the line randomly, wherever it runs out of room. Older Kindles may also insert an unwanted hyphen or hide part of the text beyond the right margin. Here, for example, is a Web address, first without the non-joiner, and then

with it. Adjust the font size on your Kindle to see the effect.

> www.aaronshep.com/kidwriter/books/Business.html
>
> www.aaronshep.com/kidwriter/books/Business.html

Actually, it's a fudge to say the Kindle supports the zero width non-joiner *character*. Older Kindles *do* support it, recognizing and acting on either the Unicode character or its entity when inserted into HTML. Newer Kindles, though, recognize the non-joiner *only* as one of its entity forms:

> ‌ *or* ‌

This is probably to let those Kindles supply the *function* of the non-joiner even with fonts that lack the Unicode character. But it makes it a bit trickier to work with the non-joiner in your source document. The app you compose in must not only support the character but must then convert it to entity form when exporting to HTML.

Luckily for users of Word for Windows, that app does both these things, offering the non-joiner on the Special Characters tab of its Symbol dialog box under

the name No-Width Optional Break. You're not so lucky, though, if you're working in one of Word's Hyperlink dialogs, which don't fully support special characters, or in Word for the Mac, which neither inserts nor translates this character.

In such cases, you might instead insert substitute code. For instance, you could insert something in your source document like "{zwnj}" or one or more vertical slashes (|)—anything your code editor can easily find later and replace with the non-joiner entity. The main disadvantage is you have to remember to remove the substitute code from your source document if your text goes to print! (I haven't always remembered.)

Why not just add the HTML entity directly into your source document? Because here's how it would look on export to HTML:

```
&amp;zwnj;
```

In other words, the initial ampersand would be turned into an ampersand *entity* so it wouldn't be confused with an ampersand in HTML code. This change would invalidate the entity you inserted and make it display as text!

For those who prefer not to mess up the source document with substitute code, another option is to insert the non-joiner directly into HTML. If you have only a narrow use for the non-joiner, you can handle this with a specialized find-and-replace.

For example, in *From Word to Kindle,* I have several hyperlinked addresses of Amazon KDP Help pages, varying only in the final identifier. Each one appears in a paragraph by itself—meaning I can count on HTML tags being placed immediately before and after. My initial solution was an operation like this:

Find: >kdp.amazon.com/help?topicId=

Replace: >kdp.amazon.com/help?‌topicId=

Note the closing angle bracket that begins both the Find and the Replace. Including this was essential, because it limited the operation to addresses found in *text*. Without it, the operation would also have changed addresses in HTML link code, making my links invalid! (In link code, such addresses are instead always preceded by a colon.)

Also note how I included text from after the final break point as well as before. This way, if the operation was repeated, it wouldn't add a second non-joiner. (An alternative would be to add a follow-up operation, finding two non-joiners together and replacing with one.)

Already specialized in function, the zero width non-joiner can be less versatile than you might think, due to the quirkiness and bugginess of Kindle support. In fact, I've given up trying to figure out all the positions in which it does or does not work properly. The list varies on different Kindles, and sometimes on the *same* Kindle.

Pinning it down is complicated by the fact that, even without the non-joiner, different Kindles break lines in different places.

My general advice, then, is to insert the non-joiner wherever you think it might be helpful, then hope for the best. But there are a couple of caveats, too. Most importantly, the non-joiner is best used in left-aligned text. That's because some older Kindles treat it more like a thin space, allowing text on either side to move apart when justified.

Also, you must take care never to place the non-joiner right after an HTML tag. On older e-ink Kindles, this can cause text from before the tag to repeat! To guard against slipups, you can run a safety operation that removes any non-joiner in that position. If you never use a right angle bracket (>) in text, a simple find-and-replace like the following will do.

Find: >‌

Replace: >

Otherwise, this grep will take care of it:

Find: (<[^>]*>)‌

Replace: \1

•

Whenever you adjust line breaking, there's a price to pay. In left-aligned text, it will make your lines less

even. In justified text, it will increase spacing between words—and in the absence of hyphenation, this can get to the point of looking really bad.

In print, you can often find a way to even things out within the paragraph, but that's obviously not possible on the Kindle. So, when deciding how much to adjust, you'll need to find a good balance, weighing the benefit of your changes against the cost.

Update!

With the introduction of Enhanced Typesetting in 2015 for the newest Kindles, Amazon has provided some nice improvements to Kindle typography. Unfortunately, to balance this, they have totally screwed up some things that were working before. Perhaps the most important of these is the Kindle's handling of nonbreaking spaces, which are in most cases now treated simply as regular spaces.

Reportedly, this was meant to prevent the nonbreaking space from being misused to cement long strings of text, which then extend past the right screen edge. In preventing that, though, Amazon has done away with legitimate usage as well. Whether this will eventually be fixed, I can't say.

Luckily, Enhanced Typesetting still honors "nonbreaking spans" such as I recommended as replacements for nonbreaking hyphens. These spans can replace the nonbreaking spaces or else supplement them—in other words, you could leave the nonbreaking spaces in place for older Kindles but create nonbreaking spans around them. An advantage to *replacing* them is that spacing between words on some Kindles will then be more even in justified text. (Note that, if you leave the nonbreaking spaces, any fix is likely to keep inserting additional tags if you run it more than once, so you'd have to correct for that.)

I'll leave you to work out the details. You might also want to come up with a fix for the way Enhanced Typesetting currently breaks decimals after a leading decimal point! (One possibility: Add a zero before the decimal point.)

•

Probably the most important new feature of Enhanced Typesetting is automatic hyphenation. This is a plus whether the text is justified or left-aligned, as it helps even out the lines. It raises the problem, though, of what to do for text you *don't* want hyphenated—like headings, poetry, or software code.

Turning off hyphenation in Word or another word processor won't make any difference in its exported HTML, but CSS to control hyphenation can be added directly. Note that the CSS declaration for this has not yet been standardized. But popular Web browser kernels—like WebKit, which Amazon uses to render Kindle text—provide temporary CSS property names to use meanwhile. And it's one of these temporary names that can currently control hyphenation on Kindle:

```
-webkit-hyphens
```

Don't omit the initial hyphen! It's essential! (That's how temporary CSS property names are normally constructed.) And keep it all lower case.

In CSS, it's fine to add declarations with alternate property names if you're not sure which one will work. So, even though this currently has no effect, I'd play safe by adding a separate declaration with the name most likely to become standard:

```
hyphens
```

Possible values for either of these properties are:

• “Auto,” for full hyphenation. This is the default for Enhanced Typesetting.

• “Manual,” for hyphenation only at existing hyphens and soft hyphens. This is the value you most likely want when turning off automatic hyphenation. (Remember, though, that *soft* hyphens cause problems on Kindle, so those shouldn’t be in your text!)

• “None,” for no hyphenation at all.

To apply one of these values to your entire document, you can add your declaration(s) to your opening body tag. Or you can apply them to individual CSS classes. Or you could do both, setting one value as a document-wide default and a contrary one for individual styles.

If you assign a value of “auto,” you might need a language attribute added to your opening html tag. For English, it would look like this:

```
<html lang="en">
```

When testing hyphenation control, keep in mind that the files Amazon produces for proofing are *not* the same files it provides to customers—and that even those files may be modified any time after publication. So, even when you don’t see these controls work in your proof or in an immediate purchase, they will hopefully work in your book later on.

In fact, if you *do* see hyphenation in your proof or immediate purchase, it may be a *different* kind of hyphenation than you would get later. Kindle on iPad and iPhone seems to have two separate hyphenation systems—a competent one that’s part of Enhanced Typesetting, and an older one that produces wild errors about half the time. In

some situations you'll get one system, in other situations, the other.

Yes, it's frustrating!

Fixes for Pictures

In the HTML for your Kindle book, pictures are inserted in the text by means of an image tag (<img>). Minimally, it might look something like this:

```
<img src="image01.jpg" width="576" height="864"
alt="">
```

As in this example, it's common to provide a picture's pixel dimensions within the tag. This is considered good practice, as it speeds up page rendering. On the oldest Kindles, it also prevents quirky picture handling. Without these stated pixel dimensions, those Kindles might automatically enlarge a small picture to the point of pixelation; or the text under a larger picture might be pushed unnecessarily to the next page.

Word is among the apps that will conveniently write these dimensions into HTML for pictures inserted in your document. Be aware, though, that on export, Word may also alter the picture from its original dimensions! (For the full story on that, see my book *Pictures on Kindle*.)

Note that older Kindles can read the picture's dimensions *only* in bare-bones HTML format, as shown above. Dimensions in CSS format—as may be written by some ebook creation software—will be ignored. The example above, but with the kind of CSS dimensions you *don't* want, would look like this:

```
<img src="image01.jpg" style="width: 576px;
height: 864px;" alt="">
```

In case you need it, here's a bit of grep for converting from CSS pixel dimensions to HTML—assuming you have no other CSS properties within the tag's style attribute. The apparently empty bracket pairs enclose spaces. Bracket pairs with straight quotes include both a single and a double. Of course, if your CSS gives height before width, you'll have to move things around.

Find: (<img[^>]*) style=['"]width:[ ]?([0-9]+)px;[ ]?
height:[ ]?([0-9]+)px[;]?['"]

Replace: \1 width="\2" height="\3"

After telling you how to properly include pixel dimensions and why, I also have to tell you that that's no longer the best practice. Because of the wide range of resolutions among Kindles, it's now best to *replace* these dimensions with CSS that simply tells the Kindle to display the picture as large as possible. Otherwise, your picture may appear much too small on high-resolution Kindles. And that's a bigger problem than any the replacement could cause on older ones.

To force a picture to display as large as possible on newer Kindles, use CSS to specify a width of 100% and a height of "auto"—or a width of "auto" and a height of 100%. Either way simply tells the Kindle to expand the picture as needed to fit between margins. If the picture

is already big enough to need no expansion, the instruction has no effect.

Here's the style attribute to place inside the picture's image tag—assuming the tag does not already have one.

```
style="width: 100%; height: auto;"
```

And here's the grep that will substitute this for the HTML pixel dimensions written by Word.

Find: <img width=[0-9]+ height=[0-9]+

Replace: <img style="width: 100%; height: auto;"

Misinterpreting Amazon's vague guidelines, Kindle book designers sometimes skip the CSS and simply specify an HTML image width of 100% to handle both older and newer Kindles. Though image size percentages were allowed in earlier HTML versions, HTML5 considers this an error—and in any case, older Kindles still can't use them. It may *appear* these Kindles are reading a percentage when the picture is enlarged—but that's just what they do when they get no dimensions at all. And they're still enlarging quirkily.

Of course, if you're going to enlarge your picture to the margins, you'll need to make sure it has enough pixels to display at a decent resolution. (Again, if you don't understand this, see *Pictures on Kindle*.)

•

Word is oddly prone to introducing HTML errors in its image tags. These errors are ignored by the Kindle converter, so there's no great need to fix them. But if you want to fix them anyway—say, to keep a syntax checker from reporting them—it's easy enough to do.

If you're exporting from the old .doc format—Word's native format in any version before 2007—add the following empty attribute anywhere within the tag. (Those are two straight double quotes, with no space between.) You can use a find-and-replace similar to the one I gave above for adding the style attribute.

```
alt=""
```

If you're exporting from .doc *and* are on a Mac, your image tags will have a problem also with their ID attributes, which will all look something like this:

```
id="_x0000_i1025"
```

ID values are not supposed to start with an underscore in older HTML versions such as used by Word, so just remove that.

If you're using the newer .docx format, on Mac or Windows, the ID attributes will have a different problem. You'll see them numbering the pictures, starting like this:

id="Picture 1"

The ID value should have no space within it, so just remove the space from between “Picture” and the numeral.

Or for either problem with the ID attribute, you can just remove the attribute entirely. It serves no useful purpose in your Kindle book. The following grep should work for any Word version.

Find: (<img[^>]*) id="[^"]+"

Replace: \1

Update!

Amazon has become stricter about the resolutions of “full-page” pictures. Basing its requirements on a maximum display size of 4 × 6 inches, Amazon wants any picture at those full dimensions to be at a resolution of at least 300 ppi, for a minimum of 1200 × 1800 pixels. Full-page pictures with lower resolutions may cause a book to be flagged for low quality or even taken off sale.

What this means in practical terms is that any picture stretched in HTML to 100% of display width must be at least 1200 pixels wide in its original dimensions. Any picture stretched to 100% of height must be at least 1800 pixels high.

In most cases, you would want that anyway. But it’s a special problem if your book is in Microsoft Word for Mac, with its image dimension limit of 22 inches and its forced export resolution of 72 ppi. Within those limits, you can’t do much better on a 2:3 image than 1008 × 1512 pixels—too short in either dimension. Only a horizontal or more-or-less square image could meet Amazon’s requirements.

If you’re using Word for Mac, then, you might consider switching to a more suitable app—Jutoh comes to mind. You could even just move your book to Word for Windows—maybe in a virtual machine—for its HTML export at 96 ppi. But if you’re sticking with Word for Mac, here is a grep sequence that will stretch the picture width *or* height to 100% in Word’s HTML *only* if the image has enough pixels in that dimension.

Find: <img width=1[2-9][0-9][0-9] height=[0-9]+
Replace: <img style="width: 100%; height: auto;"

Find: <img width=[2-9][0-9][0-9][0-9] height=[0-9]+
Replace: <img style="width: 100%; height: auto;"

Find: <img width=[0-9]+ height=1[8-9][0-9][0-9]
Replace: <img style="width: auto; height: 100%;"

Find: <img width=[0-9]+ height=2[0-9][0-9][0-9]
Replace: <img style="width: auto; height: 100%;"

Fixes for Navigation

Most users of older Kindles have seen this problem when jumping somewhere in a Kindle book, perhaps when following a link in the text or the table of contents: The chapter heading loses its formatting and appears in the Kindle's default paragraph style. And if you then try to move to the page previous, it takes two attempts.

If you've seen that yourself, chances are you've written it off as one of those obscure bugs that plague most young computing devices. But it's not. Mobipocket, the Kindle's original format, was actually *designed* to work that way. And when you understand why, you can stop it from happening.

Let's look at sample code for a typical chapter heading you might reach from a link.

```
<h1><a name="chapter1"></a>Chapter 1</h1>
```

"Chapter 1" is the text of the heading—what you see on the Kindle screen—and the entire phrase is enclosed by the opening and closing heading tags. The remainder of the code, which I've shown in bold, is called an HTML *anchor*. The opening tag of the anchor is the actual target of the link that brings you here, with the value of its name attribute serving as a unique identifier.

This sample is actually close to the way Word would write the anchor as it converts bookmarks in your

document into HTML. In Word, bookmarks—whether set manually or automatically—are how you mark destinations for the internal links, table of contents, and Go To menu items for your Kindle book.

The position of this anchor's closing tag—immediately following the opening tag—is what you get when you set your Word bookmark with the cursor in the text but nothing selected. But if you were to select the heading text, "Chapter 1"—or if you were to let Word generate a table of contents automatically—the anchor tags would instead enclose the text like this:

<h1>****Chapter 1****</h1>

The different position of the closing tag doesn't matter to the HTML, as that tag doesn't actually do anything—it's just a formality, because in HTML, you usually need to close what you open. We just have to be aware of the variation if we're going to manipulate the tag.

•

OK, we've seen how a typical heading with anchor is constructed, and you may already have guessed why older Kindles might have trouble with it. But before I spell it out, let's look at the most basic difference between Mobipocket and its competitor, EPUB, as well as Mobipocket's successor and EPUB's near relation, Kindle Format 8.

Nowadays, we're used to having gigabytes of memory on our computers, and these have no trouble holding huge documents in memory for rapid processing. But the situation was very different with the cell phones for which Mobipocket and EPUB were originally developed. In these phones, memory was miniscule. As small as is the typical file of an ebook, there was no way to hold the whole thing in the phone's memory at once.

To deal with this, two different strategies emerged. The strategy of EPUB was to break down the book into tiny files—typically a chapter each—and keep that much in memory. This let the software jump back and forth easily—as long as that was in the same chapter. But with the extremely weak processors in the phones, the price of this arrangement was slow response time.

The Mobipocket format was a radical departure from this. It held *none* of the book in memory other than the page being displayed. It simply moved forward and backward in the file. Forward and backward, in a straight line, with no memory of what came before or after. This made it quicker than EPUB, but also, well, *stupid*.

Let's get back to our sample heading. Let's say it was in an EPUB ebook. If you jumped to this heading, the software would locate the proper file and load it into memory. Then it would look at the heading—*the entire line of code*—and know that it *was* a heading. And it would know exactly how to format it.

Now let's say that it's a Kindle book on an older Kindle. You jump to the heading. The Kindle moves forward through the file until it locates the anchor. Then it displays enough of the text following to fill the screen.

What's missing? The opening heading tag! It's right there in front of the anchor, but the Kindle can't read backwards and has no memory of what came before! So, with no clue how the text should be formatted, it reverts to default paragraph formatting.

Now try to move to the page previous. (Also assume, if you will, that there's a page break just above the heading.) What happens? Nothing—or at least nothing *seems* to happen. But what *really* happens is that the Kindle moves you from your position in front of the opening *anchor* tag to a new position in front of the opening *heading* tag—the true top of the page. And of course, from that position, you can move to the previous page on the next try.

I think you can understand now the main disadvantage of the Mobipocket approach, and maybe also one way to work around it. Yes, all we would have to do is move the anchor—the opening and closing tags together—to a new position *in front of* the opening heading tag. Then, when you jumped to that position, the Kindle would read the heading tag that follows and know what to do about formatting. It would also be in position to properly move to the previous page. Though this anchor placement is not standard HTML, your syntax editor would not likely object to it, either.

After this change, our example would look like this:

```
<a name="chapter1"></a><h1>Chapter 1</h1>
```

This is in fact the solution I recommended in earlier versions of this book. But even though it works fine for Kindle, placing the anchor in front can cause problems if you later convert your HTML or MOBI file to EPUB. In some EPUB readers, links may then land you on the page *before* the target, and in rare cases, you may even see a blank page inserted.

A better solution is to convert the anchors to a different format entirely. HTML allows you to turn *any* opening tag into an anchor by simply adding an ID attribute ("id"). Continuing with our example, it would look like this:

```
<h1 id="chapter1">Chapter 1</h1>
```

This form of anchor avoids problems both in EPUB and on all Kindles.

•

With the unique attribute values and the variations in tag position, it's obviously going to take more than a simple find-and-replace to convert the anchors automatically. But here again, grep comes to the rescue.

The following operation will use the value of an anchor's name attribute to form a new ID attribute in the heading's or paragraph's opening tag. At the same

time, it will remove the original anchor. The characters in the second and third pair of square brackets are a single and double quote. The question marks are needed to prevent "greedy" searches for the longest match. (Notepad++ users: Also to prevent greedy searches, this is one time you must be absolutely sure the option ". matches newline" is *off*!)

Find: (<[^>]*)(.*)(.*?)(.*)

Replace: \1 id="\3"\2\4\5

This operation comes with just a few caveats:

• It must be performed *after* you've removed unneeded line break characters.

• You should not have more than one anchor per heading or paragraph. If you do, the operation will convert only one of them at a time. And though a repeat of the operation would remove a second anchor, it would add a second ID attribute to the opening tag, producing a syntax error.

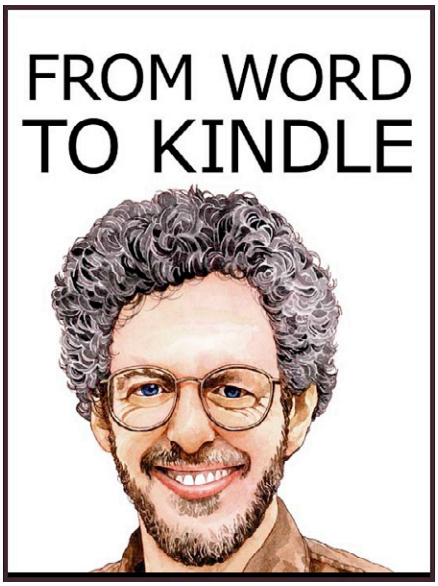
• The operation won't work if the opening and closing anchor tags enclose a link in the text. So, either make sure those anchor tags are adjacent, or keep them away from links.

• For safety, make sure that bookmarks in the source document include only those needed for navigation of the Kindle book. Also make sure that no paragraph includes more than one of your own bookmarks. (For

tips on viewing, editing, and cleaning up bookmarks in Word, see *From Word to Kindle*.)

This is a particularly powerful operation, so be sure to test carefully!

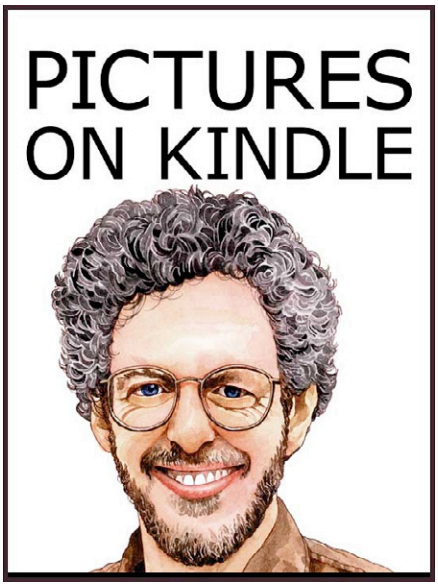
MORE BOOKS FOR YOU



From Word to Kindle

Self Publishing Your Kindle Book with Microsoft Word, or Tips on Formatting Your Document So Your Ebook Won't Look Terrible

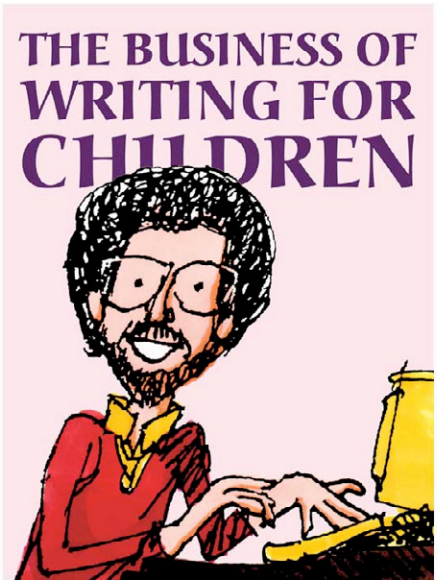
By Aaron Shepard



Pictures on Kindle

Self Publishing Your Kindle Book with Photos, Art, or Graphics, or Tips on Formatting Your Ebook's Images to Make Them Look Great

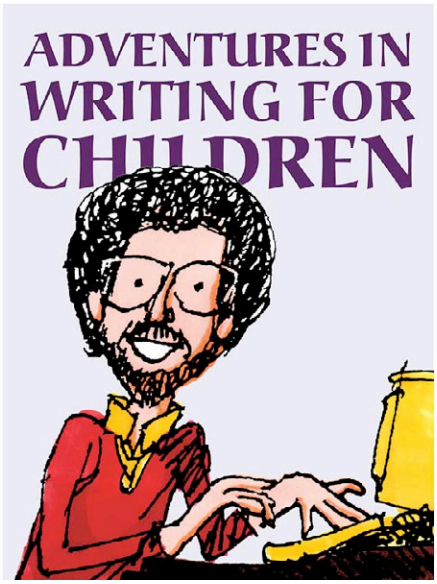
By Aaron Shepard



The Business of Writing for Children

An Award-Winning Author's Tips on Writing Children's Books and Publishing Them, or How to Write, Publish, and Promote a Book for Kids

By Aaron Shepard



Adventures in Writing for Children

More Tips from an Award-Winning Author on the Art and Business of Writing Children's Books and Publishing Them

By Aaron Shepard

Aaron Shepard's
Sales Rank Express

Quickly Check Amazon Sales Ranks and More for Print Books, Kindle Books, and Audiobooks on Amazon Worldwide with the Premier Sales Rank Checker, Book Monitor, and Market Research Tool for Authors and Publishers

www.salesrankexpress.com