O'REILLY®

The Little Book of HTML/CSS Coding Guidelines

Jens Oliver Meiert
Foreword by Lindsey Simon

The Little Book of HTML/CSS Coding Guidelines

Jens Oliver Meiert

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

The Little Book of HTML/CSS Coding Guidelines
by Jens Oliver Meiert

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meg Foley
Production Editor: Nicole Shelby
Copyeditor: Jasmine Kwityn
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

December 2015: First Edition

Revision History for the First Edition
2015-11-19: First Release

978-1-491-94257-4

[LSI]

For Michael Sage—

"Organization is not everything, but without organization, everything is nothing."

Table of Contents

Foreword

Style guides and coding conventions might sound like something creativity-encoraching—like painting inside the lines—but as a solo developer, and especially when working in larger teams, style guidelines tend to remove the least creative decisions and allow people to focus on what matters—solving problems in code to make users' lives better.

I met Jens at Google, and was an avid observer and sometimes collaborator with his work on the webmaster team there—we were both excited about trying to help codify and teach best practices for web development. Jens and I both worked on and shared a desire to build tools to help automate the decisions that we labored over and it was fantastic to see how appreciative the teams were for insights into the craft and for the ability of the tools Jens worked on to both point out mistakes automatically or even correct them where possible. As we both worked to decrease cognitive load, ditch double break tags, and educate people about usability and accessibility, more teams came to adopt coding guidelines to help speed up their development, and stylistic nitpicking and confusion gradually receded in the code review process.

Readability of code should be the goal of anyone in our field, much as the *AP Stylebook* is a resource for some of the best news organizations in the world. The rules can always be changed, but having a sound and solid framework on which to build your next great idea will make it that much easier to repurpose and share your efforts for the betterment of users, and possibly other developers you may get to work with. I've heard Dan Cederholm and Peter Paul Koch wax poetic about the craft of web development—style guides and improved readability are evidence of care for the craft.

—Lindsey Simon (former tech lead at Google)

The Little Book of HTML/CSS Coding Guidelines

Introduction

> *"It turns out that style matters in programming for the same reason that it matters in writing. It makes for better reading."*
>
> ***—Douglas Crockford***

Coding guidelines govern how we write code.[1]

Sometimes called standards, sometimes conventions, they can govern many code-related things. Wikipedia, for example, tells us that

> Coding conventions are a set of guidelines for a specific programming language that recommend programming style, practices, and methods for each aspect of a piece program written in this language. These conventions usually cover file organization, indentation, comments, declarations, statements, whitespace, naming conventions, programming practices, programming principles, programming rules of thumb, architectural best practices, etc.

Most of the time, we find coding guidelines in big organizations and large projects. As individual developers, perhaps even hobbyist developers, we don't need and perhaps appreciate them that much. But in those big organizations and large projects, coding guidelines

1 Throughout the book, I keep with the term *coding guidelines*, and use it liberally. I also apply it holistically—that is, I use this term to denote serious sets of guidelines that try to comprehensively define the formatting of all respective code, and not just represent a weak recommendation to "please indent." Normally, coding guidelines will apply to non-minified, non-compressed working code. Live code (i.e., production code) constitutes an exception to most formatting guidelines.

are critical. Software and web development leave a lot of room for preference, and preference makes for a lot of inconsistency and confusion, if not kept at bay.

As Wikipedia suggests, coding guidelines go beyond formatting; they can also cover development principles, and with that direct development with an even firmer grip.

In this *Little Book*, I share my experience with HTML and CSS coding guidelines. Why me and why guidelines for HTML and CSS? A web developer by trade, and one who's closely following the development of web standards, I'm most familiar with HTML and CSS. And I'm similarly familiar with coding guidelines. Ten years ago, I introduced HTML/CSS rules at GMX, the largest email provider in Germany. When I joined top agency Aperto, I did the same thing and created, together with Timo Wirth, guidelines that ruled all frontend code, including Aperto's large commercial and governmental customers. And later, I took the opportunity at Google to found a team and with that team revise Google's CSS guidelines and create all new HTML guidelines.

The two most fundamental lessons I learned were that coding guidelines absolutely are a cornerstone of professional web development, and second (and in contrast to this), that it's easier to set them up than to get them followed. And this brings us into a good position to start.

Acknowledgments

I'd like to thank Tony Ruscoe for his always friendly and professional help checking and improving my technical writing. I thank the O'Reilly team, notably Simon St. Laurent and Meg Foley, for their advice and help on getting another *Little Book* out (following *The Little Book of HTML/CSS Frameworks*). And, regarding the matter at hand, I like to thank all the many people I've worked with who showed and taught me how (not to) work with coding standards.

Thanks, too, go to Harry Roberts, Dan Hay, as well as Google's and WordPress's developers for all their work on coding standards (and permission to quote within this book).

The Purpose of Coding Guidelines

Let's imagine a world without coding guidelines. Or a company without coding guidelines. Or, perhaps, ourselves without coding guidelines.

For example, consider the following heap of HTML code:

```
<table cellpadding="0" cellspacing="0" border="0" summary="">
<tr valign="middle">
<td  width="1"  height="21"  class="nav-border"><img  src="/img/
pool/transparent.gif"   width="1"   height="1"   alt=""   bor
der="0" /></td>
<td class="nav-3"> </td>
<td class="nav-3"><span class="nav-on">Home</span></td>
<td class="nav-3"> </td>
<td  width="1"  class="nav-border"><img  src="/img/pool/transpar
ent.gif" width="1" height="1" alt="" border="0" /></td>
<td class="nav-1"> </td>
<td   class="nav-1"><a   href="/de/column/"   class="nav">Arti-
kel</a></td>
<td class="nav-1"> </td>
<td  width="1"  class="nav-border"><img  src="/img/pool/transpar
ent.gif" width="1" height="1" alt="" border="0" /></td>
<td class="nav-1"> </td>
<td  class="nav-1"><a  href="/de/resources/"  class="nav">Empfeh-
lungen</a></td>
<td class="nav-1"> </td>
<td  width="1"  class="nav-border"><img  src="/img/pool/transpar
ent.gif" width="1" height="1" alt="" border="0" /></td>
<td class="nav-1"> </td>
<td   class="nav-1"><a   href="/de/download/"   class="nav">Down-
loads</a></td>
<td class="nav-1"> </td>
<td  width="1"  class="nav-border"><img  src="/img/pool/transpar
ent.gif" width="1" height="1" alt="" border="0" /></td>
<td class="nav-1"> </td>
<td          class="nav-1"><a          href="/de/about/"
class="nav">&Uuml;ber ...</a></td>
<td class="nav-1"> </td>
<td  width="1"  class="nav-border"><img  src="/img/pool/transpar
ent.gif" width="1" height="1" alt="" border="0" /></td>
</tr>
<tr>
<td   height="1"   class="nav-border-hrz"   colspan="21"><img
src="/img/pool/transparent.gif" width="1" height="1" alt="" bor
der="0" /></td>
</tr>
</table>
```

Then compare it to this:

```
<ul class="nav">
    <li>Startseite</li>
    <li><a href="/de/publications/">Publikationen</a></li>
    <li><a href="/de/biography/">Biographie</a></li>
    <li><a href="/de/contact/">Kontakt</a></li>
</ul>
```

Or compare this CSS code:

```
table { background-color: #FFC; border-bottom: 1px solid
#CCCC9D; border-top: 1px solid #CCCC9D; empty-cells: show;
font-size: 1em; margin: 1em 0 0; width: 100%; }
caption, form div label { display: none; }
th, td { vertical-align: baseline; }
th { font-weight: 700; padding: .5em .7em; text-align: left;
white-space: nowrap; }
td { border-top: 1px solid #E6E6B1; padding: .2em .7em; }
td a { line-height: 150%; }
```

to the code shown here:

```
.nav {
    border-bottom: 2px solid;
    line-height: 1.5;
    padding: 71px 8.75em 2px 6.75em;
}

.nav li,
.nav li a {
    padding: 0 4px;
}

.nav li {
    margin: 0 2px;
}

.nav li a {
    margin: 0 -4px;
}
```

That is code from the same person: the author in 2002, and the author in 2005.

What do we notice? The first thing we see is that the code is written completely differently. It's inconsistent. Would we want to work on it? Probably not. Would we be *able* to work on it? Maybe.

What would change this? Focusing on high quality and an intelligible, consistent formatting of all this code.

That is the job of coding guidelines.

Coding guidelines should yield quality, produce consistency, and through that, indirectly, assist usability, collaboration, and maintainability. They may not need to do all of this—which we'll cover under "Approaches to Coding Guidelines" on page 10—and they may not succeed, but that's their purpose.

Let's look at all of these points in detail.

Consistency

The major, direct benefit of coding guidelines is improved consistency. Why? Because with comprehensive coding guidelines *all code gets formatted the same way*. Rules are always indented the same way. Declarations appear in the same order. Element names are always lowercase.

Consider this example:

```
#intro {
  background: #fff;
  color: #000;
}

.note {
  color: gray;
  background: white
}
```

Suppose you need to edit this style sheet. How do you specify and order the colors for a new author section? Meet *inconsistency*.

While one might argue that keeping the guidelines in mind makes the process of writing code itself a little slower, locating and refactoring code becomes much easier and faster.

Usability

An indirect benefit that follows consistency is improved usability. Improved developer usability, that is. Improved "ease of use and learnability of code," then, as I described in *The Little Book of HTML/CSS Frameworks*. Why? Because through coding guidelines, developers are able to set and trust expectations, which again helps locating and refactoring code.

Collaboration

More importantly, yet also consequentially, coding guidelines facilitate collaboration. They make it easier for you to understand your colleagues' code (and vice versa), and to hand over code to someone you haven't work with previously. They don't require as much time adjusting to someone else's coding style, especially not when one follows the otherwise laudable habit of sticking to the code style a given project is using.

Maintainability

Lastly, coding guidelines and the consistency they bring to our code help maintainability. They do so because guidelines constitute a form of organization, a lower degree of entropy, which makes it easier to order, or keep things in order. Although often forgotten, maintainability important, as there's no code in existence that will only be touched once. Even if it's not going to be edited or updated again, eventually it must be decommissioned. And that falls under maintenance, too.

Anatomy of a Coding Guideline

What exactly is in a coding guideline? Isn't that just a command like, "do *x*"? In its simplest form, yes. But coding guidelines can and should entail more detail, and then it's on the purpose and importance of the rule to prove value.

Structure

At this point, we should work with a few examples. Let's look at a few random coding guidelines, without judgment nor endorsement:

Harry Roberts' CSS Guidelines recommend hyphens:

> ***Hyphen Delimited***
>
> All strings in classes are delimited with a hyphen (-), like so:
>
> ```
> .page-head {}
>
> .sub-content {}
> ```
>
> Camel case and underscores are not used for regular classes; the following are incorrect:
>
> ```
> .pageHead {}
>
> .sub_content {}
> ```

Dan Hay's coding standards say the following about "verbose" HTML code:

> ***Don't use tags that STADN (sit there and do nothing)***
>
> STADN tags do just that—they don't actually contribute much to the content or layout of a page. An example of a STADN tag would be:
>
> ```
> <FONT SIZE=2><B> </B></FONT>
> ```
>
> The bold and font tags do not contribute to the layout or appearance of the non-breaking space. We could add as many surrounding tags to the non-breaking space and it still wouldn't affect the appearance of the page.
>
> Most HTML editors liberally insert STADN tags. This behavior is yet another reason why HTML editors must not be used.

(A comment, "tag" should rather say "element" here.)

And for WordPress, vendor-specific extensions are worth special attention:

> We use grunt-autoprefixer as a pre-commit tool to easily manage necessary browser prefixes, thus making the majority of this section moot. For those interested in following that output without using Grunt, vendor prefixes should go longest (`-webkit-`) to shortest (unprefixed). All other spacing remains as per the rest of standards.
>
> ```
> .sample-output {
>     -webkit-box-shadow: inset 0 0 1px 1px #eee;
>     -moz-box-shadow: inset 0 0 1px 1px #eee;
>     box-shadow: inset 0 0 1px 1px #eee;
> }
> ```

(Legal note: This coding guideline has been quoted from the CSS Coding Standards by WordPress, used under GPLv2.)

These guidelines, and guidelines in general, are very differently written, but we find similarities:

- What (not) to do
- Scope
- Examples
- Explanation

These are the main ingredients of a coding guideline.

Let's have a closer look at this structure:

What (not) to do
: We've seen with our suspicion whether "do *x*" already suffices, the key part of a guideline. We cannot do without it.

Scope
: Knowing what the guideline applies to is sometimes evident ("sort all CSS declarations alphabetically" already clarifies the scope), sometimes not ("indent by two spaces"—indent what, when, where?). For that uncertainty the scope is generally important, too.

Examples
: Here things get more blurry in that a well-written rule may not need examples; however, in practice we observe that examples do help. Glancing at a rule and an example clarifies and helps colleagues with less experience to get a solid enough idea to know when to apply a rule "when they see it." Examples may need counter-examples—that is, we should show what is expected and correct according to the rule, and then what would be incorrect.

Implementation help
: Ideally, a coding guideline comes with a tip on how to use it, to make following it easier. For example, "use configuration file *x* for your editor to enforce indentation," "include script *y* to have your code validated," or "covered by linter." Although this is a very useful component of a well-written coding guideline, it is often overlooked (even in this booklet).

Explanation
: Although this is not always required, an explanation allows us to help our colleagues *understand* what the context and purpose is, and facilitate improving or vetoing the rule in question. In a very authoritative setting, explanations may not be as welcome, but in a cooperative one, they are. As domain experts, we should be able to explain *why* we do what we do, as with imposing guidelines.

What else
: Finally, a complete coding guideline should include an appropriate level of detail. I'd like to keep with the idea of the ideal ID or class name—as long as necessary and as short as possible. Bearing this in mind, when working on a coding standard, it's better to err on the side of adding enough detail so that the team can understand the guideline and its rationale.

With that, we should have an idea of the *minima* and *maxima* of a coding guideline:

Minima

- What (not) to do
- Scope
- Example
- Detail: brief

Maxima

- What (not) to do
- Scope
- Examples
- Implementation help
- Explanation
- Detail: verbose

Priority

But is the structure all that makes a coding guideline? Let's consider the ever-popular order to indent by *x* as well as the ever-beloved idea to use "semantic markup." What makes them different?

I believe we will soon discern a difference in terms of preference versus quality.

The indentation rule is first and foremost preference, especially when noting that tab characters can be configured to be displayed with *n* spaces, meaning that every team member could produce code that's indented the same way while still enjoying their own individual preferences.

The semantic markup rule, however, has a qualitative bearing, for if we understand the use of markup according to its meaning paramount to it being parsed correctly and accessibly, then this rule results in a difference in quality of code, depending on whether and how it's followed.

For coding guidelines, then, this difference results in a sense of priority. Though preference-based rules are still relevant because they lead to consistency, which in turn gives us all the benefits we discussed earlier (usability, collaboration, maintainability), the quality rules, when sound, make code more consistent *and* better.

The suspicion grows that preference rules are easier to define and spot than quality rules, but the jury's still out on that.

Approaches to Coding Guidelines

How do we then set up and promote coding guidelines?

That approach is best based on the difference between reality and goals. How does our code currently look? How should it look going forward?

We can learn from the approach taken by linguists: they call grammars prescribing how people ought to speak or write *prescriptive grammars*, and those describing what people actually use *descriptive grammars*.

Let's see how this can be applied to coding guidelines, and what else is involved.

Descriptive

The descriptive approach works if the difference between code reality and our goals is minor. Then we can simply outline how things are done now, let the whole *mélange* sit for a few minutes, and reap the reward when we onboard new team members.

For example, if everyone on the team is validating their HTML code, as it should be done (there's no need and no excuse for not using HTML correctly), we say:

> *Release only valid HTML code*

Prescriptive

If the reality/goal difference is bigger, we want to take a *prescriptive* (i.e., *normative*) approach, meaning to tell what to do:

> *Release only valid HTML code*

But isn't that the same rule?

It is the same rule on the surface, yet a different one when looking at the context. Whether we describe or prescribe coding standards doesn't depend on the rule, but on the situation. In both cases, we want to anchor, in writing, what code we expect.

The prescriptive approach, then, depends on enforcement: when everything's good already and we only describe, there's little need to enforce.

Once there's something to prescribe, there's also something to enforce. We'll look at this "Coding Guidelines in Practice" on page 12.

Mixed

Yet then, in everyday coding life, we face coding practices we want to document (describe), and others we want to achieve (prescribe). This means that most coding guidelines and standards include rules that are mixed, using both approaches.

Decision Process

How do we decide when to use which coding guidelines? The flowchart in Figure 1 can help us:

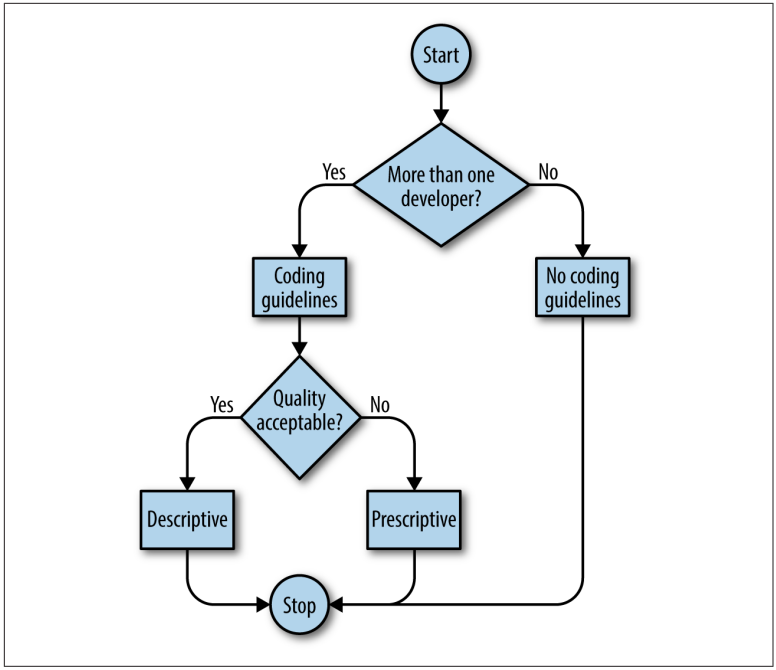

Figure 1. A flowchart for choosing an approach to coding guidelines

What we can see is that for a team of one, we don't strictly need coding guidelines. It is recommended, however, to look into using coding guidelines even in this case—perhaps making use of public ones, such as the Google HTML/CSS Style Guide with the exception of two-space indentation (even after leaving Google, I still follow these guidelines for my personal projects).

Whenever two or more people work together, however, coding guidelines become useful, and really important. And there the question is one of goals, and existing quality, to say whether we need a descriptive or prescriptive approach, considered for each guideline.

Coding Guidelines in Practice

This section briefly outlines special aspects of coding guidelines that we must consider when setting them up.

Communication

The larger the organization we're working in, the more important is the point of *communicating* our guidelines: Everyone writing code should know about them.

Fortunately, in most modern companies, teams have mailing lists to communicate guidelines to. It makes sense to share updates the same way, or to add all relevant people to a special mailing list related to coding style.

Compliance

The next important aspect is achieving compliance—that is, enforcing the guidelines. This is normally a two-fold process.

First, we need to *measure* whether coding guidelines are followed or not. For that, we need to set up the necessary infrastructure and tools, though manually probing for compliance, as with code reviews, does work, too. In practice, this piece is neglected rather frequently, and organizations don't know much about their actual compliance rates. Automation, which we will look at momentarily, is crucial here. *How* to automate the whole compliance part is not subject of this booklet, however.

Second, we need to *enforce* the code style we want to see. Here, too, automation is desirable, but we also need a way to track and score offenders. Tying coding style compliance to performance metrics that got communicated in advance is an effective approach. For example, a team member who repeatedly violates coding standards could get a lower performance rating than one who does keep with it.

Reviews

Our coding guidelines should not be considered a one-off effort. Just as we must maintain our code, so too should our guidelines be reviewed from time to time—it's important to update the documentation to reflect changes to guidelines as they arise.

It is something that gets maintained (as much as the affected code—we should not forget to update it when guidelines change). It is therefore recommended to not only assign a primary contact (or perhaps a small team of experienced volunteers) to be guideline

owners, but to also schedule at least quarterly reviews that check whether updates are needed.

Automation

Lastly, a particularly useful habit—and a key for future handling of coding guidelines—is automation. The assessment of code quality should be automated as much as possible and we should also automate improving and fixing code.

At the moment, there is no single out-of-the-box solution for this (only small scripts abound), but our vision overall should be that our development environment shows us local coding preferences, highlights violations and fixes them for us; that then, when we stage our code, additional checks are run that likewise report issues and fix them, and that at the end, optimized, minified, compressed, our code goes live in the shape we had envisioned it.

Proven HTML/CSS Coding Guidelines

After this short run through coding guidelines, I want to make recommendations for what I consider solid, useful, proven coding guidelines. Much of what follows can also be found in the Google HTML/CSS Style Guide, but that shouldn't be surprising given Google's care in most matters engineering.

Many of these guidelines are quality rather than preference guidelines. We'll keep with a bit more than just the minima: with what (not) to do in what scope, examples that illustrate each point, a rationale, and that with just the detail we need.

(Legal note: The following guidelines are a derivative of the HTML/CSS Style Guide by Google, used under CC BY 3.0 by Jens Oliver Meiert.)

General

Use UTF-8 (No Byte Order Mark)

Make sure your editor uses UTF-8 as character encoding, without a byte order mark.

Specify the encoding in HTML templates and documents via `<meta charset="utf-8">`. Do not specify the encoding of stylesheets, for these assume UTF-8 by default.

Omit the Protocol from Embedded Resources

Omit the protocol portion (`http:`, `https:`) from URLs unless the respective files are not available over both protocols.

Omitting the protocol—which makes the URL relative—prevents mixed content issues and results in (albeit tiny) extra file size savings.

Correct:

```
<script src="//www.google.com/js/gweb/analytics/
autotrack.js"></script>
```

Indent by One Tab

Only use tab characters for indentation. [Except for in this book ;)]

Correct:

```
<ul>
    <li>HTML
    <li>CSS
</ul>
```

Use Only Lowercase

Where possible, code should be lowercase: this includes HTML element names, attributes, attribute values (unless `text/CDATA`), CSS selectors, properties, and property values (with the exception of strings, because case can be relevant here).

Correct:

```
color: #cc0078;
```

Incorrect:

```
<A HREF="/">Home</A>
```

Remove Trailing Whitespace

Trailing whitespace is unnecessary, as it can complicate diffs.

Incorrect:

```
<p>What?_
```

(...where "_" signifies a space character.)

Mark TODOs and Action Items with TODO

Highlight TODOs by using the keyword `TODO` only.

Append a contact (username or mailing list) in parentheses as in `TODO(contact)`.

Correct:

```
<!-- TODO(john.doe): revisit centering -->
<center>Test</center>
```

HTML

Use HTML 5

Use HTML 5 (HTML syntax) for all HTML documents: `<!DOCTYPE html>`.(this spelling is for historical reasons).

Although technically correct, do not close void elements—write `<br>`, not `<br />`.

Use HTML According to Purpose

Use elements for what they have been designed for. For example, use heading elements for headings, `p` elements for paragraphs, `a` elements for anchors, and so on.

Using HTML according to its purpose is important for accessibility, reuse, and code efficiency reasons.

Use Valid HTML

Dto.: Use valid HTML.

Use tools such as the W3C HTML validator to test.

Using valid HTML is a baseline quality attribute that ensures proper HTML use and contributes to learning about technical constraints.

Correct:

```
<!DOCTYPE html>
<meta charset="utf-8">
<title>Test</title>
<article>This is only a test.</article>
```

Provide Alternative Contents for Multimedia

For multimedia, such as images, videos and animated objects via canvas, make sure to offer alternative access. For images, that means use of meaningful alternative text (`alt`); video and audio transcripts or captions should also be provided, if available.

Providing alternative contents is important for accessibility reasons, for not all multimedia contents are equally accessible to users.

Correct:

```
<img src="muscles.jpg" alt="Medical illustration of the mus
cles in the leg.">
```

Separate Structure from Presentation from Behavior

Strictly keep structure (markup), presentation (styling), and behavior (scripting) apart, and keep the interaction between the three to an absolute minimum.

That is, make sure documents and templates contain only HTML and HTML that is solely serving structural purposes. Move everything presentational into style sheets, and everything behavioral into scripts. Link as few style sheets and scripts as possible from documents and templates.

Separating structure from presentation from behavior is important for maintenance reasons. It is always more expensive to change HTML documents and templates than it is to update style sheets and scripts.

Do Not Use Entity References

There is no need to use entity references like `&mdash;`, `&rdquo;`, or `&#x263a;`, assuming the same encoding (UTF-8) is used for files and editors as well as among teams.

The only exceptions apply to characters with special meaning in HTML (like `<` and `&`) as well as control or "invisible" characters (like no-break spaces).

Correct:

```
<p>The currency symbol for the Euro is "€".
```

Omit Optional Tags

For file size optimization and scannability purposes, omit optional tags. (Refer to the HTML 5 specification for what tags can be omitted.)

Correct:

```
<!DOCTYPE html>
<title>Saving Space</title>
<p>Qed.
```

Omit type Attributes for Style Sheets and Scripts

Do not use `type` attributes for style sheets (unless not using CSS) and scripts (unless not using JavaScript).

Specifying type attributes in these contexts is not necessary as HTML5 implies `text/css` and `text/javascript` as defaults. This can be safely done even for older browsers.

Correct:

```
<link rel="stylesheet" href="//example.com/default.css">
```

Use a New Line for Every Block, List, or Table Element, and Indent Every Such Child Element

Independent of the styling of an element (as CSS allows elements to assume a different role per display property), put every block, list, or table element on a new line.

Also, indent them if they are child elements of a block, list, or table element.

(If you run into issues around whitespace between list items, it's acceptable to put all `li` elements in one line. A linter is encouraged to throw a warning instead of an error.)

Correct:

```
<table>
    <thead>
        <tr>
            <th scope="col">Income
            <th scope="col">Taxes
    <tbody>
        <tr>
            <td>$ 5.00
            <td>$ 4.50
</table>
```

When Quoting Attribute Values, Use Double Quotation Marks

Use double (""), not single quotation marks (''), around attribute values.

Correct:

```
<a class="action promo">Sign in</a>
```

CSS

Use Valid CSS Where Possible

Unless dealing with CSS validator bugs or requiring proprietary syntax, use valid CSS code.

Use tools such as the W3C CSS validator to test.

Using valid CSS is a baseline quality attribute that allows us to spot CSS code that may not have any effect and can be removed, and ensures proper CSS usage.

Avoid User Agent Detection and CSS "Hacks"

It's tempting to address styling differences over user agent detection or special CSS filters, workarounds, and hacks. Both approaches should be considered as a last resort in order to achieve and maintain an efficient and manageable code base. Put another way, giving detection and hacks a free pass will hurt projects in the long run as projects tend to take the way of least resistance. That is, allowing and making it easy to use detection and hacks means using detection and hacks more frequently—and more frequently is too frequently.

Use Functional or Generic ID and Class Names

Instead of presentational or cryptic names, always use ID and class names that reflect the purpose of the element in question, or that are otherwise generic.

Names that are specific and reflect the purpose of the element should be preferred, as these are most understandable and the least likely to change.

Generic names are simply a fallback for elements that have no particular or no meaning different from their siblings. They are typically needed as "helpers."

Using functional or generic names reduces the probability of unnecessary document or template changes.

Incorrect:

```
/* Meaningless */
#yee-1901 {}

/* Presentational */
.button-green {}
.clear {}
```

Correct:

```
/* Specific */
#login {}
.video {}

/* Generic */
.aux {}
.alt {}
```

Use ID and Class Names that Are as Short as Possible but as Long as Necessary

Try to convey what an ID or class is about while being as brief as possible.

Using ID and class names this way contributes to acceptable levels of understandability and code efficiency.

Incorrect:

```
#navigation {}
.atr {}
```

Correct:

```
#nav {}
.author {}
```

Prefix Selectors with an Application-Specific Prefix Where Safer

In large projects and for all code that gets embedded in other projects or on external sites, use prefixes (as namespaces) for ID and class names. Use short, unique identifiers followed by a dash.

Using namespaces helps prevent naming conflicts and can make maintenance easier (e.g., in search-and-replace operations).

Correct:

```
.foo-help {}
#bar-note {}
```

Use Shorthand Properties Where Possible

CSS offers a variety of shorthand properties (like font) that should be used whenever possible, even in cases where only one value is explicitly set.

Using shorthand properties is useful for code efficiency and understandability.

Incorrect:

```
border-top-style: none;
font-family: palatino, georgia, serif;
font-size: 100%;
line-height: 1.6;
padding-bottom: 2em;
padding-left: 1em;
padding-right: 1em;
padding-top: 0;
```

Correct:

```
border-top: 0;
font: 100%/1.6 palatino, georgia, serif;
padding: 0 1em 2em;
```

Omit Units After 0 Values

Do not use units after 0 values unless they are required.

Correct:

```
margin: 0;
padding: 0;
```

Omit Leading 0s in Values

Do not use put 0s in front of values or lengths between –1 and 1.

Correct:

```
font-size: .8em;
```

Use Three-Character Hexadecimal Notation Where Possible

For hexadecimal color values, three-character hexadecimal notation is shorter and more succinct.

Correct:

```
color: #ebc;
```

Separate Words in ID and Class Names by a Hyphen

Do not concatenate words and abbreviations in selectors by any characters (including none at all) other than hyphens, in order to improve understanding and scannability.

Incorrect:

```
.demoimage {}
```

Correct:

```
.ad-sample {}
```

Alphabetize Declarations

Put declarations in alphabetical order in order to achieve consistent code in a way that is easy to remember and maintain.

Ignore vendor-specific prefixes for sorting purposes. However, multiple vendor-specific prefixes for a certain CSS property should be kept sorted as well (e.g., `-moz` prefix comes before `-webkit`).

(Exceptions prove the rule, so in the event of the cascade pushing order on us, that's fine.)

Correct:

```
background: fuchsia;
border: 1px solid;
-moz-border-radius: 4px;
-webkit-border-radius: 4px;
border-radius: 4px;
color: black;
text-align: center;
text-indent: 2em;
```

Indent All Block Content

Indent all block content—that is, rules within rules as well as declarations, so to reflect hierarchy and improve understanding.

Correct:

```
@media screen, projection {

    html {
          background: #fff;
          color: #444;
    }

}
```

Use a Semicolon After Every Declaration

End every declaration with a semicolon for consistency and extensibility reasons.

Incorrect:

```
.test {
    display: block;
    height: 100px
}
```

Correct:

```
.test {
    display: block;
    height: 100px;
}
```

Use a Space After a Property Name's Colon

Always use a single space between property and value (but no space between property and colon) for consistency reasons.

Incorrect:

```
h3 {
    font-weight:bold;
}
```

Correct:

```
h3 {
    font-weight: bold;
}
```

Use a Space Between the Last Selector and the Declaration Block

Always use a single space between the last selector and the opening brace that begins the declaration block. The opening brace should be on the same line as the last selector in a given rule.

Incorrect:

```
#video{
    margin-top: 1em;
}

#video
{
    margin-top: 1em;
}
```

Correct:

```
#video {
    margin-top: 1em;
}
```

Separate Selectors and Declarations by New Lines

Always start a new line for each selector and declaration.

Correct:

```
h1,
h2,
h3 {
    font-weight: normal;
    line-height: 1.2;
}
```

Separate Rules by New Lines

Always put a blank line (two line breaks) between rules.

Correct:

```
html {
    background: #fff;
}

body {
    margin: auto;
    width: 50%;
}
```

Use Single Quotation Marks for Attribute Selectors and Property Values

Use single (`''`) rather than double (`""`) quotation marks for attribute selectors or property values. Do not use quotation marks in URI values (`url()`).

Exception: If you do need to use the `@charset` rule, use double quotation marks, as single quotation marks are not permitted.

Correct:

```
@import url(//example.com/default.css);

html {
    font-family: 'helvetica neue', helvetica, sans-serif;
}
```

Summary

This has been a little, rather tiny, treatise on coding guidelines. Although short, it covered several key ideas:

Coding guidelines govern how we write code.

Coding guidelines directly help consistency, and through that, indirectly, impact usability, collaboration, and maintainability.

Coding guidelines are important.

The main ingredients of a coding guideline are: what (not) to do within a particular scope, examples, and an explanation.

Coding guidelines can deal with preference or with quality.

Coding guidelines can be descriptive, prescriptive, or both.

Coding guidelines must be communicated, enforced and reviewed.

And, there are some solid coding guidelines out there.

At Google we used to say, "the point of having style guidelines is to have a common vocabulary so people can concentrate on what you're saying rather than on how you're saying it." I hope that despite the brevity of this pamphlet, you, too, can now help your team concentrate on what you're saying, a little better.

About the Author

Jens Oliver Meiert is a German author, philosopher, adventurer, artist, and developer. He has written a few books and a few more articles, all of which appear on his website, *meiert.com*

Colophon

The cover image is by 掬茶 (Own work) [CC BY-SA 3.0 (*http://creativecommons.org/licenses/by-sa/3.0*)], via Wikimedia Commons.