HTML and CSS Tutorial

Chapter 1: Getting Started

Welcome to the world of HTML! If you have never written HTML code before, you should start with this section. If you've written HTML before, you should feel free to skip ahead to Chapter 2, *Markup Basics*.

When you're done with this chapter, you should be able to:

- create and alter a simple web page
- explain what HTML elements and attributes are
- use your browser to help you view and reuse code from other websites

Section Summaries

The Getting Started chapter contains these sections:

1. Diving In — Demonstrates basic HTML concepts by diving in and creating a simple web page.
2. Structure — Examines document structure: the `html` element, the `head`, the `title`, and the `body`.
3. Tinkering — Makes alterations to the "Simple Web Page". We'll add text, change the background color, and explore how HTML handles whitespace.
4. Elements — Explores elements, which are the basic building blocks of a web page. What are the components of a element? What happens if you misspell a element? And is there a good use for the `blink` element? The answer might surprise you!
5. Attributes — Explores attributes, which modify the behavior of a element. We'll look at the components of an attribute and talk about attribute whitespace, attribute misspellings, and other issues.

6. Browser Tools — Explains how to use key browser features, such as viewing source, viewing page properties, and saving images.

Diving In

The first thing we're going to do is dive in and create a simple webpage. We'll explore what's going on soon, but for now, just follow the directions as best you can.

To create your web page, you need:

- A browser (you're using one right now)
- A plain text editor (you already have one on your machine)

Step 1: Open a Text Editor

A text editor is a program that edits plain text files. When authoring HTML, do not use Word, Wordpad, Pages, or any other program that does not save files as plain text.

- Windows users: open Notepad (Start | Programs | Accessories | Notepad)
- Mac OS X users: open **vi** or TextEdit (Applications | TextEdit)
- Linux and other UNIX users: open **vi** or **emacs**

Unfortunately, Notepad and TextEdit are not very good text editors. By default, Notepad always tries to save and open files with a `.txt` extension. You must constantly fight with it to make it save and open webpages (files with a `.htm` or `.html` extension). Be strong, and ye shall overcome.

Likewise, TextEdit does not use plain text by default. Before using TextEdit to edit HTML, go into Preferences and set the Format for new documents to be Plain Text. For good measure, you might have to convert your existing document to plain text (Format | Make Plain Text).

Using Notepad to write HTML is kind of like using a butter knife to cut a birthday cake — it's the wrong tool for the job. A better choice would be a more advanced text editor, such as:

- Crimson Editor (Windows, free)
- Smultron (Mac OS X, free)
- TextMate (Mac OS X, not free, but well worth the money)
- jEdit (Cross-platform, free)

All of these editors avoid the problems with file extensions and plain text described above, and have numerous extra features for advanced users. You don't have to download one of these editors right now, but if you continue with web development, you'll need to upgrade your tools.

Step 2: Create and Save the Web Page

Copy-and-paste the text in Example 1.1, "A Simple Webpage" into your text editor, starting with the text, `"<html>"`.

Example 1.1. A Simple Webpage

view htmlview plainprint?

```
<html>
  <head>
```

```
    <title>A Simple Webpage</title>
  </head>
  <body>
    This is a simple webpage.
  </body>
</html>
```

To see what this HTML code sample looks like as a web page, click the view html link above the code sample.

The words that are surrounded with angle brackets (< >) are called elements. We will talk about what a element is soon, but first let's finish creating and displaying your webpage.

Once you have copied the text over to your text editor, save the file in your home directory or Desktop as the file `simple.html`.

Note

Notepad will try to append a `.txt` extension to your file name. Don't let this happen. In the Save As dialog box, set File name to `simple.html` and change Save as type from Text Documents to All Files (*.*).

Step 3: Display the Webpage in Your Browser

Open the file `simple.html` by typing **Ctrl-O** (**Cmd-O** for Mac users) and selecting the file. Internet Explorer users should select Browse to find the file.

After selecting `simple.html`, click Open. Your webpage should appear in the browser window. Compare it to Example 1.1. Does it match? (To compare results, click the view html link above Example 1.1.)

If it does, congratulations! Let's move on to the next section, where we'll try to answer the question, what the heck is going on?

Structure

The previous section demonstrates how to create a simple web page. If you haven't saved this example on your computer as the file `simple.html`, do so now.

Example 1.2. A Simple Webpage

view htmlview plainprint?

```
<html>
  <head>
    <title>A Simple Webpage</title>
  </head>
  <body>
    This is a simple webpage.
  </body>
</html>
```

If you view `simple.html` in your browser, you will see the words "This is a simple webpage" on a white or grey background. Where did everything else go? And what are those words with the angle brackets, anyway?

A Brief Introduction to Elements

The web page `simple.html` uses these elements: `html`, `head`, `title`, and `body`.

- Elements are delineated by angle brackets (< >).
- Elements are "invisible"; they don't directly appear in the web page. Instead, they serve as instructions to your browser. They can change your text's appearance, create a link, insert an image, and much more.
- An element starts with an opening tag (`<element>`) and ends with a closing tag (`</element>`). Some elements do not require closing tags.

We'll discuss the general properties of elements in some detail in Elements. For now, let's focus on the particular elements in the "Simple Webpage" example.

Structure of the Simple Webpage

Although the "Simple Webpage" doesn't look like much, its elements (`html`, `head`, `title`, and `body`) are fundamental to the structure of all HTML documents. Here's what these elements mean:

- `<html>`: "Here begins an HTML document."

  The `html` element helps identify a file as an HTML document.

- `<head>`: "Here begins the header."

  The header contains elements that apply to the overall document. For example, it might contain elements that are designed for search engines to process or elements that change the overall appearance of the webpage. However, elements in the header do not display directly as normal webpage content.

  The reasons you would be concerned about the header are a bit abstract at this stage, so we won't worry about it much until later.

- `<title>`: "Here begins the document title." (Must be in the header)

  If you view the file `simple.html` in your browser, along the top of your browser window you should see the words, "A Simple Webpage". These words appear because of the `title` element.

  As an exercise, change the title of the `simple.html` webpage to, "My First Webpage". Save your changes and view them by clicking the browser's **Refresh** or **Reload** button.

  Titles might not seem important at first glance, but they're actually quite useful. For example, if a search engine displays your page in a list of search results, it will probably display the `title` as your page's title. If you omit the `title` element, the search engine will make up one for you. This is Not a Good Thing.

  Note

  You might have noticed that the `title` element is contained within the `head` element. Is this kosher? Absolutely! In fact, many elements are designed to contain other elements, and you will be nesting elements within other elements quite frequently as you continue.

- `<body>`: "Here begins the body."

The body is where you put text and elements to be displayed in the main browser window. The reason that the words “This is a simple webpage” appear when you display `simple.html` is becaused you placed them in the `body`.

So why do we only see “This is a simple webpage” when we display `simple.html` in a browser? The answer is, after you remove all the elements that are not designed to display in the browser window, the sentence “This is a simple webpage” is the only thing left.

In the next section, we’ll tinker with our example webpage, just to see what happens. After that, we’ll provide a more formal definition of elements and element properties.

Tinkering

Let’s make a few alterations to our “Simple Webpage” from “Diving In”.

Example 1.3. Adding Text

view htmlview plainprint?

```
<html>
<head>
  <title>A Simple Webpage</title>
</head>
<body>
  This is a simple webpage.
  And I mean really simple.
</body>
</html>
```

To view this alteration:

1. Add the text in bold to your `simple.html` file.
2. Save the file as `adding.html`.
3. Open the `adding.html` file in your browser.

HTML and Whitespace

What happened here? Why are the two sentences on the same line?

HTML ignores whitespace (spaces, tabs, or carriage returns) in the source. It replaces all continuous chunks of whitespace with a single space: ” “. Although it might take you a while to get used to this behavior, the advantage is that that you can use whitespace to make your code more readable, and the browser can freely adjust line breaks for each user’s window size and font size.

Example 1.4, “Scattered Text” displays in the browser identically to Example 1.3, “Adding Text”. Try it out.

Example 1.4. Scattered Text

view htmlview plainprint?

```
<html>
<head>
 <title>A Simple Webpage</title>
</head>
```

```
<body>
  This is a
  simple webpage.
  And I
  mean

  really simple.
</body>
</html>
```

Note

For brevity, subsequent examples will leave out the `html`, `head`, `title`, and `body` elements, since we're just changing the `body` content. If the `head` content becomes important again, we'll add it back.

Of course, HTML provides many ways to control whitespace. For a couple of simple examples, refer to "Paragraph Breaks".

Emphasizing Text

Now change the text of your simple web page to:

Example 1.5. Emphasized Text

view htmlview plainprint?

```
This is a simple webpage.
And I mean <em>really</em> simple.
```

Save your changes and view the file in your browser.

As you can see, the text becomes italic, starting at the `<em>` and ending at the `</em>`. The `em` element represents emphasized text — text that you would stress when speaking. The `em` element is our first example of an element that can affect the physical appearance of the text.

For more examples of elements that can affect the appearance of text, refer to "Font Styles".

Changing the Background Color

Let's do something a little more dramatic:

Example 1.6. Background Color

view htmlview plainprint?

```
<html>
<head>
  <title>A Simple Webpage</title>
</head>
<body style="background: yellow">
  This is a simple webpage.
  And I mean <em>really</em> simple.
</body>
</html>
```

A bit garish… but things could be worse. Try `background: red`, for example.

This is our first example of an attribute, a component of an element that modifies that element's behavior. For some elements, like the `body` element, attributes are optional. For other elements, attributes are essential. We'll discuss attributes at some length later, in "Attributes".

Try some other color words: `blue`, `teal`, `limegreen`, `papayawhip`… What works? What doesn't? (If you're interested, feel free to skip ahead to the section called "Colors".)

Elements

After all this experimenting with our example webpage, it's time to define elements more formally. Elements have three main components:

view htmlview plainprint?

```
1. <name attribute1="value1" attribute2="value2"...>
2.   ...(text or more elements go here)...
3. </name>
```

- *`name`*: Identifies the type of the element, such as `body` or `em` element. The name of an element can be upper case, lower case, or anything in between. Upper case makes it easier to differentiate elements and text when examining HTML source, while lower case is a more modern style and is easier to type. Either way, it's a matter of taste: `<BODY>`, `<body>`, and even `<bODy>` are all the same element.
- *`attribute`*: Changes the behavior of an element. For example, the `style` attribute enables you to change the appearance of an element and its content.
- *`value`*: Enables particular attributes to vary. For example, `background: yellow`, when present in an element's `style` attribute, changes the background of the element to yellow.

Some elements do not require a closing element to work properly. For example, the `br` element creates a line break, and should not enclose anything. We'll discuss `br` further in "Paragraph Breaks".

Some elements do not require any attributes. In "Example 1.6, Background Color", the `body` element had a `style` attribute, but the `body` element works just fine without without it. By contrast, the a element, which creates a hyperlink, has an `href` attribute that defines the link's destination. If you don't include the `href` attribute, the link won't point anywhere.

Misspellings and Misunderstandings

Vastly simplified, a browser processes an HTML page something like this:

1. Identify anything that is between angle brackets (< >) as a "element"
2. Determine whether the browser has any instructions for using the element:
    - If yes, the element is real — follow the instructions
    - If no, the element is garbage — ignore it

When a browser ignores a element, that means one of two things:

- The element is misspelled or does not exist. For example, `mxyzptlk`, for example, is not an HTML element.

- The element means something, but that particular browser does not support it. For example, the `blink` element works in Firefox, but not in Internet Explorer or Safari.

Example 1.7. Blinking Text

view htmlview plainprint?

1. Warning: <blink>For external use only</blink>.
2. Avoid contact with eyes.

If you are using Firefox, the text in the “view html” box should be blinking. If you are using Internet Explorer or Safari, the text will not be blinking. This is actually a common situation when composing web pages — one browser won’t support a particular feature, or supports it incorrectly.

Caution

Most people find the `blink` element distracting and irritating, and in fact `blink` is not part of the official HTML standard. In fact, I am aware of only one example of a good use of `blink` on the entire Web. Well, maybe two.

A Digression: What’s a “Tag”?

You’ll often hear people refer to “tags,” as in, “The markup tags tell the Web browser how to display the page.” Almost always, they really meant to say “elements.” Tags are not elements, they *define the boundaries of an element*. The `p` element begins with a `<p>` open tag and ends with a `</p>` closing tag, but it is not a tag itself.

- Incorrect: “You can make a new HTML paragraph with a `<p>` tag!”
- Correct: “It’s a good idea to close that open `<p>` tag.”

Sometimes you’ll hear people say “`alt` tag,” which is even worse. An “`alt tag`” is really an `alt` *attribute*. This important attribute provides alternative text for images, in case the user can’t see the image for some other reason. We’ll talk more about this attribute later.

The element vs. tag confusion is sort of understandable: it’s a common mistake even among professionals, and they both look like angle-brackety things, after all. But attributes are not tags, not even close. If you hear someone say “alt tag,” this is a key indication that the speaker does not understand HTML very well. (You probably shouldn’t invite them to your next birthday party.)

Attributes

Attributes modify the behavior of a element. Attributes allow you to change sizes, colors, specify link locations, and much more. They have two main components:

view htmlview plainprint?

1. <name attribute1="value1" attribute2="value2" ...>
2.   ...(text or more elements go here)...
3. </name>

- *`attribute`*: Defines the attribute that you’re using. As with elements, case is irrelevant for the attribute name. `STYLE`, `style`, and `sTyLE` all do the same thing.
- *`value`*: Selects a particular kind of behavior for the attribute. Case sometimes matters for the values inside of the attribute, so be careful here.

"Changing the Background Color" includes an example of using the `style` attribute to control the background color of the entire webpage. Here's a second example, the `href` attribute:

Example 1.8. A Link to The New York Times

view htmlview plainprint?

```
<a href="http://www.nytimes.com">go to nytimes.com!</a>
```

The `a` element (or, "anchor" element) creates a link to another web page. The `href` attribute specifies the location, which we set to the value `http://www.nytimes.com`. If you view the results and select the link "go to nytimes.com!", you will indeed go to the New York Times.

Note

The `style` and `href` attributes are very different beasts. The `style` attribute is optional, and may be applied to nearly any HTML element in the `body`. The `href` attribute is essential for specifying a link location, and only applies to a narrow set of elements.

Attribute Properties

Attributes follow these rules:

- Elements may have multiple attributes. The order of the attributes is irrelevant.
- Each element has a list of permitted attributes. Some attributes may be assigned to most (or even all) elements. But in general, you cannot freely add any attribute to any element.
- Whitespace between attributes is irrelevant. You may freely insert spaces, tabs, or even carriage returns between attributes, as long as you don't forget the closing angle bracket at the end of the element (>). For example:

  view htmlview plainprint?

  ```
  <a href="http://www.gutenberg.org"
    style="background: yellow"
    >Books in the public domain</a>
  ```

  is perfectly acceptable. For elements with many attributes, adding whitespace can sometimes make your code more readable.
- Although upper case and lower case is irrelevant for attribute names, the case can be important for attribute *values*. For example,

  view htmlview plainprint?

  ```
  <a href="http://www.cnn.com/index.html">Go to CNN</a>
  ```

  creates a link that takes you to the CNN.com home page, while

  view htmlview plainprint?

  ```
  <a href="HTTP://WWW.CNN.COM/INDEX.HTML">Go to CNN</a>
  ```

  is a broken link.
- It is good practice to quote attribute values, although often this is optional. That is, `href=http://www.yahoo.com` is the same as `href="http://www.yahoo.com"`. If the attribute value contains a special character such as a space or an angle bracket (< >), you *must* use quotes, or the browser will interpret the special character as the end of the attribute or element.

Caution

Take care to close any open quotes you start. Otherwise, your attribute value will “continue” until the next quote mark, which will badly garble your page.

- As with elements, browsers attempt to ignore incorrect attributes. Misspelled attributes and attribute values, such as `hhref` and `backgrounde: yellow`, have no effect. Misspelling `background` means that your background won’t turn yellow. Misspelling `href` means that your link won’t function. However, if you misspell `http://www.yahoo.com`, your link will function, but it won’t take you to the right place.

Browser Tools

A browser’s primary function is to display web pages. However, for a web developer, browsers are much more useful than that. Whenever you visit a web page, your browser fetches a tremendous amount of valuable information, such as the HTML source of the page, image files that appear on the page, and other associated files. This section describes how to use your browser to retrieve that information.

Note

Since browser makers love to change the names and locations of these functions from version to version, the instructions below might not apply if you’re using a different browser version than the one tested at the time this section was written.

View Source

By far the most valuable browser tool is View Source, which displays the HTML source of the page. Viewing source is one of the fastest ways to learn how to code. If you’re not sure how some feature of HTML works, you can view the source of a site that uses the feature, copy and paste sections of code, make alterations, and see what happens. To view source:

- Internet Explorer, Firefox: Right-click on the window and select View Source.
- Safari: **Ctrl**-click on the window and select View Source.
- Opera: Right-click on the window and select Source.

Figure 1.1. View Source (Firefox)

Although viewing the source is incredibly useful, keep in mind these two caveats:

- It’s okay to copy and alter a site’s source for learning purposes on your home machine, but don’t post any such material on your public website. This goes for both the text on the site and the underlying code or design.
- Most sites on the web have *terrible* underlying code. Learning how to code by randomly viewing websites is like learning how to compose verse by randomly reading poetry written by kids. You might encounter some brilliant stuff… but the odds are pretty low. Never assume that just because `BigFamousWebsite.com` coded their home page a certain way, that this was the best way to do it.

Save Images

Another invaluable browser function is the ability to save image files. Every time your browser visits a web page, it also downloads all the images that display on the page. You can save these files and reuse them for educational purposes. To save an image:

- Internet Explorer: Right-click on the window and select Save Picture As…
- Firefox: Right-click on the image and select Save Image As… to save the image as a file, View Image to view the image by itself in the browser, or Properties to view the image's file size and other properties.
- Safari: **Ctrl**-click on the image and select Save Image to the Desktop to save the image as a file or Open Image in New Window to view the image by itself in the browser.
- Opera: Right-click on the image and select Save Image… to save the image as a file, Open Image to view the image by itself in the browser, or Image properties to view the image's file size and other properties.

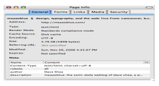
As with View Source, you should never take someone else's images, altered or unaltered, and post them on a public site unless you have permission. There are numerous free image repositories all over the web, where you can get clip art, backgrounds, and photos that are either in the public domain or offered under reasonably liberal licenses.

View Page Info

The View Page Info function provides secondary technical information about the page, such as the page size in bytes and the text encoding. This information might not seem all that useful when you're first starting out, but it's invaluable for more advanced coders. To view page info:

- Internet Explorer: Right-click on the window and select Properties.
- Firefox: Right-click on the window and select View Page Info.
- Opera: Select Tools | Appearance | Panels, check the Info checkbox, and click OK.

Figure 1.2. View Page Info (Firefox)



The information displayed varies from browser to browser. Firefox provides more useful information than Internet Explorer, including a full list of all images on the page (with height, width, and size in bytes). Opera takes an interesting approach: if you activate the Info panel, you can view this information alongside each page you browse to. Opera also provides a shortcut (**Ctrl-J**) for displaying all the links on the page.

Webdev Toolbars

Several browsers offer some sort of "webdev toolbar": a downloadable set of menus and options for your browser that are specifically designed to help understand the structure of a webpage. Toolbars can offer a dizzying array of useful functions, but as with View Page Info, they are probably more useful for experienced web developers.

Toolbars

- Internet Explorer

- Firefox
- Opera

Chapter 2: Markup Basics

This chapter, covers some of the basic HTML elements and attributes in a systematic way. Each section displays one or more short webpages, which we will alter several times to demonstrate how various elements work. Before proceeding, you should be comfortable with the idea of saving HTML to a file, altering it with a plain text editor, and viewing the results in your browser. If not, please read Chapter 1, *Getting Started* and work through the examples until you are comfortable with these concepts.

When you're done with this section, you should be able to:

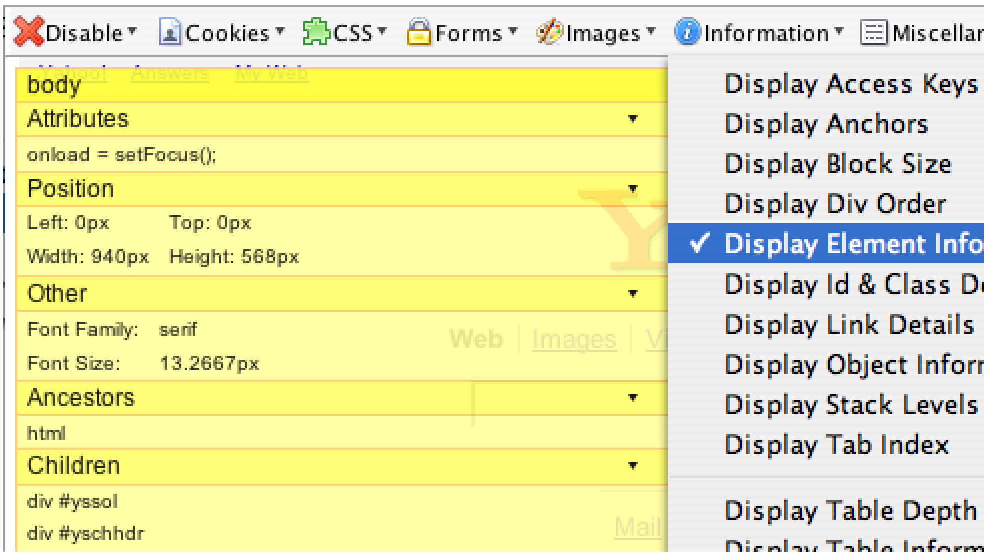
- create a document with headings, paragraphs, and line breaks
- change the text color and background color
- use physical styles and logical styles to change the appearance of your text
- display images and create links to other web pages
- understand more complex HTML elements that require nesting, such as lists

Section Summaries

The Markup Basics chapter contains these sections:

1. Paragraph Breaks — Describes the paragraph element and the line break element, including a discussion of correct and incorrect usage.
2. Headings — Describes the heading elements. In this section, we'll discover why it's really important to close your open tags properly.
3. Font Styles — Explores elements that help control the font. We'll touch on the difference between inline and block elements, and compare physical style elements versus logical style elements.
4. Colors — Explains how to change the background and foreground color of an entire page, a paragraph, or even a single word. We'll see examples of color names and hex codes, and we'll discuss which method is preferable.
5. Links — Breaks down a URL into its constituent parts, explains relative links versus absolute links, and provides examples of how to use IDs to link within pages.
6. Images — Describes what images are, how to embed them in a web page, how to stretch and squash them, and how to "float" them with respect to the other content on the page.
7. Lists — Describes ordered lists, unordered lists, and definition lists. This is our first example of elements that require nesting to function properly.

Figure 1.3. Webdev Toolbar (Firefox)



body
Attributes
onload = setFocus();
Position
Left: 0px Top: 0px
Width: 940px Height: 568px
Other
Font Family: serif
Font Size: 13.2667px
Ancestors
html
Children
div #yssol
div #yschhdr

Paragraph Breaks

In Chapter 1, *Getting Started*, we covered some basic HTML concepts, including the fact that HTML ignores whitespace. This seems a little crazy, doesn't it? How can we do we compose a readable webpage without everything collapsing into a single giant glob of text?

Let's say we want our website to display some lines from an obscure playwright:

Example 2.1. Unformatted Text

view htmlview plainprint?

```
<html>
  <head>
    <title>Macbeth: 1, Reckless Youngsters: 0</title>
  </head>
  <body>
    MACBETH: My name's Macbeth.

    YOUNG SIWARD: The devil himself could not pronounce a title
    More hateful to mine ear.

    MACBETH: No, nor more fearful.

    YOUNG SIWARD: Thou liest, abhorred tyrant; with my sword
    I'll prove the lie thou speak'st.

    [They fight, and young Seward is slain.]

    MACBETH: Thou wast born of woman.--
    But swords I smile at, weapons laugh to scorn,
    Brandish'd by man that's of a woman born.
  </body>
</html>
```

Now, this is a perfectly acceptable web page. The `html`, `head`, `title`, and `body` elements are all in the right places. In fact, this example is just like the “Simple Web Page” from Diving In, except that the text is a little different.

There is one small problem, however. HTML collapses whitespace, which means that all our text will collapse into a single paragraph and look something like this:

> MACBETH: My name’s Macbeth. YOUNG SIWARD: The devil himself could not pronounce a title More hateful to mine ear. MACBETH: No, nor more fearful. YOUNG SIWARD: Thou liest, abhorred tyrant; with my sword I’ll prove the lie thou speak’st. [They fight, and young Seward is slain.] MACBETH: Thou wast born of woman.– But swords I smile at, weapons laugh to scorn, Brandish’d by man that’s of a woman born.

This is terrible! Let’s try to make this page more readable.

The `p` Element

The `p` element breaks text into paragraphs. Any text that you surround with a `<p>` and a closing `</p>` becomes a separate block. Let’s see how this works. For brevity, we will start to leave off the structural elements: `html`, `head`, and `body`.

Example 2.2. Paragraphs

view htmlview plainprint?

```
<p>
  MACBETH: My name's Macbeth.
</p>
<p>
  YOUNG SIWARD: The devil himself could not pronounce a title
  More hateful to mine ear.
</p>
<p>
  MACBETH: No, nor more fearful.
</p>
<p>
  YOUNG SIWARD: Thou liest, abhorred tyrant; with my sword
  I'll prove the lie thou speak'st.
</p>
<p>
  [They fight, and young Seward is slain.]
</p>
<p>
  MACBETH: Thou wast born of woman.--
  But swords I smile at, weapons laugh to scorn,
  Brandish'd by man that's of a woman born.
</p>
```

Each block is followed by two line breaks. This is a good start!

Each paragraph is surrounded by an opening tag, `<p>`, and a closing tag, `</p>`. The `p` element is a block element; it “does something” to a block of text. The browser must identify where the block ends, so the `<p>` open tag needs a closing tag, `</p>`.

Unfortunately, many people take the `p` element to mean “make a new paragraph here” or “put two line breaks here,” rather than “enclose this text and make it a paragraph.” And thus, you see a lot of code that looks like this:

Example 2.3. Incorrect `p` Usage

view htmlview plainprint?

```
Text Block 1: Blah blah blah... <p>

Text Block 2: Blah blah blah... <p>

Text Block 3: Blah blah blah...
```

What happens? (Try it.) Text Block 2 and 3 are paragraphs (the browser “inserts” a closing </p> element right before each <p> element). Text Block 1 is not actually a HTML paragraph. However, the page displays with two line breaks between each text block, which appears to result in the same thing as doing it the proper way.

So what’s the big deal? Why not do it the “improper” way, and avoid typing all those `</p>` closing tags? Well, later we’ll talk about how to use a technique called Cascading Style Sheets (CSS) to control the appearance of all elements of a particular type. For example, you can use CSS to set all your paragraphs to be red 12pt Times New Roman. But if you’re using `p` incorrectly, you’ll be in for an unpleasant surprise, as many of your paragraphs (or rather, what you *thought* were paragraphs) won’t get the proper style applied.

The `br` element

We still need to break up the dialogue into lines, as they appear in the original play. Now, we could do this with the `p` element, but then all lines would be equally spaced. and so it wouldn’t be clear which line belongs with which speaker. How do we get finer control… say, one line break?

Example 2.4. Line Breaks

view htmlview plainprint?

```
<p>
  MACBETH: My name's Macbeth.
</p>
<p>
  YOUNG SIWARD: The devil himself could not pronounce a title<br>
  More hateful to mine ear.
</p>
<p>
  MACBETH: No, nor more fearful.
</p>
<p>
  YOUNG SIWARD: Thou liest, abhorred tyrant; with my sword<br>
  I'll prove the lie thou speak'st.
</p>
<p>
  [They fight, and young Seward is slain.]
</p>
<p>
  MACBETH: Thou wast born of woman.-- <br>
```

```
But swords I smile at, weapons laugh to scorn,<br>
Brandish'd by man that's of a woman born.
</p>
```

Unlike the `p` element, the `br` element does not enclose any other HTML. There is no such thing as a closing `</br>` element. Also note that we don’t have to put a final `br` element on the last line of each dialogue exchange. The <p> element will take care of the last line break for us. So now we’ve separated each speaker with a new line.

How does this look? (Try it.) Now each line of dialogue is on its own line on the page. The lines are considerably easier to read.

As with the `p` element, many web designers use `br` incorrectly. You’ll often see code that looks like this:

Example 2.5. Incorrect `br` Usage

view htmlview plainprint?

```
Text Block 1: Blah blah blah...
<br><br>
Text Block 2: Blah blah blah...
<br><br>
Text Block 3: Blah blah blah...
```

This code is even worse than the code in Example 2.3, “Incorrect `p` Usage”. *None* of the text blocks are HTML paragraphs, which makes them difficult to style. If you want paragraphs, use `p`, not `br`.

In fact, there aren’t that many places where you want to force a line break with `br`. The most common cases are:

- Song lyrics
- Lines of poetry
- Lines of dialogue

Aside from that, there are more elegant ways to control line spacing, such as using a `p` element and adjusting the spacing using CSS. But that’s a later lesson.

Headings

So far we’ve covered paragraphs and line breaks… but in a long document, we probably want more structure than that. How do we create section headings?

The `h` Elements

There are six levels of heading elements: `h1`, `h2`, `h3`, `h4`, `h5`, and `h6`. These block elements tell the browser to set off the heading from the rest of the text. The `h1` heading level is the most important heading, while the `h6` level is the least important.

Most browsers display headings in bold, with `h1` the largest and `h6` the smallest.

Example 2.6. Headings

view htmlview plainprint?

```
<h1>Heading 1</h1>
```

```

<h2>Heading 2</h2>

<h3>Heading 3</h3>

<h4>Heading 4</h4>

<h5>Heading 5</h5>

<h6>Heading 6</h6>
```

Like the paragraph element `p`, the heading elements are block elements. Each matched set of heading elements creates a new block, automatically separated from other blocks with line breaks.

Mangled Headings

In “Paragraph Breaks”, we discussed the consequences of failing to close your open `<p>` tags. This is even more important for heading elements.

Example 2.7. Mangled Heading and Text

view htmlview plainprint?

```
<h1>STAVE I: MARLEY'S GHOST

MARLEY was dead: to begin with. There is no doubt
whatever about that. The register of his burial was
signed by the clergyman, the clerk, the undertaker,
and the chief mourner. Scrooge signed it: and
Scrooge's name was good upon 'Change, for anything he
chose to put his hand to. Old Marley was as dead as a
door-nail.
```

Whoops! If you view the resulting web page, you'll see that the `h1` element’s oversized, bold font has spilled over onto the rest of the text. The browser has no indication where the `h1` element is supposed to end, and so it interprets the `h1` element as enclosing everything else on the page.

Let’s fix this by closing the open `<h1>` tag. We’ll put the text in its own HTML paragraph for good measure:

Example 2.8. Proper Heading and Text

view htmlview plainprint?

```
<h1>STAVE I: MARLEY'S GHOST</h1>
<p>
  MARLEY was dead: to begin with. There is no doubt
  whatever about that. The register of his burial was
  signed by the clergyman, the clerk, the undertaker,
  and the chief mourner. Scrooge signed it: and
  Scrooge's name was good upon 'Change, for anything he
  chose to put his hand to. Old Marley was as dead as a
  door-nail.
</p>
```

The heading and paragraph are now properly separated.

Font Styles

The style elements provide a straightforward way to control the font. These elements fall into two categories: physical styles and logical styles. The style elements provide only crude control over the appearance of your text. For more sophisticated techniques, refer to “Font Control”.

Logical Styles

A logical style indicates that the enclosed text has some sort of special meaning. We’ve already seen the `em` element, which indicates that the enclosed text should be emphasized (stressed when spoken). The list of available logical style elements includes:

- `em` — Emphasizes text. Usually rendered as italic.

  view htmlview plainprint?

  ```
  Please do <em>not</em> feed the monkeys.
  ```
- `strong` — Strongly emphasizes text. Usually rendered as bold.

  view htmlview plainprint?

  ```
  <strong>WARNING:</strong> Never feed the monkeys
  under any circumstances.
  ```
- `cite` — Indicates a citation or reference. Usually rendered as italic.

  view htmlview plainprint?

  ```
  For more information, please refer to
  <cite>The Dangers of Monkey Feeding, Vol 2</cite>.
  ```
- `dfn` — Indicates the defining instance of a term; usually applied when the term first appears. Usually rendered as italic.

  view htmlview plainprint?

  ```
  Monkeys have sharp <dfn>canines</dfn>, sharp pointy
  teeth to bite you with.
  ```
- `abbr` — indicates an abbreviation or acronym, such as RADAR (RAdio Detection And Ranging); also provides a `title` attribute that may contain the fully-spelled out version. Hovering your mouse over the `abbr`>> causes many browsers to display a “tooltip” with the contents of the `title` attribute. Rendering is inconsistent; some browsers display a dotted underline, while others do nothing special.

  view htmlview plainprint?

  ```
  In particular, beware of
  <acronym title="Monkeys Of Unusual Size">MOUS</acronym>es.
  ```
- `code` — Indicates computer code fragments and commands. Usually rendered in a monospaced font.

  view htmlview plainprint?

  ```
  <code>10 PRINT "I LOVE MONKEY CHOW"<br>
  20 GOTO 10</code>
  ```

Wait — “*Usually* rendered as italic?” What does that mean? Shouldn’t `em`, `cite`, and `dfn` *always* render as italic?

Well, not necessarily.

- By default, most browsers render `em` as italic. However, this is only a convention. Nothing requires browsers to use italic, and in fact, some browsers (such as an text-to-speech synthesis browser) might be completely unable to use italic.

- Although the default is italic, you can override this using CSS. For example, you could specify that on your website, all `em` elements render as red and bold.

Okay… but why do `em`, `dfn`, and `cite` all render the same by default? If I want italic, why wouldn't you just use `em` and forget about the rest?

Well, sure, you could do that. However, using a richer set of elements gives you finer control. For example, you could declare that emphasized text is red and bold, but all citations are green and italic. You can also use logical style elements to extract more meaning out of a website. For example, if you knew that a website uses the `cite` element consistently, you could easily write a program to extract a list of citations. (But don't obsess over that point; there are better ways to store and consume this sort of information.)

The key point to remember is that a `cite` element is a *citation*, not a chunk of italic text. The italics are just a useful side effect.

Inline vs. Block Elements

Unlike the paragraph and header elements, the style elements listed above don't mark off a "block" of text. The physical style elements are inline elements that perform their work without adding extra line breaks:

Example 2.9. Inline vs. Block Elements

view htmlview plainprint?

```
<p>
1. This is a paragraph with a section of
<em>emphasized text</em> inside of it.
</p>

<em>
2. This is a section of emphasized text with
<p>a paragraph</p> inside of it.
</em>
```

The first sentence results in one "block" with a couple of emphasized words inside. In the second sentence, the `p` element breaks the text up into multiple blocks.

Physical Styles

Physical style elements specify a particular font change. For example, to make text bold, you can mark it off with the `b` element: `<b>bold text</b>`. The list of available physical style elements includes:

- `b` — Makes text bold. Appropriate for product names, …

  view htmlview plainprint?

  ```
  Today, UniStellarDefenseCorp is proud to announce
  <b>Neutrinozon</b>, the only neutrino-based death ray
  on the market.
  ```

- `i` — Makes text italic. Appropriate for ship names, internal monologues and thoughts, …

  view htmlview plainprint?

  ```
  This exciting new product has already been installed on many
  advanced warships, including the <i>I.S.S. Hood</i>.
  ```

- `sub` — Makes text a subscript. Appropriate for scientific and mathematical notation, …

  view htmlview plainprint?
  ```
  Although the standard electron neutrino (μ<sub>e</sub>)
  configuration packs plenty of punch, muon neutrino
  (μ<sub>v</sub>) upgrades are available on request.
  ```
- `sup` — Makes text a superscript. Appropriate for footnotes, scientific and mathematical notation, …

  view htmlview plainprint?
  ```
  With an intensity of 1.76x10<sup>6</sup>
  cm<sup>-2</sup>s<sup>-1</sup>, nothing can repel firepower
  of this magnitude!
  ```

The physical styles are subtly different from the logical styles. The logical style element `strong` means "something strongly emphasized," while the physical style element `b` just means, "something that is bold."

A Digression: Physical Styles and Semantic Markup

These days, you'll sometimes hear people claim that the physical styles are yucky and bad and should never be used. This is because logical styles contain small quantities of a rare earth named "Semanticism". Semanticism can be mined and processed into the power source for farms of spinning Academic Paper Turbines, which serve to feed and clothe members of our society who would otherwise starve to death.

Although it is true that certain physical styles are obsolete, the `i`, `b`, `sub`, and `sup` elements *are* appropriate to use in certain situations. For example, compare these code samples:

- view htmlview plainprint?
  ```
  My grandfather served on the <i>U.S.S. Maine</i>.
  ```
- view htmlview plainprint?
  ```
  My grandfather served on the <em>U.S.S. Maine</em>.
  ```

In this case, `i` is more appropriate than `em`, unless you think it's appropriate to always be shouting the "U.S.S. Maine" part. Not that `i` is all that wonderful, but `em` is just flatly wrong. Maybe it would be nice if we had a `vessel` element, but HTML is a small language, and the `i` element is the best we can do.

For a more extreme example, consider the quantity "2 to the x power," represented in HTML as `2<sup>x</sup>`. If we take away the superscript, this leaves us with `2x`, which is most emphatically not equal to "2 to the x power." Even though the `sup` element literally means, "move the 2 above the line", this physical change to the text has actual mathematical meaning! (Thanks to Jacques Distler for pointing this one out.)

Colors

"Changing the Background Color" briefly introduced how to change the background color to yellow. That's kind of neat, but this just raises more questions. If yellow is available, what are the other colors I can use? Can I change the foreground color? Can I apply colors to just a small selection of text?

Text (Foreground) Color

Here's how to change the text color:

1. Add an attribute called `style` to the element. (`<element style="">`)
2. Inside the `style` attribute, specify the color you want by typing `color:` followed by the color you want. (`<element style="color: red">`)

Note

As with all attributes, do not include the `style` attribute in the closing tag, `</p>`.

For example:

Example 2.10. Red Paragraph

view htmlview plainprint?

```
<p>
  The grandmother lived out in the wood, half a league from the village,
  and just as Little Red-Cap entered the wood, a wolf met her. Red-Cap
  did not know what a wicked creature he was, and was not at all afraid
  of him.
</p>
<p style="color: red">
  'Good day, Little Red-Cap,' said he.
</p>
<p>
  'Thank you kindly, wolf.'
</p>
```

In the earlier example, Example 1.6, "Background Color", the background of the entire page turned yellow. But in our example above, the text turns red. A `style` attribute value of `background:` sets the background color, while `color:` sets the foreground color.

Note

The keywords `color` and `background` are called CSS properties. CSS (Cascading Style Sheets) is a technique for applying style to elements. Changing the element's color is just one small example of what CSS can do. We'll learn a lot more about CSS in Chapter 3, *Styling Basics*, but for now let's focus on color specifically.

Background Color

To change the background color of a element, you just need to use a different property in the `style` attribute: the `background` property. If you apply the background to the `body` element, this changes the background color of the entire web page. (We saw an example of this in "Tinkering".) Not only can you change an element's background color, you can change the text color at the same time:

Example 2.11. Text and Background Color

view htmlview plainprint?

```
<p style="background: yellow; color: purple">
```

```
  There's a yellow rose in Texas, That I am going to see,<br>
  Nobody else could miss her, Not half as much as me.<br>
  She cried so when I left her, It like to broke my heart,<br>
  And if I ever find her, We nevermore will part. <br>
</p>
```

The paragraph has two properties, `background` and `color`. When you apply multiple properties, you can add them in any order, but you must separate them with a semicolon. If you forget the semicolon:

view htmlview plainprint?

```
<p style="background: yellow color: purple">
  There's a yellow rose in Texas, That I am going to see,<br>
  Nobody else could miss her, Not half as much as me.<br>
  She cried so when I left her, It like to broke my heart,<br>
  And if I ever find her, We nevermore will part. <br>
</p>
```

then the browser interprets this as setting the background color to the value `yellow color: purple`. Since `yellow color: purple` is not a valid color, the browser does nothing (and you might be left puzzling over why your styles aren't appearing).

The results, Example 2.11, "Text and Background Color" are particularly garish and difficult to read. This brings up an important point: it is very easy to go wrong when choosing text and background colors. Use the color and background properties with caution.

Inline Color Styles

Although so far we've only applied the color properties to block elements such as `body` and `p`, you may apply the color properties to inline elements as well, including the elements from "Font Styles". For example,

view htmlview plainprint?

```
<p>
  <strong style="color: red">DANGER:</strong> Flying monkeys.
</p>
```

The red color applies to the `strong` element, but not to the rest of the paragraph.

If you just want to change the color of a selection of text without also making it bold or italic, there is a special inline element called the `span` element. The `span` element "does nothing" by default, and so it is safe for applying styles without affecting anything else. We'll discuss this element further in the section "Div and Span".

Note

Although you may apply the `style` attribute to a wide array of elements, there are some exceptions. For example, you might think you can change the color of the title in your browser's titlebar by adding a `style` attribute to the `title` element. However, this is invalid HTML, and the browser ignores it.

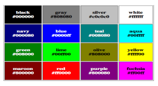
In general, you may apply style to nearly any element that falls under `body`, including `body` itself.

Specifying Colors

In the previous examples, we were using color keywords such as `red`, `green`, and so on. What other words are available? The answer it, it depends on your browser.

Here are sixteen color keywords that are nearly universal across all browsers that support color:

Figure 2.1. Standard Color Keywords



black #000000	gray #808080	silver #c0c0c0	white #ffffff
navy #000080	blue #0000ff	teal #008080	aqua #00ffff
green #008000	lime #00ff00	olive #808000	yellow #ffff00
maroon #800000	red #ff0000	purple #800080	fuchsia #ff00ff

The numbers after the pound sign (#) are called hex codes. Hex codes are another way to represent colors in HTML. It gets a little technical after this, so hang on…

Note

Those of you who grew up in the 1960s and experienced the "New Math" might have a small advantage here. See, I bet you thought you'd never use that stuff again!

Hex Codes

Each of the six numbers in a hex code is a hexadecimal, or base-16 number.

- In the ordinary decimal system, each numeral (0-9) represents a power of ten. You need ten symbols to represent decimal numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, and last but not least, 9.

  With two decimal numerals, the highest number you can make is "99". "9" in the tens place + "9" in the ones place = (nine x ten) + (nine x one) = 90 + 9 = 99.

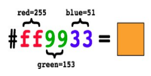
- In a hexadecimal system, each numeral represents a power of sixteen, and so you need sixteen symbols (0-9 and a, b, c, d, e, f). Here, a = 10 (decimal), b = 11 (decimal), and so on up to f = 15 (decimal).

  With two hexadecimal numerals, the highest number you can make is "ff", which is 255 in decimal notation. "f" in the sixteens place + "f" in the ones place = (fifteen x sixteen) + (fifteen x one) = 240 + 15 = 255.

A color hex code consists of three two-digit hexadecimal numbers. Each number indicates how much red, green, and blue the browser should display. The first hex number indicates how much red to use, the second indicates the amount of green, and the third specifies the blue: RRGGBB. `00` means "use none of this color", while `ff` means "use as much of this color as possible."

Figure 2.2, "Hex Code #ff9933" demonstrates how this works. The hex code #ff9933 provides the maximum amount of red, a middling amount of green, and a small amount of blue. Mixing the three together results in an orange-ish color.

Figure 2.2. Hex Code #ff9933



Here are some more examples:

- `color: #ff0000` = All red, no green, no blue. Result: bright red.
- `color: #00ffff` = No red, all green, all blue. Result: bright teal (or "aqua").
- `color: #000000` = No red, no green, no blue. Result: black.

- `color: #ffffff` = All red, all green, all blue. Result: white.
- `color: #cc99ff` = A lot of red, some green, all blue. Result: lavender.

Hex codes can be uppercase or lowercase. Just don’t forget the pound sign at the beginning. There are actually several more ways to specify colors, but color names and hex codes are the most common.

Hex Codes vs. Color Names: Which is Better?

Hex codes are a bit harder to understand than color names, particularly if you’re not used to hexadecimal notation. However, it’s probably better to use hex codes anyway.

First, hex codes provide much more flexibility — you can specify about 16 million different combinations with hex codes. (Many computers can’t display that many colors, but that’s another matter.)

Second, hex codes are more stable and universal. Simple color names such as “red” and “black” will work in just about any browser, but support for fancy color names (“papayawhip”? “eggshell”?) is a bit dodgier. It’s best to stick with hex codes, which work universally.

Links

Without hyperlinks, the Web wouldn’t be “the Web”. This section explains how to create links to other websites (external links) and links within your own website (internal links).

The `a` Element

The anchor element, `a`, creates links. Here’s an example:

Example 2.12. External Links

view htmlview plainprint?

```
<p>
A link to <a href="http://www.google.com/">Google's
home page</a>.
</p>
<p>
A link to
<a href="http://www.google.com/help/basics.html">a
page within Google</a>.
</p>
```

The syntax is straightforward: the `href` attribute indicates where the link should go, and the text between the `<a>` and `</a>` tags becomes the highlighted link text.

Anchor Titles

The optional `title` attribute allows you to provide a little extra information about the link. Some older browsers ignore this attribute, but most modern browsers produce a little floating “tooltip” if you hover your mouse over the link.

Example 2.13. The title Attribute

view htmlview plainprint?

```
Can physics be
<a href="http://www.dctech.com/physics/humor.php"
title="The most hilarious collection of physics
humor anywhere!">funny</a>?
Is that a rhetorical question?
```

You can use the `title` attribute to indicate where the link is going or provide some other useful information. It's nifty, but it doesn't hurt to leave it out, either.

Components of the URL

To construct links to other websites, we need to understand the structure of the URL. Let's take a closer look at the URL "`http://www.google.com/help/basics.html`":

- The **protocol** ("`http://`"): Specifies that we would like to use the HTTP (HyperText Transport) protocol. This is the standard protocol for retrieving web pages, although there are others, such as FTP (File Transfer Protocol) and HTTPS (Secure HyperText Transport Protocol). Most of the time, though, you'll be using "http://" to retrieve pages from external sites.

  You may specify the protocol in upper case, lower case, or anything in between, but most people use all lower case.

- The **website** ("`www.google.com`"): This specifies that we would like to access a page from google.com. More technically, it means:

  1. Go to the top-level domain "com" (a collection of many millions of websites).
  2. Within "com", find the the domain name "google".
  3. Within "google", find the the sub-domain named "www".

  Finding the website is similar to finding a mailing address. The address "1600 Pennsylvania Avenue NW, Washington, DC 20500" means: go to the zip code "20500"; within "20500", find the city "Washington D.C."; within "Washington, DC", find the building "1600 Pennsylvania Avenue NW".

  As with the protocol above, you may use any case you like for specifying the website, although most people use all lower case.

- The **file** path ("`/help/basics.html`"): This specifies that we would like to access the file `basics.html`, which is inside the directory `help/`.

  Unlike the other components, the filepath can be case-sensitive. Depending on the website you're visiting, the file `Basics.html` might be different from the file `basics.html`.

Wait, Which File Was That Again?

This is a little odd — in our first example, "http://www.google.com/", we didn't specify a file or a directory. So what did we ask for, exactly?

First, it turns out the directory is specified. The single forward slash ("/") represents the "root" or "top-level" directory of the website. We're asking Google to give us a file from the top directory of its website.

Okay, but which file are we asking for? We didn't specify the name of the file explicitly, we just asked for a directory. This means means we're asking Google, "Give us the default file for that directory, if you have one." As it turns out, Google is going to give us a file named "index.html", but the exact name could be anything. It's up to Google.

Internal links

Internal links are similar to external links except that you can leave off the protocol and website name. You only need to specify the file path.

File paths for internal links can be absolute or relative. Absolute links always begin with a forward slash. They specify a path that starts at the root directory of your web server. Relative links never begin with a forward slash. They specify a path that is relative to the page that the link is on.

Examples of absolute links:

- `href="/"`: Go to the web root directory and get the default file. (The home page of goer.org.)
- `href="/HTML/`: Go to the web root directory, then the `HTML` directory, and get the default file. (The first page of this tutorial.)
- `href="/HTML/basic/links/index.html"`: Go to the web root directory, then the `HTML` directory, then the `basic` directory, then the `links` directory, and get the file `index.html` (This page.)

Examples of relative links (relative to this page, http://www.goer.org/HTML/basic/links/index.html):

- `href="index.html"`: Get the `index.html` file that is in the current directory. (This page.)
- `href="../"`: Go up one directory from the current directory and get the default file. (The first page of Chapter 2.)
- `href="../../introduction/"`: Go up two directories from the current directory, enter the `introduction` directory, and get the default file. (The first page of Chapter 1.)

As the examples demonstrate, URLs can vary in complexity while still pointing to the same location.

For pages that are "near" the current page, relative links are often simpler. Relative links also work better if you are working with files on your desktop, as opposed to a real websever.

Because absolute links are the same no matter where you are on the website, they're good for standard navigation links, header and footer links. However, absolute links don't work as well if you are just messing around with files on your desktop. On a webserver, an absolute link such as `/HTML/index.html` starts with the webserver's root directory. But on your local desktop, there is no "webserver root directory", and so your computer attempts to follow the filepath `/HTML/index.html` from the root directory of your hard drive. In general, this will fail.

Linking Within Pages

We can link to web pages in external websites and internal websites. What about linking to a particular location in a page?

All elements provide this capability through the `id` attribute. Once you've assigned an `id` to the element, you can link directly to the location on the page where the element appears:

Example 2.14. Linking to an ID

view htmlview plainprint?

```
<h2 id="fruitbat_migratory">
Migratory Patterns of the Fruit Bat
</h2>
...
...
...
<p>
<a href="#fruitbat_migratory">link to 'Migratory Patterns of the Fruit Bat'</a>.
</p>
...
```

The `id`'s value is `fruitbat_migratory`. Further down the page, we have a link to this ID, specified by `href="#fruitbat_migratory"`.

You can combine named anchors with longer URLs: for example, `http://www.goer.org/HTML/basic/links/#parts`.

The pound sign in the URL makes the link point to the named anchor. If we had accidentally written `href="parts"`, that would have been a relative link to the directory named "parts".

Caution

Never give two elements the same ID on the same page.

Images

Once you understand links, adding images is straightforward.

Images are a little different than the HTML headings and paragraphs that we've seen so far. An image is actually a separate file that you embed in your web page using the `img` element. Let's see how this works:

Example 2.15. Example Image

view htmlview plainprint?

```
<p>
  Yet more evidence that I did not go to Art School:
</p>
<img src="/images/html/oscope.gif" alt="a cartoon of an oscilloscope">
```

To try this example on your own, you not only need to copy the HTML, but you also need to download the image "oscope.gif" to your system. If you don't know how to do this, refer to "Save Images". Save the file in the same directory as your test page, and try it out.

The `img` element has two required attributes:

- The `src` attribute provides a URL that points to the image file. If the URL is missing or incorrect, the image won't appear. The `src` attribute is very similar to the `href` attribute of the `a` element. It's too bad that they couldn't both be called "href", because then this would all be easier to remember.
- The `alt` attribute provides "alternative text" for the image, should the image not be

displayable. This could happen if the user turns off images in their browser, or uses a browser that doesn't support images (such as a text-only browser or a screen reader). The alternative text could also appear while the user is waiting for the image to download, or if the link to the image is broken.

Some browsers display the `alt` text as a "tooltip" that pops up if you hover your mouse over the image. Strictly speaking, this is incorrect behavior, because the `alt` attribute is for when the image cannot be displayed. To provide "extra information" about an image, use the optional `title` attribute instead.

An image file is not coded in the HTML language. Images have their own formats, such as GIF, JPEG, or PNG. There are several ways to acquire or create digital images:

- take a photo with a digital camera
- use drawing software to create a digital image
- download an image off the web using your browser's Save Image function

Note

It's fine to mess around with images for learning purposes, but please don't publish someone else's images on a website unless you know you have the owner's permission.

Floating Images

Positioning elements in HTML can be tricky, and we'll be talking about how to do this in the more advanced sections of this tutorial. That said, let's take a sneak preview at one of the basic concepts of positioning: the `float` property.

Example 2.16. Floating Images

view htmlview plainprint?

```
<h1>Not Your Father's Oscilloscope...</h1>

<img src="/images/html/oscope.gif" height="64" width="90"
alt="Right-floating oscilloscope"
style="float: right">

<p>
The <b>Model XJ-2000</b>:
Quality, quality, quality. That's what our bosses
asked for when we designed the XJ-2000, and that's
what we delivered. Our competitors claim to deliver
"quality"... but do they? Sure, they prattle on about
about sample rates, picosecond/div sweep rates,
Fast Fourier Transforms, and other technical
mumbo-jumbo. But the fact is that the XJ-2000 is
light-years ahead of the competition. From the
scratch-resistant chrome case to the innovative
green phosophorescent screen, only one word should
spring to mind when you look at the XJ-2000: quality.
</p>
```

The `float: right` directive does something peculiar: it removes the image from the regular flow of the page and “floats” it to the right. The paragraph flows around the image. If we had specified `float: left`, the image would float to the left of the paragraph. (Try it!)

Just like the `color` and `background` properties, you can actually apply the `float` property to nearly any element. We’ll return to this concept later.

Lists

HTML provides several ways to display lists of information:

- unordered lists: bulleted lists (like this one)
- ordered lists: numbered lists, procedures, formal outlines
- definition lists: lists of terms and definitions

Lists are different from anything we’ve covered so far, because generating a list requires at least two elements working together.

Ordered Lists

Ordered lists are contained by an `ol` element. Within the `<ol>` and `</ol>` tags, each list item is defined by an `li` element:

Example 2.17. Ordered Lists

view htmlview plainprint?

```
<h3>My Dot-com Business Plan</h3>

<ol>
  <li>Buy domain name</li>
  <li>Put up website</li>
  <li>???</li>
  <li>Profit!</li>
</ol>
```

The default ordered list uses arabic numerals (1, 2, 3, …), but you can easily alter this behavior.

Directly within the `ol` element, you can only include `li` elements. However, within the `li` elements you may place nearly any HTML content that you want: text, paragraphs, images, even another list.

Example 2.18. Nested Ordered List

view htmlview plainprint?

```
<h3>Schedule for Monday</h3>

<ol style="list-style-type: upper-roman">
  <li>Suppress Gaul
    <ol style="list-style-type: upper-alpha">
      <li>Two more legions or three?</li>
      <li>Where to put victory arch? Forum's looking crowded...</li>
    </ol>
  </li>
  <li>Have Cicero killed</li>
  <li>Lunch</li>
  <li>Head-and-hands-on-pike viewing at the Forum (casual dress ok?)</li>
</ol>
```

Placing one list within another is a nested list. The outer list will use upper-case Roman numerals, and the inner list will start counting independently, using upper-case letters. There are a large number of available list-style-types for ordered lists. The most common and well-supported values are `upper-roman`, `lower-roman`, `upper-alpha`, `lower-alpha`, and the default, `decimal`. If for some reason the browser doesn't understand the value you specify, it just falls back to the default (decimal numbers).

You can use `ol` for any list of items where the order is important. This includes outlines, procedures, TODO lists, lists of ranked items.

Unordered List

Unordered lists are similar to ordered lists, except that they do not have a particular order. Unordered lists even recycle the `li` element, which makes it easy to turn an ordered list into an unordered list and vice-versa. All you need do is turn your `ol`s into `ul`s:

Example 2.19. Unordered List

view htmlview plainprint?

```
<h3>My Dot-com Business Plan</h3>

<ul style="list-style-type: square; color: #009900">
  <li>Buy domain name</li>
  <li>Put up website</li>
  <li style="list-style-type: disc; color: #ff0000">???</li>
  <li>Profit!</li>
</ul>
```

In the parent element `ul`, we've specified that the list should be green and that the bullets should be squares. We overrode this behavior in the third bullet point, specifying a red, circular bullet. Notice how the style cascades from the parent element to the child elements — each bullet takes on the characteristics of its parent, unless we specify otherwise.

Ordered and unordered lists indent everything enclosed by the `ul` or `ol` block. You might be tempted to use `ul` to indent any old HTML:

```
<ul>
  <p>This is an indented paragraph.</p>
</ul>
```

Don't do this. This is incorrect HTML, and there are better ways to achieve the same effect, as we'll see in "Align and Indent". While we're on the subject, `li` elements must always be enclosed by a `ul` or `ol` element. Don't put `li` elements directly in the `body` of the document; this is also bad practice.

Definition Lists

A definition list provides a list of terms and definitions. Definition lists require *three* elements to construct:

- `dl`: Denotes the definition list itself. Within the `dl` you may only include `dt` and `dd` elements.
- `dt`: Denotes a term within the definition list. Within the `dt` you may only include inline content, such as plain text and inline style elements.
- `dd`: Denotes a definition for a term. Within the `dd` you may include any block or inline content: paragraphs, other lists, whatever you like.

A definition list consists of terms (`dt`) immediately followed by definitions (`dd`).

Example 2.20. Definition List

view htmlview plainprint?

```
<h1>The Devil's Dictionary</h1>

<dl>
 <dt>ABSURDITY, n.</dt>
 <dd>A statement or belief manifestly inconsistent with
 one's own opinion.</dd>

 <dt>ACADEME, n.</dt>
 <dd>An ancient school where morality and philosophy were
 taught.</dd>

 <dt>ACADEMY, n.</dt>
 <dd>[from ACADEME] A modern school where football is
 taught.</dd>

 <dt>ACCIDENT, n.</dt>
 <dd>An inevitable occurrence due to the action of immutable
 natural laws.</dd>
</dl>
```

You may provide multiple definitions for a term, or even multiple definitions *and* multiple terms, although this is less common. Some people also use definition lists to mark up screenplays and other exchanges of dialogue: the speaker is a `dt` and their dialogue is a `dd`. If you're writing technical manuals, you can use definition lists to document variables, interface components, and more. Definition lists are good for any "listed information that comes in pairs."

Congratulations, you've finished the basic section! In the next section we learn about advanced font styling, alignment, margins, and borders. Read on for more…

Chapter 3: Styling Basics

So far we've learned some basic HTML constructs: paragraphs, headings, and lists. That's all very nice, but by themselves, these elements are kind of lifeless. An `h1` is just like any other `h1` out there… right?

Fortunately, no. We can customize the appearance of our `h1`s (and any other element) using a language called Cascading Style Sheets (CSS). We already saw a hint of what style sheets can do when we were mucking around with the foreground color and background color. But CSS can do much more than this.

As an example of what CSS can do, take a walk through the CSS Zen Garden. Each page in the

CSS Zen Garden has exactly the same text and markup. The only difference is the CSS, but each page appears *radically* different.

First, we're going to learn a new method for applying styles that is much more efficient than what we've been doing in the previous chapters. At first, we'll only be changing the color, because color is simple and gets the point across. But by the end of this chapter, you'll have expanded your style vocabulary dramatically.

When you're done with this section, you should be able to:

- Create a style sheet for your web page or for your entire web site.
- Create named, reusable styles using classes and IDs.
- Apply sophisticated font changes to any HTML element.
- Change the padding, margins, and borders.
- Align and indent your text.

Section Summaries

The Styling Basics chapter contains these sections:

1. The `style` Attribute — Describes the basic problem that style sheets are designed to solve.
2. Style Sheets — Describes style sheets, which provide a convenient method for collecting styling rules in one place and applying them to an entire website. We'll cover CSS rule syntax and the basics of cascading.
3. Classes and IDs — Explains how to create more specialized CSS rules by using CSS classes and IDs.
4. Div and Span — Describes the "generic" HTML elements `div` and `span`, which provide a way to apply styles to any given swath of HTML.
5. Font Control —Provides more sophisticated methods for controlling a document's font. We examine how to change the font family, font size, and font styling. We'll also take a look at a stylesheet that is a bit more complex than the ones we've seen in earlier sections.
6. Borders — Explains how to set border properties such as color and thickness, how to set different borders on all four sides, and how to use border shorthand notation.
7. Margins and Padding — Describes padding (extra space inside an element's border) and margins (extra space outside an element's border).
8. Align and Indent — Describes text indentation and alignment. We'll learn how to create hanging indents and align text horizontally. (Aligning blocks will come later, in the sections on positioning.)

The style Attribute

So far we've dealt with two major components to our web pages: the elements, which define the basic structure of the page, and the styling, which further refines how the page should look. We saw the effects of styling in "Colors":

Example 3.1. Red Paragraph (Redux)

view htmlview plainprint?

```
<p>
  The grandmother lived out in the wood, half a league from the village,
  and just as Little Red-Cap entered the wood, a wolf met her. Red-Cap
  did not know what a wicked creature he was, and was not at all afraid
  of him.
</p>
<p style="color: red">
  'Good day, Little Red-Cap,' said he.
</p>
<p>
  'Thank you kindly, wolf.'
</p>
```

Now, it's nice that we can make a single paragraph red, but what if we wanted *all* paragraphs to be red? Or what if we wanted to do something more than just changing the paragraph's color?

In the early 1990s, the only way to do control the color, size, and font of your text was the `font` element. So people wrote a lot of HTML that looked like this:

Example 3.2. The font Element

view htmlview plainprint?

```
<B><I><FONT Size="+2">
<FONT FACE="Palatino, Arial, Helvetica">
<FONT color="#ff0000">
Kneel Before the Power of the Font Element!
</FONT>
</font></Font></i></B>

<p><FONT Size="-1">
<FONT FACE="Arial, Helvetica, sans-serif">
<FONT color="#123456">
Ow! The bad HTML code! It burns us, burns us, precious!
</font></font></FoNt></p>
```

This was a tremendous pain for many reasons. You had to keep typing this kind of markup all over the page, and if your boss told you later on to change the font size, you had to redo all this code again.

The `style` attribute is an improvement over the `font` element, but it still suffers from the same basic problem: it isn't reusable. You don't want to have to add a `style` element to every element on the page. Fortunately, there is a much better way to do all of this. These days, you should never use the `font` element, and the `style` attribute only sparingly if at all. In the next section, we'll look at a *much better* way to do this.

Style Sheets

Every time we've specified `style="(something)"`, we've actually created a CSS rule. Although CSS (Cascading Style Sheets) is a different language from HTML proper, it is an indispensable tool for controlling the appearance of your website. Think of each webpage as having three components:

- **Structure**: The markup is like the page's *skeleton*. It provides the basic structure for the page: headings, sections, paragraphs, lists, and more. Without structure, your web page would just be an undifferentiated blob of text.

- **Content**: The content (text, images, and even audio or video), is like the page's *flesh*. This is the "meat" of the page. Without content, there's nothing to read, and no reason to visit the page in the first place.
- **Presentation**: The CSS is the page's *skin, hair, clothes, and makeup*. Without presentation, your page might be readable, but it won't be very attractive.

So far we've only been working with individual styles applied to individual elements. Although this works, it is not the most efficient way to apply our styles. A far better technique is to collect all our CSS rules in one place, a style sheet, and then apply the style sheet to every single element on the page… or for that matter, every single element throughout the entire website. Let's see how this works.

Embedded Style Sheets

An embedded style sheet affects every element on the page. In order to create an embedded style sheet, we need to go to the `head` of the document. (Remember the `head` element? Finally, here's something else that goes there besides the `title` element.)

Here's an example of an embedded style sheet (please forgive the garish green-on-yellow):

Example 3.3. All Green Paragraphs

view htmlview plainprint?

```
<html>
<head>
  <title>When Irish Eyes Are Smiling</title>
  <style type="text/css">
    p {color: #008000; background: #ffff00}
  </style>
</head>
<body>
  <h1>When <em>Irish Eyes</em> Are Smiling</h1>
  <p>
    Here's how the chorus goes:
  </p>
  <p>
    When Irish eyes are smiling,<br>
    <em>Sure, 'tis like the morn in Spring.</em><br>
    In the lilt of Irish laughter<br>
    <em>You can hear the angels sing.</em><br>
    When Irish hearts are happy,<br>
    <em>All the world seems bright and gay.</em><br>
    And when Irish eyes are smiling,<br>
    <em>Sure, they steal your heart away.</em>
  </p>
</body>
</html>
```

The syntax is a little different from what we are used to, but the core idea (the CSS rule) should look familiar. Let's examine the style sheet's components:

- `<style type="text/css">`: The `style` element indicates the beginning of the style sheet. It must go inside the `head`. The `type` attribute specifies that you are using a CSS style sheet, as opposed to using some other, yet-to-be-invented style sheet language. It's a bit silly, but the HTML specification requires it, so go ahead and add it.

- `p`: The CSS selector. This says, "do something to all `p` elements."
- `{color: #008000; background: #ffff00}`: The CSS declarations. We've seen these before in the `style` element, although without the curly braces. Together with the selector `p`, the declarations constitute a CSS rule: "make all paragraphs green text with a bright yellow background." As mentioned in "Background Color", you must separate multiple declarations with a semicolon.

Note

You will sometimes see people wrap their embedded CSS rules in an HTML comment, like so: `<style><!-- p {color: #008000;} --></style>`. This is a trick that hides your CSS rules from extremely ancient browsers that cannot process CSS at all, namely Mosaic and Netscape Navigator 1.0. Fortunately, these browsers are long-dead, and there is no need to do this anymore.

Now, we could have achieved the same effect by adding these declarations to the `style` attribute of each paragraph. The key difference is that the style sheet affects *all* paragraphs. The selector enables us to specify the declarations *once*, and the browser then applies them to all elements that match the selector.

Multiple CSS Rules and Selectors

You may specify as many rules as you like in a style sheet. Consider the following:

Example 3.4. Green Paragraphs, Red Body

view htmlview plainprint?

```
<html>
<head>
 <title>When Irish Eyes Are Smiling</title>
 <style>
  body {background: #ff0000}
  p {color: #008000; background: #ffff00}
 </style>
</head>
<body>
 <h1>When <em>Irish Eyes</em> Are Smiling</h1>
 <p>
  Here's how the chorus goes:
 </p>
 <p>
  When Irish eyes are smiling,<br>
  <em>Sure, 'tis like the morn in Spring.</em><br>
  In the lilt of Irish laughter<br>
  <em>You can hear the angels sing.</em><br>
  When Irish hearts are happy,<br>
  <em>All the world seems bright and gay.</em><br>
  And when Irish eyes are smiling,<br>
  <em>Sure, they steal your heart away.</em>
 </p>
</body>
</html>
```

Here we’ve specified that the body of the page should have a red background, while all paragraphs should have a yellow background with green text. This color combination is so awful that I am loathe to discuss this example further. Let’s move on.

The key feature of CSS rules is that they are reusable. We’ve already seen that we can use them to apply a set of CSS declarations to, say, all `p` elements. But it gets better — we can also give a rule with multiple selectors. Just separate the selectors with commas:

```
element1, element2 { declarations }
```

This applies the declarations to both `element1` and `element2`. For example:

Example 3.5. Green Paragraphs and Headings

view htmlview plainprint?

```
<html>
<head>
<title>When Irish Eyes Are Smiling</title>
 <style>
   p, h1 {color: #008000; background: #ffff00}
 </style>
</head>
<body>
 <h1>When <em>Irish Eyes</em> Are Smiling</h1>
 <p>
   Here's how the chorus goes:
 </p>
 <p>
   When Irish eyes are smiling,<br>
   <em>Sure, 'tis like the morn in Spring.</em><br>
   In the lilt of Irish laughter<br>
   <em>You can hear the angels sing.</em><br>
   When Irish hearts are happy,<br>
   <em>All the world seems bright and gay.</em><br>
   And when Irish eyes are smiling,<br>
   <em>Sure, they steal your heart away.</em>
 </p>
</body>
</html>
```

Now the green text and yellow background applies to paragraphs *and* level 1 headings.

So far, we’ve only seen simple selectors that just select a single element. However, you can design selectors to be more “choosy” than this. One of the most useful variations is:

```
element1 element2 { declarations }
```

This looks similar to Example 3.5, “Green Paragraphs and Headings”, but notice there is no comma between the elements. That one little comma makes a big difference. This rule applies its declarations only to `element2`s that are contained within `element1`s. Let’s see how this works:

Example 3.6. Green paragraphs, Blue `em`s

view htmlview plainprint?

```
<html>
<head>
  <title>When Irish Eyes Are Smiling</title>
  <style>
    p {color: #008000; background: #ffff00}
    h1 em { color: #0000ff }
  </style>
</head>
<body>
 <h1>When <em>Irish Eyes</em> Are Smiling</h1>
 <p>
   Here's how the chorus goes:
 </p>
 <p>
   When Irish eyes are smiling,<br>
   <em>Sure, 'tis like the morn in Spring.</em><br>
   In the lilt of Irish laughter<br>
   <em>You can hear the angels sing.</em><br>
   When Irish hearts are happy,<br>
   <em>All the world seems bright and gay.</em><br>
   And when Irish eyes are smiling,<br>
   <em>Sure, they steal your heart away.</em>
 </p>
</body>
</html>
```

This specifies `em` elements to be blue, *if and only if* they are contained within an `h1`. The em in the `h1` at the top of the page turns blue, but the `em`s in the paragraphs below are unaffected.

We'll see a couple more examples of useful selectors in "Classes and IDs". For a complete list of selector syntax, refer to the official CSS specification.

Style Sheet Cascading

What happens when CSS rules conflict? Here's where the "Cascading" part of Cascading Style Sheets comes in. The official rules are complicated, but they basically boil down to "whichever CSS rule is 'nearer' or 'more specific' wins."

Returning to our song lyric example, let's say we didn't want the first paragraph to be in green text:

Example 3.7. One Green Paragraph

view htmlview plainprint?

```
<html>
<head>
  <title>When Irish Eyes Are Smiling</title>
  <style type="text/css">
    p {color: #008000; background: #ffff00}
  </style>
</head>
```

```
<body>
  <h1>When <em>Irish Eyes</em> Are Smiling</h1>
 <p style="color: #000000">
   Here's how the chorus goes:
 </p>
 <p>
   When Irish eyes are smiling,<br>
   <em>Sure, 'tis like the morn in Spring.</em><br>
   In the lilt of Irish laughter<br>
   <em>You can hear the angels sing.</em><br>
   When Irish hearts are happy,<br>
   <em>All the world seems bright and gay.</em><br>
   And when Irish eyes are smiling,<br>
   <em>Sure, they steal your heart away.</em>
 </p>
</body>
</html>
```

The first paragraph has black text. Our style attribute is more specific than the general page-level CSS, and so the black-text rule overrides the green-text rule. However, the paragraph still has a yellow background. The yellow-background rule cascades down to this paragraph, because we didn’t specify anything about it.

Another important cascading rule is that parent elements apply their style to their children (enclosed elements). Rules applied to the `body` element cascade down to enclosed `p` elements, rules applied to `p` elements cascade down to enclosed `em` elements, and so on. In fact, we saw this form of cascading in Example 3.4, “Green Paragraphs, Red Body”. The background of the page is red, and any child elements also have a red background… except for paragraphs, which specifically override this rule.

Reusable Style Sheets

What’s better than applying styles to an entire page? Applying it to an entire website.

CSS enables you to place your style sheet in a separate file. The stylesheet then becomes a central repository of CSS rules that you can apply to any page in a site. There are two ways to “call” a global style sheet into a given page:

Example 3.8. Global Style Sheets: The “link” Method

view htmlview plainprint?

```
<html>
<head>
  <title>When Irish Eyes Are Smiling</title>
  <link rel="stylesheet" href="/css/style1.css" type="text/css">
  <link rel="stylesheet" href="/css/style2.css" type="text/css">
  <style>
    p {color: #008000; background: #ffff00}
  </style>
</head>
```

Example 3.9. Global Style Sheets: The “@import” Method

view htmlview plainprint?

```
<html>
<head>
  <title>When Irish Eyes Are Smiling</title>
  <style>
    @import url("/css/styles3.css");
    @import url("/morecss/styles4.css");
    body {background: #ff0000}
    p {color: #008000; background: #ffff00}
  </style>
</head>
```

The two methods have bizarrely different syntax, but functionally they are similar. They both allow you to specify multiple stylesheet files, and you may use them both in conjunction with an embedded style sheet.

A linked style sheet also affects the rules for cascading:

- A rule in a linked style sheet has low priority.
- A rule in an embedded style sheet has medium priority.
- A declaration in a `style` attribute has high priority.

This means you can specify a style in a linked style sheet, override that style for a particular page, and then override it *again* for a particular element. For example, if we have the page:

view htmlview plainprint?

```
<html>
<head>
  <title>What is your favorite color?</title>
  <link rel="stylesheet" href="/css/style.css" type="text/css">
  <style>
    em {color: yellow}
  </style>
</head>
<body>
  My favorite color is <em style="color: blue">blue</em>.
</body>
```

and the linked stylesheet “`style.css`” contains:

```
em {color: blue}
```

then the browser uses the following logic to color the `em`:

1. The linked style sheet specifies the `em` to be blue.
2. The embedded style sheet specifies the `em` to be yellow, overriding the linked style sheet.
3. The `style` attribute specifies the `em` to be blue again, overriding the embedded style sheet.

So the resulting color is “blue”. Whew.

Classes and IDs

The last section demonstrated how a style sheet can apply a style to all elements in a web page. If we want all our paragraphs to be red, we just need to specify `p {color: red}` in the style sheet. Easy, we’re done.

But what if we want only some paragraphs to be red? We could override our “red-text” rule by making copious use of the style attribute: `<p style="color: black">`. However, as [“The `style` Attribute”] points out, this is not a good solution in the long term.

Fortunately, there is a way to “name” a style and re-use it over and over. This method is called a style sheet class.

Style Sheet Classes

In Example 3.10, “Two CSS Classes”, we have an exchange of dialogue between two characters. We’ll differentiate each character’s lines by applying a unique style. It would be tedious and counterproductive to write out the full style in each `p` element, so instead we’ll define each style once in the style sheet and use a class to call it multiple times.

Example 3.10. Two CSS Classes

view htmlview plainprint?

```
<html>
<head>
  <title>An Ideal Husband</title>
  <style type="text/css">
    p {color: #000000}
    p.goring {color: #ff0000}
    p.phipps {color: #008000}
  </style>
</head>
<body>
  <p class="goring">
    LORD GORING. [Taking out old buttonhole.] You see, Phipps, Fashion
    is what one wears oneself. What is unfashionable is what other
    people wear.
  </p>
  <p class="phipps">
    PHIPPS. Yes, my lord.
  </p>
  <p class="goring">
    LORD GORING. Just as vulgarity is simply the conduct of other
    people.
  </p>
  <p class="phipps">
    PHIPPS. Yes, my lord.
  </p>
  <p class="goring">
    LORD GORING. [Putting in a new buttonhole.] And falsehoods the
    truths of other people.
  <p class="phipps">
    PHIPPS. Yes, my lord.
  </p>
  <p class="goring">
    LORD GORING. Other people are quite dreadful. The only possible
    society is oneself.
  </p>
```

```
<p class="phipps">
  PHIPPS. Yes, my lord.
</p>
<p class="goring">
  LORD GORING. To love oneself is the beginning of a lifelong romance,
  Phipps.
</p>
<p class="phipps">
  PHIPPS. Yes, my lord.
</p>
</body>
</html>
```

For each character, we've defined a class, `goring` and `phipps`. Note that we've introduced a brand-new CSS selector: instead of

```
element { declarations }
```

We have

```
element.class { declarations }
```

Once we define the selectors `p.goring` and `p.phipps`, we can apply them to any paragraph through the `class` attribute. You may add the `class` attribute to any element that can take the `style` element (in other words, most of them). The `class` attribute is usually much better to use than the `style` attribute, because it's much easier to change what a class means later on.

In Example 3.10, "Two CSS Classes", the classes only work with the `p` element. Applying them to some other element, as in, `<em class="goring">`, would fail. To make the selector "generic" so that it works with any element, just leave off the element name:

```
.goring {color: #ff0000}
```

Unique IDs

Another method for assigning styles is to use an ID. An ID is a unique identifier for a element on a page. IDs are different from classes in a couple of key ways:

- IDs are unique. You may only use a particular ID once on a page.
- You can use IDs to create a link to a specific location within a page, as discussed in "Linking Within Pages".

To apply a style to an ID, the syntax should look familiar. Instead of using a period, use a hash mark (#):

```
p#goring {color: #ff0000}
```

That's it. In order to apply the style, you would just write `<p id="goring">`, as opposed to `<p class="goring">`.

The only problem here is that you may only apply `id="goring"` once on the entire web page. In other words, we are presuming that Lord Goring is speaking only once. Given what we know of Lord Goring, this is probably a bad assumption.

If you use IDs in your style sheets, make sure that they are for unique elements. IDs are ideal for navigation bars, headers, footers, and other elements that appear only once on the page.

Div and Span

As we’ve seen in previous sections, using CSS is straightforward. Provide a selector (”select all `p` elements”), provide a rule (”make their text red”), and we’re done. Both the selectors and the rules can be much more complicated, but this is the core of it.

Let’s take a look at another component of our style sheet toolbox: the “generic” elements, `div` and `span`. These elements are designed to help you mark off arbitrary chunks of HTML for styling.

The `span` Element

The `span` element is the generic inline element. Let’s see this element in action:

Example 3.11. Empty `span`

view htmlview plainprint?

```
<p>
  Towards thee I roll, thou all-destroying but
  unconquering whale; <span>to the
  last I grapple with thee; from hell's heart I stab
  at thee; for hate's sake I spit my last breath at
  thee.</span> Sink all coffins and all
  hearses to one common pool! and since neither can
  be mine, let me then tow to pieces, while still
  chasing thee, though tied to thee, thou damned
  whale! Thus, I give up the spear!
</p>
```

If you view the results, it looks like… the `span` didn’t do anything. And that’s the point: by default, the `span` element isn’t supposed to do anything. It serves as a placeholder or marking point for beginning and ending a style.

Let’s make a minor change. We’ll create a class for our `span` element and apply that class to our selection from Moby-Dick.

Example 3.12. Red `span`

view htmlview plainprint?

```
<html>
<head>
  <title>Moby-Dick</title>
  <style type="text/css">
    span.khan {color: #ff0000}
  </style>
</head>
<body>
  <p>
    Towards thee I roll, thou all-destroying but
    unconquering whale;
    <span class="khan">to the last I
    grapple with thee; from hell's heart I stab at
    thee; for hate's sake I spit my last breath at thee.
    </span> Sink all coffins and all
    hearses to one common pool! and since neither
    can be mine, let me then tow to pieces, while
    still chasing thee, though tied to thee, thou
    damned whale! Thus, I give up the spear!
  </p>
</body>
</html>
```

Now we have an inline element, `<span class="khan">`, that can make any inline selection of HTML red.

Tip

Before using the generic `span` element, try using an `em`, `strong`, `code`, or other logical style. If possible, it's better to start with an element that has some intrinsic meaning.

The `div` Element

The `div` element is the generic block element. You can use `div` to apply styles to large sections of HTML, such as multiple paragraphs. Like all block elements, the `div` element defines a “box” of HTML.

Example 3.13. Red `div`

view htmlview plainprint?

```
<html>
<head>
  <title>Product Brochure</title>
  <style type="text/css">
    div.important {color: #ff0000}
  </style>
  </head>
<body>
  <p>
  Congratulations on the purchase of your sword!
  Using a sword is fun and easy. Just be
  sure to follow these important safety tips.
</p>
<div class="important">
  <p>
    <em>Never</em> hold your sword by the pointy end.
  </p>
  <p>
    <em>Always</em> be sure to stick the pointy end
    into the other guy before he does the same to you.
  </p>
</div>
<p>
  And remember, if you or your surviving kinsfolk are
  not fully satisfied, we have a money-back guarantee!
</body>
```

27.</html>

The div element applies the style to the second and third paragraphs. Note that even though the div element is a “block” element, it does not create extra line breaks between paragraphs 1 and 2 or paragraphs 3 and 4. However, if we had tried to insert the div inside one of the paragraphs, we would have created a new text block. (Try it.)

Font Control

In our earlier section about font styles, we looked at the effects of the em and strong elements on text. While these elements enable you to apply some simple font changes (italic, bold), it would be nice to have more options.

Now that we’ve covered the basics of style sheets, it’s time to examine more sophisticated methods of font control. CSS font control is divided into three main components:

- font-family: What is the name of the font?
- font-size: How big is the font?
- font-weight: Is the font bold?
- font-style: Is the font italic?
- text-decoration: Is the font underlined, struck through, …?

We’ll tackle each of these properties one-at-a-time.

The font-family Property

The font-family property selects the name of the font you wish to use. This component is pretty straightforward:

```
body { font-family: times new roman }
```

This CSS rule specifies that all text in the body should be in Times New Roman. Times New Roman isn’t necessarily a great font to use on the web, but it should be familiar to anyone who uses a word processor.

Specifying Multiple Fonts

One of the unavoidable issues of web design is that you can’t count on any particular browser rendering your page exactly the way you want it to be. Fonts are no exception. You might want your text to use some fancy font, but what if your reader’s browser doesn’t have that font available?

To resolve this issue, font-family enables you to specify multiple fonts. Here’s an example:

```
body { font-family: georgia, palatino, times new roman, serif }
```

The list of fonts proceeds in descending order of importance. The browser first tries to display the font Georgia. If Georgia is not available, it will try Palatino; failing that, Times New Roman; and if all hope is lost, a generic serif font.

There are some fonts that are *usually* available on on most people’s browsers. These include Times New Roman, Georgia, Palatino, Lucida, Impact, Trebuchet MS, Verdana, Comic Sans MS, Arial, Courier New, and Courier. However, even this impoverished selection is not always available. Worse, some of these “canonical” fonts are ill-suited for reading on a computer screen. Many websites declare their fonts to be “Arial, Helvetica, sans-serif”. But as accessibility expert Joe Clark says in his book Building Accessible Websites:

> “Don’t use Helvetica. Typographic neophytes think Helvetica is “legible.” Try running a few tests with confusable characters like Il1i!Â¡|, 0OQ, aeso, S568, or quotation marks. Related grotesk typefaces like Univers suffer similarly. (One more time: Don’t use Arial. It’s a bastardized variant of Helvetica, it’s ugly, it bespeaks unsophistication, and it sticks you with all the same confusable characters as other grotesks.)”

Generic Fonts

If you specify the keyword “sans-serif” at the end of a list of fonts, this directs the browser to use some form of sans-serif. Specifying a generic font keyword is always a last resort, to be used if none of the fonts you name in your list are available.

There are five varieties of generic fonts, but only the first three are all that important.

- `sans-serif`: lacks decoration or flaring at the end of each stroke.

  Examples include Verdana, Arial, and Geneva.

- `serif`: has decoration or flaring at the end of some of its strokes.

  Examples include Times New Roman, Garamond, and Georgia. Printed materials often use serif fonts, because the serifs make the characters easier to distinguish and thus, easier to read. However, on the web, the reverse can be true. On paper, the little complicated serifs are sharp and easy to distinguish, but on a fuzzy computer screen, it’s often easier on the eyes to use sans-serif.

- `monospace`: uses the same width for each character, often used to represent computer code or terminal text.

  Examples include Courier, Courier New, Monaco, and Andale Mono. The most well-supported monospace fonts are Courier and Courier New.

- `cursive`: designed to mimic handwriting.

  Examples include Brush Script MT, Aristocrat LET, and Comic Sans MS. The most well-supported cursive font is Comic Sans MS, a cartoony font designed to resemble a child’s writing. Most other cursive fonts are not well-supported.

- `fantasy`: highly decorative.

  Examples include Impact and Copperplate. Most fantasy fonts are not well-supported.

The font-size Property

The `font-size` property enables you to specify your font size in a number of absolute and relative ways:

- points (”`font-size: 12pt`“): Measured in “points” (units of 1/72 of an inch).
- pixels (”`font-size: 14px`“): Measured in screen pixels.
- percentages (”`font-size: 150%`“): Measured relative to the default font or the parent element’s font. A font size of “`75%`” would shrink the font by a factor of 3/4.
- em-widths (”`1.2em`“): (Not to be confused with the `em` element.) Measured relative to the default font or the parent element’s font. The value “`1.0em`” is the same as multiplying by

one.

- absolute keywords (”`x-large`“): Specifies one of seven sizes: `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, and `xx-large`. The exact interpretation of what constitutes “`medium`” or “`x-large`” varies from browser to browser.
- relative keywords (”`larger`“): Bumps the font up one size or down one size from its default or inherited size. The options are `larger` and `smaller`.

Note

Always keep in mind that people can and will override your font size settings. All browsers provide a text-zoom function, and people with weak eyesight (think, “people over thirty”) are quick to make use of it.

Modern browsers handle these ways of specifying the font size reasonably well, although there are some inconsistencies. But in the older browsers, font size handling is spotty at best. For example:

- Old versions of Internet Explorer used a default font size of `small` instead of `medium`.
- For the size keywords, Netscape 4 used a *huge* scaling factor between the different size keywords. Using `xx-small` or `x-small` often made the text unreadable, `x-large` and `xx-large` were a sight to behold.
- The only way to get consistent font sizes across all the old browsers was to use pixels (`px`). However, due to a bug in old versions of Internet Explorer, using pixels disabled Internet Explorer’s ability to resize the text! That’s why you’ll see some websites with “text resize buttons”, even though these buttons shouldn’t be necessary.

Although these issues are generally improving over time, it can be quite a struggle to get your font sizes looking consistent and resizing properly even in the newer browsers.

Tip

The Yahoo! User Interface team provides a free `fonts.css` file that solves many of these consistency and resizing problems for you.

Additional font-styling

Beyond specifying the font name and its size, you can further alter the font’s appearance. We’ve already covered how to change the text color in “Colors”. Further alterations include making the text bold (darker), italic (slanted), and much more.

font-weight (boldness)

The `font-weight` component specifies how thick the font’s strokes should be. The important values to know are `normal` and `bold`.

You may also specify, `bolder`, `lighter`, or a number from 100 to 900. (400 corresponds to `normal` and 700 to `bold`.) However, these variations don’t do much even in the newer browsers. In general, everything 500 and above is just bold. There isn’t any such thing as *really* bold, at least not yet.

The `normal` option removes any boldness that has already been applied. For example, most browsers display `strong` as bold text. If you wanted to take this effect away, you could write `strong {font-weight: normal}`; in your style sheet.

font-style (italic)

The `font-style` component specifies whether or not the font is slanted (italic). The most commonly-used value is `italic`.

As with `font-weight`, there is also a value of `normal` that enable you to un-italicize something that would ordinarily be italicized. For a demonstration, refer to “An Example: Styling the `em` Element”.

text-decoration (underlines and more)

Sadly, `text-decoration` doesn’t have a sensible, consistent name (such as “font-decoration”). But its purpose is similar to the components we’ve already discussed. `text-decoration` provides five options:

- `none`: cancels all text-decoration.
- `underline`: creates underlined text.
- `overline`: creates overlined text.
- `line-through`: creates line-through (”strike-through”) text.
- `blink`: creates blinking text.

Caution

Avoid using `blink`. Most people hate blinking text, and many browsers don’t implement this option anyway.

Take care using `underline`. Most people associated underlined text with hyperlinks. Avoid underlining your (non-link) text unless absolutely necessary.

An Example: Styling the `em` Element

While most browsers display `em` as simply italic, there’s no intrinsic reason why this should be so — it just happens to be the convention. Let’s say that instead we want our `em` elements to be bold, red, and larger than our default font size. How would we do this?

Example 3.14. Styled Elements

view htmlview plainprint?

```
<html>
<head>
<title>Oh Better Far to Live and Die</title>
  <style type="text/css">
    p {
      font-family: courier, monospace;
      font-size: 12pt;
    }
    em {
      font-style: normal;
      font-weight: bold;
      font-size: 14pt;
      color: #ff0000;
  }
 </style>
</head>
<body>
 <p>
  KING:<br>
  For... I <em>am</em> a Pirate <em>King!</em><br>
  And it is, it is a glorious thing<br>
  To be a Pirate King!<br>
  For I am a Pirate <em>King!</em>
 </p>
 <p>
  ALL:<br>
  You are! <br>
  <em>Hurrah</em> for the Pirate King!
 </p>
 <p>
  KING:<br>
  And it is, it is a glorious thing <br>
  To be a Pirate King.
 </p>
</body>
</html>
```

We've defined a default style for our paragraphs, and the `em` element style overrides it. Notice that we don't have to specify the font-style for our paragraphs: the default value is normal (non-italic), and presumably we're happy with that.

The font styling that we apply to the `p` element cascades down to the `em` element. Let's take a look at what's happening step-by-step.

1. `font-family`:
   - `p` element: set to `courier, monospace`. If the browser can display Courier it will; otherwise it will serve up a generic monospace font.
   - `em` element: inherits its font-family from the parent element, `p`.
2. `font-size`:
   - `p` element: set to `12pt`.
   - `em` element: set to `14pt`, overriding its parent.
3. `font-weight`:
   - `p` element: not set; takes on the default value of `normal`.
   - `em` element: set to `bold`, overriding the default `em` value of `normal`.
4. `font-style`:
   - `p` element: not set; takes on the default value of `normal`.
   - `em` element: set to `normal`, overriding the default `em` value of `italic`.
5. `color`:
   - `p` element: not set; takes on the default value of `#000000`.
   - `em` element: set to `#ff0000`, overriding the default `em` value of `#000000`.

Font Shorthand Notation

Typing `font-family`, `font-size`, and `font-style` over and over again can get cumbersome. Fortunately, there is a shorthand for all of these properties with a simple name, `font`.

You need merely place the desired values in the proper order. This shorthand only encompasses the properties that begin with “font-”, so if you want to do something else such as underlining text, you still need to use the appropriate property, `text-decoration`.

Here is the order of the properties:

1. `font-style` or `font-weight` or both
2. `font-size`
3. `font-family`

You may leave out any of these properties except for the `font-family`, as long as everything is in the proper order. Let’s take a look at an example of some paragraph classes.

Example 3.15. The font Property

```
p {
    font: verdana, sans-serif;
}
p.styled {
    font: 12pt verdana, sans-serif;
}
p.reallystyled {
    font: bold italic 9pt verdana, sans-serif;
    text-decoration: underline;
    color: #008000;
}
```

The default paragraph is just Verdana, in whatever font size the page default is currently in. The standard class, `styled`, is 12pt Verdana. The second class, `reallystyled`, is 9pt bold italic underlined green Verdana.

The `font` property makes things a bit more elegant, but keep in mind that some browsers will display styles that are not properly ordered, while others are more strict about following the rules. To be safe, always verify that that your `font` declarations are in the proper order before proceeding.

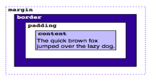
Box Model: Borders

The CSS box model is a collection of CSS properties that apply to all visual HTML elements. As Figure 3.1, “The Box Model” indicates, any HTML element, such as:

```
<p>The quick brown fox jumped over the lazy dog.</p>
```

is nested inside three boxes:

Figure 3.1. The Box Model



The margin encloses the border, which encloses the padding, which encloses the content. You can use CSS to style all three of these independently.

Border Widths, Styles, and Colors

The easiest component of the box model to understand is the border, because it can be so visually striking. Most elements do not have borders by default, but they're easy enough to add with these CSS properties:

- `border-width`: A length, such as `3px` or `0.2em`. Most web designers specify their borders in pixels.
- `border-color`: A color, such as `green` or `#ff0000`. The default color is usually black.
- `border-style`: A keyword that indicates how to draw the border's lines. The basic style is `solid`, but there are fancier options available, such as `dashed`, `dotted`, `double`, `groove`, `inset`, `outset`, and `ridge`. There is also a `none` option that suppresses borders. If a browsers does not support a particular "fancy" border style, it usually replaces that style with `solid`.

You can apply borders to any visible element you like: divs, lists, and so on. In the following example, we create generic classes, and apply them to `p` elements.

Example 3.16. Four Border Examples

view htmlview plainprint?

```
<html>
<head>
 <style>
  .thickline {
    border-width: 10px;
    border-style: solid;
  }

  .thinblueline {
    border-width: 1px;
    border-color: #0000ff;
    border-style: solid;
  }

  .greenridge {
    border-width: 5px;
    border-color: green;
    border-style: ridge;
  }

  .empurpledash {
    border-width: 0.5em;
    border-color: #800080;
    border-style: dashed;
  }
 </style>
</head>
<body>
 <p class="thickline">
  "1st, The man-mountain shall not depart from our
  dominions, without our license under our great seal.
```

```
</p>
<p class="thinblueline">
  2d, He shall not presume to come into our metropolis,
  without our express order; at which time, the inhabitants
  shall have two hours warning to keep within doors.
</p>
<p class="greenridge">
  3d, The said man-mountain shall confine his walks
  to our principal high roads, and not offer to walk, or
  lie down, in a meadow or field of corn.
</p>
<p class="empurpledash">
  4th, As he walks the said roads, he shall take the
  utmost care not to trample upon the bodies of any
  of our loving subjects, their horses, or carriages, nor
  take any of our subjects into his hands without their
  own consent.
</p>
</body>
</html>
```

The first three borders are specifed in pixels. The last is in `em`, which means that if we change the `p`'s font size, the border thickness should scale with it.

Four Different Sides

To apply a different border to each side, you must break down `border-width`, `border-color`, and `border-style` into their constituent components:

- `border-width` becomes: `border-top-width`, `border-bottom-width`, `border-right-width`, and `border-left-width`.
- `border-color` becomes: `border-top-color`, `border-bottom-color`, `border-right-color`, and `border-left-color`.
- `border-style` becomes: `border-top-style`, `border-bottom-style`, `border-right-style`, and `border-left-style`.

Here's an admittedly silly example with four different borders:

Example 3.17. Four Different Sides

view htmlview plainprint?

```
<html>
<head>
  <style>
    p.fourborders {
      border-top-width: 5px;
      border-top-color: #ff0000;
      border-top-style: dashed;

      border-bottom-width: 10px;
      border-bottom-color: #336600;
      border-bottom-style: groove;
  
      border-right-width: 20px;
      border-right-color: #000080;
      border-right-style: solid;
  
      border-left-width: 10px;
      border-left-color: #CC9900;
      border-left-style: solid;
    }
</style>
</head>

<body>
  <p class="fourborders">
    "But I thought all witches were wicked," said the girl, who
    was half frightened at facing a real witch. "Oh, no, that is a
    great mistake. There were only four witches in all the Land of
    Oz, and two of them, those who live in the North and the South,
    are good witches. I know this is true, for I am one of them
    myself, and cannot be mistaken. Those who dwelt in the East and
    the West were, indeed, wicked witches; but now that you have
    killed one of them, there is but one Wicked Witch in all the Land
    of Oz--the one who lives in the West."
  </p>
</body>
</html>
```

Border Shorthand Notation

As with the `font-` properties, there is a shorthand notation named `border`. Unlike `font`, you may specify the `border` properties in any order.

Below is a version of Example 3.18, “Four Border Examples”, reformulated to use the shorter notation. The properties appear in essentially random order, but the results are the same as before.

Example 3.18. Four More Border Examples

view htmlview plainprint?

```
<html>
<head>
 <style>
  .thickline {
    border: 10px solid;
  }

  .thinblueline {
    border: 1px #0000ff solid;
  }

  .greenridge {
    border: ridge 5px green;
  }

  .empurpledash {
    border: #800080 0.5em dashed;
  }
 </style>
</head>
<body>
 <p class="thickline">
  "1st, The man-mountain shall not depart from our
  dominions, without our license under our great seal.
 </p>
 <p class="thinblueline">
  2d, He shall not presume to come into our metropolis,
  without our express order; at which time, the inhabitants
  shall have two hours warning to keep within doors.
 </p>
 <p class="greenridge">
  3d, The said man-mountain shall confine his walks
  to our principal high roads, and not offer to walk, or
  lie down, in a meadow or field of corn.
 </p>
 <p class="empurpledash">
  4th, As he walks the said roads, he shall take the
  utmost care not to trample upon the bodies of any
  of our loving subjects, their horses, or carriages, nor
  take any of our subjects into his hands without their
  own consent.
 </p>
</body>
<html>
```

This shorthand is convenient, but keep in mind that the `border` property can only set the same border on all four sides. Fortunately, there are yet four *more* shorthand properties that apply the border's width, color, and style to just one side at a time: `border-top`, `border-right`, `border-bottom`, and `border-left`. For example, if you wanted to apply the green ridge border to just the left side of an element, the code:

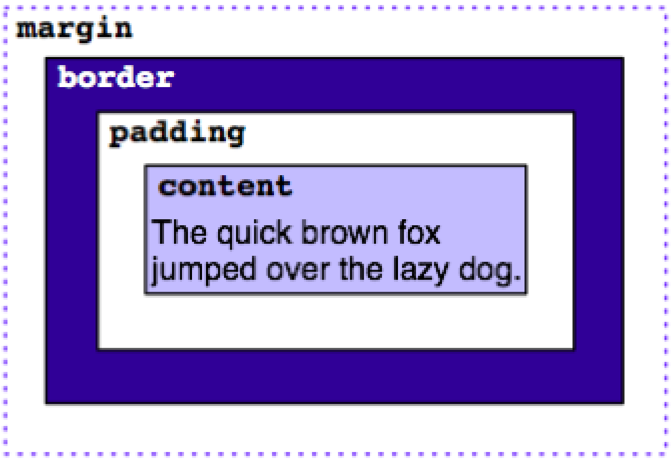
```
.greenridge {
 border-left: ridge 5px green;
}
```

would do the trick.

Box Model: Margins and Padding

Now that we've covered borders, let's return to our examination of the box model.

Figure 3.2. The Box Model (Again)



Padding

In “Borders”, you might have noticed that in all border examples, the text and the borders were rather close together.

An element’s padding is the element’s “extra space inside the border”. There are four padding properties: `padding-top`, `padding-bottom`, `padding-left`, and `padding-right`. You may set these values to `px`, `em`, percentages, or any other valid length unit.

Let’s take a look at an example. Example 3.19, “Non-uniform Padding” specifies two CSS rules:

- The first rule applies to all `div` elements, and changes the background, the text color, and the font family. The gray background is necessary so that we can easily see padding for all divs, while the other two properties are purely cosmetic.
- The second rule applies only to `div` elements with a class of `padded`. Any `div` with this class will have special, non-uniform padding.

Example 3.19. Non-uniform Padding

view htmlview plainprint?

```
<html>
<head>
 <style>
  div {
    background: #444;
    color: white;
  }

  div.padded {
    padding-top: 10px;
    padding-right: 0px;
    padding-bottom: 0.25in;
    padding-left: 5em;
  }
 </style>
</head>
<body>
```

```
  <div>
    No padding (but some style added)
  </div>
  <br>
  <div class="padded">
    Padded<br>
    Top: 10px; bottom: 0px<br>
    Left: 5em; right: 0px
  </div>
</body>
</html>
```

The example includes a `padded` div, followed an ordinary div for comparison. The padded div has padding that varies for each side of text, as we specified. It turns out that 5em (”five m-widths”) is considerably larger than 10px, so the text looks shoved over to the right. As for the ordinary div, we didn’t specify any padding rules for it, so it takes the browser default padding: 0px on all sides.

Padding Shorthand Notation

There is also a shorthand property, `padding`. This property allows you to specify all your padding on one line:

- `padding: 20px` — Padding for all four sides is 20 pixels.
- `padding: 20px 80px` — Padding for the top and bottom is 20 pixels, the left and right is 80 pixels.
- `padding: 20px 80px 10px` — Padding for the top is 20 pixels, the left and right is 80 pixels, and the bottom is 10 pixels.
- `padding: 20px 80px 10px 40px` — Padding for the top is 20 pixels, the right is 80 pixels, the bottom is 10 pixels, and the left is 40 pixels.

As with the previous example, Example 3.20, “Shorthand Padding” provides basic styling for all `div`s, and then uses classes to change the padding for specific `div`s. For additional decoration, we’ve also added a dark red border to each div.

Example 3.20. Shorthand Padding

view htmlview plainprint?

```
<html>
<head>
  <style>
    div {
      background: #444;
      color: white;
      border: #600 6px ridge;
    }

    div.one {
      padding: 20px;
    }

    div.two {
      padding: 20px 80px;
    }

    div.three {
      padding: 20px 80px 10px;
    }

    div.four {
      padding: 20px 80px 10px 40px;
    }
  </style>
</head>
<body>
  <div class="one">
    padding: 20px (all)
  </div>
  <br>
  <div class="two">
    padding: 20px 40px (top/bottom, right/left)
  </div>
  <br>
  <div class="three">
    padding: 20px 40px 10px (top, right/left, bottom)
  </div>
  <br>
  <div class="four">
    padding: 20px 40px 10px 30px (top, right, bottom, left)
  </div>
</body>
</html>
```

The padding is completely contained within the borders. For example, the inside edge of the first `div`'s border is 20px from the far left edge of the text. Try changing the size of the borders. Even if you make the border very thick, the padding stays the same thickness.

Of these four notations, the first (same padding for all sides) is the most commonly used and the easiest to remember. However, if you want to baffle and impress your web designer friends, you should memorize all four.

Margins

Margins are similar to padding. There are four margin properties: `margin-top`, `margin-bottom`, `margin-left`, and `margin-right`. There is also a shorthand property, `margin`, which uses the same syntax as the `padding` property.

The key difference between margins and padding is that padding adds extra space *inside* the border, while margins add extra space *outside* the border:

Example 3.21. Two Margin Blocks

view htmlview plainprint?

```
<html>
<head>
  <style>
```

```
    div.even {
      background: #cfc;
      border: 1px solid;
      padding: 5px;
      margin: 40px;
    }

    div.uneven {
      background: #fcf;
      border: 1px solid;
      padding: 5px;
      margin-top: 20px;
      margin-right: 10px;
      margin-bottom: 40px;
      margin-left: 80px;
    }
  </style>
</head>
<body>
  <div class="even">
    margin: 40px (all)<br>
    padding: 5px (all)
  </div>
  <div class="uneven">
    margin-top: 40px<br>
    margin-right: 30px<br>
    margin-left: 20px<br>
    margin-bottom: 10px<br>
    padding: 5px (all)
  </div>
</body>
</html>
```

We’ve set the padding to a uniform 5px, and we’ve set the background color to provide a little contrast. The first box’s margin is set to a uniform 40px. The second box’s margin is set to four different values. (The equivalent shorthand setting is “`margin: 20px 10px 40px 80px`”.)

Collapsing Margins

Without a border, it’s hard to tell the difference between the padding and the margin. One key difference is that the element’s background color applies to the padding, but not the margin. Another key difference is that vertical margins collapse.

At first glance you might guess that in Example 3.21, “Two Margin Blocks”, the vertical separation between the two inner blocks would be 60px: 40px for the bottom margin of the first block plus 20px for the top margin of the second block. However, the margins collapse to 40px.

To simplify things greatly, margin collapse occurs when:

- the margins are vertical (horizontal margins never collapse)
- and:
  - the blocks are directly *adjacent* (if there is anything between the blocks, the margins do not collapse)
  - **or** the blocks are *nested* and the parent block has no vertical borders or padding (if

the parent block has vertical borders or padding, the margins do not collapse. For an example, refer to “Fancier Blockquotes”)

- and the blocks are left in the normal “flow” of the page (something you only need to worry about when using CSS for more advanced things, such as page layout)

Margin collapsing sounds pretty complicated, and it is. You might be wondering why it’s there in the first place.

Suppose we lived in a world where there is no such thing as margin collapsing. If we had a sequence of five paragraphs, and we want even spacing of, say, 10px between each paragraph, how would we achieve this, exactly? In short, you’re not going to get nice spacing without a lot of hand-tweaking.

- If each paragraph has a top and bottom margin of 10px, then the margin at the top and bottom of the group of paragraphs would be 10px (good), but the spacing in between each paragraph would be 20px (bad).
- Okay, what if we just specify `margin-bottom` as 10px, and leave `margin-top` as 0px? That’s better, but we still would have a problem at the very top, where the first paragraph butts up against the top of the page (or whatever is above the paragraph).
- Specifying `margin-top` as 10px creates a similar problem: there would be no spacing between the bottom of the last paragraph and the top of the next element.

Back in the real world, margin collapsing is in effect, and so the aforementioned problem doesn’t exist. If you write a bunch of paragraphs in a row, the browser just “does the right thing”, collapsing margins so that even though each paragraph defines a top and bottom margin, the space between paragraphs doesn’t get doubled.

Styling with the Box Model

Now that we’ve covered the components of the box model, you might be thinking, “Sure, it’s nice that you can surround every paragraph with a bright lavender ridged border, but is that really all that practical?” Let’s step through a couple of more real-world examples.

Text Decoration with Borders

Selective borders enable you to add styling to inline text that you couldn’t ordinarily apply. As an example, one popular convention on the Web is to style `abbr` with a dotted underline. It would be nice if we could use `text-decoration` to simulate this effect. Creating a dotted underline using `text-decoration` might be possible in the future, but at the time of this writing this CSS feature is still in draft status. However, we can use borders to work around this problem:

Example 3.22. Dotted Underline

view htmlview plainprint?

```
<html>
<head>
  <style>
    abbr {
      border-bottom: 1px #000000 dotted;
    }
  </style>
</head>
<body>
```

```
<p>
  When my
  <abbr
    title="Person of Opposite Sex Sharing Living Quarters"
  >POSSLQ</abbr>
  and I went
  <abbr
    title="Self-Contained Underwater Breathing Apparatus"
  >SCUBA</abbr>
  diving yesterday, we were attacked by
  <abbr
    title="North Atlantic Treaty Organization">NATO</abbr>-controlled
  sharks with frickin'
  <abbr
    title="Light Amplification by Stimulated Emission of Radiation"
  >LASER</abbr>
  beams attached to their heads.
</p>
</body>
</html>
```

Once the latest CSS specification solidifies, the browsers will be able to follow suit and implement true dotted underlines, and the recipe above will become obsolete.

Fancier Blockquotes

The `blockquote` element is for quoted text that consists of one or more full paragraphs. By default, blockquote indents everything to the right. In the past, people abused `blockquote` (and `ul` and `ol`) to simply indent text. But there are much better ways to indent text, and so it's best to use `blockquote` for its original purpose: quoting paragraphs of text.

Let's say that simply indenting the text is too plain. We'd like to add some color. In Example 3.23, "Blockquotes", we change the appearance of the `blockquote` element step-by-step until we achieve the desired effect.

Example 3.23. Blockquotes

view htmlview plainprint?

```
<html>
<head>
  <style>
    blockquote { background: #ccccff; }
    blockquote.padded { padding: 3px; }
    blockquote.bordered { padding: 3px; border-left: 3px #0000cc solid; }
    blockquote.bordered p { margin: 10px; }
  </style>
</head>
<body>
  <blockquote>
    <p>"I also have a paper afloat, with an electromagnetic theory of light,
    which till I am convinced to the contrary, I hold to be great guns."</p>
  </blockquote>
  <p>- James Clerk Maxwell</p>
```

```

<blockquote class="padded">
  <p>"An expert is a person who has made all the mistakes that
  can be made in a very narrow field."</p>
  <p>"Your theory is crazy, but it's not crazy enough to be true."
</blockquote>
<p>- Niels Bohr</p>

<blockquote class="bordered">
  <p>[As a student, attending a lecture by Einstein]
  "You know, what Mr. Einstein said is not so stupid..."</p>
  <p>[On his wife leaving him for a chemist]
  "Had she taken a bullfighter, I would have understood, but an
  ordinary chemist..."</p>
</blockquote>
<p>- Wolfgang Pauli</p>
</body>
</html>
```

1. `blockquote { background: #ccccff; }`

   First, we try setting the background to pale blue. Unfortunately, since the blockquote has no `padding`, the colored space looks crowded.

2. `blockquote.padded { padding: 3px; }`

   Next, we'll try fixing this up by creating a new class, `padded`. This class inherits the background color assigned to all `blockquote`s, and includes a little extra padding so that the child paragraphs don't look so crowded. Whoops! A whole lot of extra space appears above and below the paragraphs. This looks even worse. What happened?

   Adding padding to `blockquote` has triggered an interesting side-effect of the rules for collapsing margins. As soon as you add padding or borders to the parent `blockquote` element, margin collapsing goes away. This causes the paragraph's margins to suddenly push out the `blockquote` above and below.

3. `blockquote.bordered { padding: 3px; border-left: 3px #0000cc solid; }`

   Let's try one more time. We'll create another class, `bordered`, that includes the padding and adds a decorative dark blue border to the left side of the blockquote.

4. `blockquote.bordered p { margin: 10px; }`

   Finally, we'll try to account for the problem introduced by the padding. The selector `blockquote.bordered p` directs the browser to "apply this style to all `p` elements that are inside a `blockquote` with a class of `bordered`." As a quick fix, we'll assign these paragraphs a uniform margin of 10px. This pushes the paragraphs out to the right, and provides a more uniform spacing around each paragraph. The spacing between each paragraph is 10px, while the spacing between the `blockquote` and the paragraphs is 13px. To make the spacing even more uniform, you could reduce the `blockquote` padding to 1px, or play a few more tricks with margins and padding, but let's move on.

Align and Indent

You've probably noticed that most of your HTML paragraphs don't have an initial indentation. You can use `margin` or `padding` to push paragraphs to the right, but that affects the entire paragraph, not just the first line. Furthermore, text is always left-aligned. How do we make centered text, or right-aligned text?

As we saw in the “Borders” section, the box model controls the area around the text. This section shows how to move the text around inside the box itself.

Indenting Text

To indent text, use the handily-named `text-indent` property. The following CSS snippet adds a 2.0 em indent to all paragraphs:

Example 3.24. Indented Paragraphs

view htmlview plainprint?

```
<html>
<head>
  <style>
    p {
      text-indent: 2.0em;
    }
  </style>
</head>
<body>
 <p>
   "On the contrary," said Holmes, quietly, "I have every reason
   to believe that I will succeed in discovering Mr. Hosmer Angel."
 </p>
 <p>
   Mr. Windibank gave a violent start, and dropped his gloves.
   "I am delighted to hear it," he said.
 </p>
 <p>
   "It is a curious thing," remarked Holmes, "that a typewriter
   has really quite as much individuality as a man's handwriting.
   Unless they are quite new no two of them write exactly alike.
   Some letters get more worn than others, and some wear only on
   one side. Now, you remark in this note of yours, Mr. Windibank,
   that in every case there is some little slurring over the e, and
   a slight defect in the tail of the r. There are fourteen other
   characteristics, but those are the more obvious."
 </p>
</body>
</html>
```

If you enter a negative value for `text-indent`, the first line of the text will shift to the left. This is useful for creating “hanging indents” for headings and the like. The only trick is that you must set the padding of containing block accordingly.

Example 3.25. Hanging Indent

view htmlview plainprint?

```
<html>
```

```
<head>
  <style>
    div {
      padding-top: 5px;
      padding-bottom: 5px;
      padding-right: 5px;
      padding-left: 30px;
      border: 3px solid;
    }
    h2 {
      text-indent: -25px;
    }
  </style>
</head>
<body>
  <div>
    <h2>The Red-Headed League</h2>
    <p>
      I had called upon my friend, Mr. Sherlock Holmes, one
      day in the autumn of last year, and found him in deep
      conversation with a very stout, florid-faced elderly
      gentleman, with fiery red hair. With an apology for my
      intrusion, I was about to withdraw, when Holmes pulled
      me abruptly into the room and closed the door behind me.
    </p>
  </div>
</body>
</html>
```

The "container" `div` has a uniform padding of 5px all around, except for the left padding, which is 30px. The `h2` element has its text-indent set to negative 25px. The result is that the `h2` shifts 30px to the right due to the padding, and then shifts 25px back to the left due to the negative `text-indent`. All other elements are 30px from the left edge. The result is a 25px hanging indent.

Horizontal Alignment

The `text-align` property allows you to align text to the right, left, or center within its containing block. The four options are:

- `left`: Aligns the text along the left edge of the box, leaving a ragged edge to the right. This is the default.
- `center`: Centers the text within the box.
- `right`: Aligns the text along the right edge of the box, leaving a ragged edge to the left.
- `justify`: Aligns the text along both the right and left edges of the box. This can cause the word spacing in the line to become uneven. The `justify` option isn't as well-supported in some older browsers.

Let's take a look at all four options:

Example 3.26. Left, Center, Right, and Justified

view htmlview plainprint?

```
<html>
<head>
  <style>
    div { border: 3px solid; padding: 5px; }
    div.left { text-align: left; background: #ffcccc; }
    div.center { text-align: center; background: #ccffcc; }
    div.right { text-align: right; background: #ccccff; }
    div.justify { text-align: justify; background: #ffffcc; }
  </style>
</head>

<body>
  <div class="left"><em>Left-aligned text.</em></div>

  <div class="center"><em>Centered text.</em></div>

  <div class="right"><em>Right-aligned text.</em></div>

  <div class="justify"><em>Justified text.</em> Upon this a question
  arises: whether it be better to be loved than feared or feared than
  loved? It may be answered that one should wish to be both, but, because
  it is difficult to unite them in one person, it is much safer to be
  feared than loved, when, of the two, either must be dispensed with.
  Because this is to be asserted in general of men, that they are
  ungrateful, fickle, false, cowardly, covetous, and as long as you
  succeed they are yours entirely; they will offer you their blood,
  property, life, and children, as is said above, when the need is far
  distant; but when it approaches they turn against you.</div>
</body>
</html>
```

Keep in mind that `text-align` aligns text and other inline content *within* boxes. It does not affect blocks themselves. For example, you can't use `text-align: center` to center a smaller box within a larger box. This sort of thing is part of a more advanced concept called CSS positioning.

Tables

Tables enable you to arrange HTML elements in a grid. Examples of tabular data include calendars, project resource assignments, scientific data sets, and other types of data that make sense to display in rows and columns.

Note

Since tables can arrange HTML elements in a grid, one obvious usage for tables is to define the layout of a web page. This ancient layout technique is now superseded by CSS-based layouts. For brevity, this tutorial only focuses on tables that display tabular data.

When you're done with this section, you should be able to:

- Create a basic table with cells containing tabular data
- Add captions and summaries
- Change the padding, margins, and borders of table cells using CSS

- Define more structured tables with column groups, footers, and headers

Section Summaries

The Tables chapter contains these sections:

- Table Structure — Explains how to construct a simple table by adding rows, columns, and captions.
- Cellpadding and Cellspacing — Explains how to control the borders and spacing around table cells using the `border`, `cellpadding`, and `cellspacing` attributes.
- Tables and CSS — Explains how to control the borders and spacing around table cells using CSS.

Table Structure

Even a simple table involves several levels of nested elements. A table's basic structure is:

- The `table` element defines the table itself.
- Within the `table` are one or more `tr` elements that define table rows.
- Within the `tr` elements are one or more `th` or `td` elements. These elements define table header cells and table data cells, respectively.

Here's a simple example of a table with one row and two columns. To better illustrate the structure of the table, we've added a snippet of CSS to provide each table cell with a border.

Example 4.1. 1×2 Table

view htmlview plainprint?

```
<html>
<head>
  <title>1x2 Table</title>
  <style>
    td { border: 1px solid #000000; }
  </style>
</head>
<body>
  <table>
    <tr>
      <td>California</td>
      <td>Michigan</td>
    </tr>
  </table>
</body>
</html>
```

Adding Rows and Columns

A `tr` element represents a row. To add another row, just add another `tr` element with the same number of `td`s (columns) as the rest of the table:

Example 4.2. 2×2 Table

view htmlview plainprint?

```
<table>
<tr>
<td>California</td>
<td>Michigan</td>
</tr>
<tr>
<td>Nevada</td>
<td>Ohio</td>
</tr>
</table>
```

A `td` represents a table data cell. To add another column, add another `td` element to each `tr` in the table:

Example 4.3. 2×3 Table

view htmlview plainprint?

```
<table>
  <tr>
    <td>California</td>
    <td>Michigan</td>
    <td>Mississippi
  </tr>
  <tr>
    <td>Nevada</td>
    <td>Ohio</td>
    <td>Alabama</td>
  </tr>
</table>
```

Adding Headers

So far, we've seen table rows containing `td` (table data) elements. However, rows can also contain `th` (table header) elements. Table headers label a row or column. By default, most browsers render table headers as bold.

Example 4.4. Table with Headers

view htmlview plainprint?

```
<table>
<tr>
    <th>West</th>
    <th>Midwest</th>
    <th>South</th>
  </tr>
  <tr>
    <td>California</td>
    <td>Michigan</td>
    <td>Mississippi
```

```
</tr>
<tr>
  <td>Nevada</td>
  <td>Ohio</td>
  <td>Alabama</td>
</tr>
</table>
```

Tip

If you leave any table data or table header empty (`<td></td>`), the cell will collapse and look unattractive. To prevent this from happening, you can place a special “non-breaking space” entity, `&nbsp;`, in the cell. This inserts an invisible character that forces the browser to draw the borders of the cell.

Adding Captions and Summaries

You can provide additional information about a table by giving it a caption or a summary.

The `table` element’s `summary` attribute provides a short text summary of the table. Most browsers do not display the summary directly, but screen readers and text-only browsers can provide this text to their users. If you hover your mouse over the table, some browsers will display the summary text as a tooltip.

The `caption` element, which must immediately follow the `<table>` open tag, provides the table’s title. By default, the caption appears directly above the table. Some browsers display the caption centered above the table, others display it left-aligned. You may use CSS to align and style the caption accordingly.

Table Rows, Cells, and Headers

Once you have placed the (optional) `<caption>`, it’s time to start constructing the actual table. Tables are composed of table rows (`<tr>`). Each pairing `<tr>...</tr>` defines a row.

Within each table row are one or more cells, which are either table data (`<td>`) or table headers (`<th>`). Table header cells are supposed to contain “header” information, such as the title of a row or colum. Table data cells are supposed to contain the data. Note that if you leave any table cell completely empty (”`<td></td>`“), the cell will collapse and look unattractive. It’s best to put something in a blank cell, even if it’s just a `<br>` element.

Let’s take a look at an example that incorporates all of these elements. The previous table had one row and two columns. The following table has four rows and four columns.

Example 4.5. 4×4 Table

view htmlview plainprint?

```
<table cellpadding="5" border="2"
summary="All 12 months, organized by season">
  <caption>The Four Seasons</caption>
  <tr>
    <th>Spring</th>
    <th>Summer</th>
    <th>Fall</th>
    <th>Winter</th>
  </tr>
  <tr>
    <td>March</td>
    <td>June</td>
    <td>September</td>
    <td>December</td>
  </tr>
  <tr>
    <td>April</td>
    <td>July</td>
    <td>October</td>
    <td>January</td>
  </tr>
  <tr>
    <td>May</td>
    <td>August</td>
    <td>November</td>
    <td>February</td>
  </tr>
</table>
```

As you can see, even a simple 4×4 table takes a fair amount of coding.

A few things to consider:

- Look carefully back and forth between the code and the results, and make sure that you understand how each month ends up in the proper row and column. Do you understand why there is one row of seasons and three rows of months? Do you understand why each column of months lines up the way it does?
- The `<th>` text is bold, while the `<td>` is plain. This is the default font weight for these elements, although of course you can change this with style sheets.
- The summary attribute does not appear anywhere on the screen. It's really meant for non-graphical browsers, but we'll keep being a good citizen and continue to put it in.

Cellpadding and Cellspacing

The `<table>` element has three attributes that provide some basic control over the spacing around table cells: `border`, `cellpadding`, and `cellspacing`. These three table attributes are similar to the CSS properties `border`, `padding`, and `margin`. Let's see how they work.

First we'll create a table with no borders, cellpadding, or cellspacing. In order to make things easier to see, we'll use CSS to color the background of the table (green) and the individual table cells (yellow).

Example 4.6. Table: No Spacing

view htmlview plainprint?

```
<html>
<head>
  <style>
    table { background: #009900; }
    td { background: #ffff00; }
  </style>
</head>
<body>
  <table
    border="0" cellpadding="0" cellspacing="0"
    summary="No padding, spacing, or borders">
    <tr>
      <td>Left Cell</td>
      <td>Middle Cell</td>
      <td>Right Cell</td>
    </tr>
  </table>
</body>
</html>
```

With no border, cellspacing, or cellpadding, our three cells are so cramped that we can’t see the green background of the table — just the yellow background of the individual cells. Let’s try setting each of these attributes to `"5"` five pixels) separately. We’ll maintain the same green and yellow coloring scheme.

Example 4.7. Table: 5px Spacing

view htmlview plainprint?

```
<html>
<head>
  <style>
    table { background: #009900; }
    td { background: #ffff00; }
  </style>
</head>
<body>

  <h4>5px Border</h4>
  <table
    border="5" cellpadding="0" cellspacing="0"
    summary="Table with a 5px border">
    <tr>
      <td>Left Cell</td>
      <td>Middle Cell</td>
      <td>Right Cell</td>
    </tr>
  </table>

  <h4>5px Cellpadding</h4>
  <table
    border="0" cellpadding="5" cellspacing="0"
    summary="Table with 5px cellpadding">
    <tr>
      <td>Left Cell</td>
      <td>Middle Cell</td>
      <td>Right Cell</td>
    </tr>
  </table>

  <h4>5px Cellspacing</h4>
  <table
    border="0" cellpadding="0" cellspacing="5"
    summary="Table with 5px cellspacing">
    <tr>
      <td>Left Cell</td>
      <td>Middle Cell</td>
      <td>Right Cell</td>
    </tr>
  </table>

</body>
</html>
```

What are the results?

- In the first table, `border="5"`. We see a five-pixel border around the edge of the table.
- In the second table, `cellpadding="5"`. The size of each cell has expanded by five pixels in every direction.
- In the third table, `cellspacing="5"`. Each cell is separated from its neighbors by five pixels. This spacing enables us to see the green background of the table itself.

Finally, let's set all three attributes. We'll set the border to 5px, the cellspacing to 10px, and the cellpadding to 20px.

Example 4.8. Table: Variable Spacing

view htmlview plainprint?

```
<html>
<head>
  <style>
    table { background: #009900;}
    td { background: #ffff00; }
  </style>
</head>
<body>
  <table
    border="5" cellpadding="20" cellspacing="10"
    summary="Table with a 5px border,
      20px cellpadding, and 10px cellspacing">
    <tr>
      <td>Left Cell</td>
      <td>Middle Cell</td>
      <td>Right Cell</td>
    </tr>
  </table>
</body>
</html>
```

The ridged border around the table is five pixels wide, the cells are ten pixels apart, and the cells themselves have twenty pixels of padding. Some browsers will display the border colored with the background color, green; others will leave the ridged border colorless.

Tables and CSS

For more sophisticated control over borders, spacing, and padding, you can and should use CSS rather than the `border`, `cellspacing`, and `cellpadding` attributes. The CSS properties are as follows:

- The `padding` CSS property corresponds to the `cellpadding` table attribute.
- The `border-spacing` CSS property corresponds to the `cellspacing` table attribute.
- The `border` CSS property corresponds to the `border` table attribute.

The CSS properties allow us to use any valid CSS length: `px` are okay, as are `em`, `pt`, and so on. We'll stick with pixels for our example below.

Example 4.9. Table: CSS Borders

view htmlview plainprint?

```
<html>
<head>
  <style>
    table {
      background: #009900;
      border: 4px #ff00ff ridge;
      border-spacing: 5px;
    }
    td {
      background: #ffff00;
      padding: 10px;
      border: 2px #0000ff dotted;
    }
    td.middle {
      border: 2px #ff0000 solid;
      padding-right: 25px;
    }
  </style>
</head>
<body>
  <table
    border="0" cellpadding="0" cellspacing="0"
    summary="Table with CSS borders">
    <tr>
      <td>Left Cell</td>
      <td class="middle">Middle Cell</td>
      <td>Right Cell</td>
    </tr>
  </table>
</body>
</html>
```

The overall table has a green background and a ridged, lavender border. The result is garish to say the least… presumably your tables will be far more subdued and tasteful.

Ordinary table cells have a yellow background, ten pixels of padding, five pixels of border-spacing (i.e. cellspacing), and a blue, dotted border.

Table cells with `class="middle"` have an extra fifteen pixels of padding to the right and a red, solid border.

Note that some modern browsers still don't support the `border-spacing` attribute. If you're using one of these browsers, all three cells will be adjacent to each other, and you will not see the green background of the table itself. One workaround is to use CSS to do your primary styling, and the `border` table attribute for backup.