Frontend Engineering II: Pro CSS

Print Preview

The Frontend Engineering Series

I: Pro HTML

II: Pro CSS

III: JavaScript with Native Objects

IV: JavaScript with DOM Objects

V: JavaScript with User Objects

Frontend Engineering II: Pro CSS

CSS 1
CSS 2.1
CSS 3

Martin Rinehart

Frontend Engineering II: Pro CSS

Knowit Publishing, LLC

Hudson Valley, New York, USA

Designed and Written in LibreOffice Writer, Published to PDF, all in the USA.

Printed as near to our customers as our technology permits.

Table of Contents

Introduction

CSS is a “declarative” language with which you add styles to your HTML. Declarative computer languages let you describe what you want done without the step-by-step, how-to-do-it instructions that computers need to actually do something. “I'd like a solid blue border, five pixels wide, around this paragraph,” is a declarative statement. In CSS that's

```
border: solid blue 5px;
```

The original dream behind CSS was to help the World-Wide Web graduate from a system for sharing academic papers to a system capable of displaying pages just as attractive as magazine pages. CSS brings that goal within reach. And far more.

While you learn to style your pages with CSS you'll work on upgrading the HTML-only site you built in *Volume I: Pro HTML* to a stylish, 21st century site.

Before we're done we'll explore some CSS 3, learning how to add two types of animation to our pages. This is far beyond the early dreams of “magazine-quality” presentation, approaching the field of “television-quality” presentation. In fact, it goes beyond television, because the viewer is an active, not passive, participant.

Along the way, we'll leave the 20th century's single focus on desktop computers. Instead we'll enter the 21st century world of smartphones, tablets and monitors of all sizes. Your website project will look good on every device bigger than a wristwatch and smaller than a 4K monitor.

Enjoy the journey!

1 Write Your First CSS

This is a “knowit” (knowledge unit). That means (in this case) that it has an online part, as well as this print part. The online part is at:

MartinRinehart.com

We'll point to locations in the online work with a “drilldown”:

```
Online: Knowits > CSS II > Welcome
```

You click “Knowits” on the first menu at MartinRinehart.com. The second line of menus appears (if it was not already there). You click “CSS II,” and click “Welcome” on the third menu. The “Welcome” screen explains the knowit. Skip it if you are a veteran of *Pro HTML*.

If you are not a veteran of *Pro Html*, this knowit continues right now at

```
Online: Knowits > CSS II > Welcome
```

Warning: If you want to learn to write CSS you must follow the drilldowns to the online portions. They include specific code information and coding assignments. Just reading this text is no more likely to succeed than just reading *How to Ride a Bicycle*. To learn to ride a bicycle, you have to sit on the seat and pedal. To write CSS, you have to get your fingers on your keyboard and write CSS.

Knowit Recap

A knowit, knowledge unit, can combine media. This one combines print and online. Print is easier to read. (There are about a quarter million "pixels" in a square centimeter of this print. There are about 800 pixels in a square centimeter of a typical desktop monitor.) But putting your mouse on the printed page and clicking is not going to get you anywhere. Both media have advantages, so we use both.

Now, off to the Companion page, the start of the knowit for this chapter:

```
Online: Knowits > CSS II > Online > 1
```

(We hope you noticed that if "Knowits" and "CSS II" were already punched, it took just two clicks to get to "Online" and then "1". A great goal of all frontend engineering: make the visitors job simple!)

Now, let's get started. Go Online 1a will help you create a template if you don't have one already.

```
Online: Knowits > CSS II > Online > 1 > 1a
```

Then we'll begin with a few words about the history of CSS before we dive into styling with Cascading Style Sheets.

History

HTML was barely out of diapers when people discovered the need for style sheets. (To be more exact, they discovered that repeatedly adding the same attributes to every heading in a document, for example, was tedious work. The kind of work for which computers were better suited than were people.)

Håkon Wium Lie, of Opera, and Bert Bos (author of an early browser), who would later join the W3C and chair the CSS working group, implemented an experimental style sheet language to show what could be done. This language was proposed as an adjunct to HTML as early as 1994, just five years after Tim Berners-Lee invented the World-Wide Web and HTML.

This started what was to become CSS on the very bumpy road that would lead it, today, to being a comfortable standard from which we all benefit.

Cascading Style Sheets

CSS is an abbreviation for “cascading style sheets.” It occasionally refers to style sheets in a cascade. More often it refers to the styles that can be specified on such sheets. They can be specified in, among other places, your “markup.”

The “markup” refers to your HTML files which, if you started with a finished written document, not for the web, and then added tags, could be said to have been “marked up.”

What Are Styles?

The basic idea of CSS is that you start with content in your HTML (semantics) and then use style rules to specify the finished look of the document (presentation). Style rules specify fonts (sizes, weights, families), colors, layout and more. Style rules are also available for aural and other non-visual documents, though these are beyond the scope of this knowit.

Rules

Note: When you refer to CSS "rules," as we do here, you exclude CSS's "at-rules" (they start with an "@" character). We introduce at-rules in Chapter 6.

A CSS rule has two parts:

- A "selector" choosing the element(s) to which it applies.
- A "declaration" that specifies the styling to apply.

```
selector { declaration }
```

The declaration is wrapped in braces (not parentheses) as shown.

```
Braces: { } Parentheses: ()
```

The declaration also has two parts:

- A "property" name.
- The property's value.

The following rule would provide a light brown background for all your `<h2>` headings:

```
h2 { background-color: wheat }
```

Values are often the same as the values you use for HTML attributes. The color `wheat` is also the color `#f5deb3`. Unlike HTML attribute values, you do not use quotes around the values in CSS except when a single value includes space characters:

```
font-family: 'Times New Roman'
```

Places for Rules

CSS rules go in the `style` attribute of any HTML tag, or in separate style sheets. (The `style` attribute is universal in HTML 4.01, global in HTML 5.)

Styling a Single HTML Element

This is a style applied to a single heading:

```
<h2 style='background-color: wheat'>
     Whole Wheat for Healthy Eating
</h2>
```

The `style` attribute's value is enclosed in quotes (it is an HTML attribute value and includes a space). The `wheat` value of the CSS property `background-color` is not quoted.

Try inline styles on your own. Go Online 1b explains how:

Online: Knowits > CSS II > Online > 1 > 1b

An Embedded Style Sheet

Styling a document one element at a time gets tiresome very quickly (as was discovered in the early days of HTML). Also, if you style every heading differently, your site's visitors would get tired, very quickly. The easy way, and the best, is to create a style sheet that specifies how elements are styled. Embedded style sheets (there may be more than one) go in your HTML's `<head>` section, as Listing 1-1 shows.

Listing 1-1

```
<head>
    <style>
        h1 { background-color: wheat}
        h2 { background-color: wheat}
        h3 { background-color: wheat}
    </style>
</head>
<body>...
```

As with HTML, the use of whitespace, such as the indenting shown in Listing 1-1, is entirely optional. Our format has evolved to make our style sheets easy to read and easy to modify. To try it yourself, add an

embedded style sheet into your own page, and style all the paragraph elements as Go Online 1c shows.

Online: Knowits > CSS II > Online > 1 > 1c

External Style Sheets

Style sheets may also be placed in their own files. If your website has multiple pages, this is the technique that lets you provide a single style sheet that every page will use. You incorporate the style sheet with a `<link>` tag in the `<head>` section. We'll show the details later in this chapter.

Grouping Rules

To make your job simpler (and to encourage good styling) you may group multiple selectors in front of a single declaration, this way:

```
h1, h2, h3 { background-color: wheat }
```

And you may use multiple declarations in a single declaration block, separating them with semicolons, as in Listing 1-2:

Listing 1-2

```
h1, h2, h3 {
    background-color: wheat;
    color: SaddleBrown;
}
```

(`SaddleBrown` is a dark brown. The `color` property is the color of the text.)

An extra semicolon following the last declaration is permitted and we always use it (which means we never forget to add the necessary semicolon when we add another declaration in a declaration block).

Selectors

We've shown HTML tags as selectors. In CSS these are called “type selectors.” This is only one of a wide range of possibilities. We'll present four more here; that will be enough for this chapter's work.

Class Selectors

Another HTML attribute that is universal (and global) is the `class`. If you want some elements of your page to be a bit larger than most you could create an `oversize` class. If you want some elements to use blue type (the default is black) you could create a `blue_text` class. Your markup could look like Listing 1-3.

Listing 1-3

```
<p>Plain paragraph.</p>
<p class='oversize'>
     Larger type here.</p>
<p class='blue_text'>
     Blue type here.</p>
<p class='oversize blue_text'>
     Larger, blue text!</p>
```

Without CSS (or JavaScript) a class designation is meaningless. With CSS you can make classes real. You precede the class name with a period in your style sheets, as Listing 1-4 shows.

Listing 1-4

```
<style>
    .blue_text { color: blue; }
    .oversize { font-size: 1.2em; }
</style>
```

(Not familiar with the `em` size? From old typography, an em was the width of the letter “M” in the font in use. In CSS it specifies size relative to the current font. `1.2em` is twenty percent larger than the current size.)

Try Go Online 1d for using classes:

Online: Knowits > CSS II > Online > 1 > 1d

ID Selectors

Another universal/global HTML attribute is the ID. That is a unique identifier (you are responsible for ensuring that it is unique) you assign to a single element. We use ID selectors constantly and the result is much more readable than doing the same job with inline styles.

To use an ID in the style sheet, you precede it with a number sign: "#". This is commonly called a "hash" and sometimes "pound sign." We prefer "hash" as our British cousins might think of a different symbol (£) when one says "pound sign."

Listing 1-5 shows two elements styled in an embedded style sheet.

Listing 1-5

```
<head>
    <style>
        #foo {
            /* styles for 'foo' here */
        }
        #bar {
            /* styles for 'bar' here */
        }
    </style>
</head>
<body>

<p id='foo'>Foo styles for me.</p>

<p id='bar'>Bar styles here.</p>
```

In Go Online 1e we style a link (which will be needed in the next section.)

Online: Knowits > CSS II > Online > 1 > 1e

Pseudo-Class Selectors

Now we get to another selector alternative used regularly in places such as navigation menus: the pseudo-class `:hover`. Unlike other classes, it is chosen by user action. When the user moves the mouse pointer over an element, the element joins the `:hover` pseudo-class (until the mouse pointer leaves).

A common way to style links to other pages in a navigation menu is to change the cursor from the default arrow to the pointing finger. These are values `default` and `pointer` of the `cursor` property. You might want to change the links' background color from its default (possibly the page background) to white, to emphasize the fact that the mouse is over an active element. Listing 1-6 shows a common treatment for navigation links.

Listing1-6

```
<style>
    .nav {
        cursor: default;
        /* other styles here */
    }
    .nav:hover {
        background-color: white;
        cursor: pointer;
    }
```

The viewer's browser will change style from `nav` to `nav:hover` automatically as the mouse pointer moves over and away from elements in the `nav` class. (Warning: this is great for desktops and laptops, but smaller devices use finger gestures, not mice.)

Descendant Selectors

This is a selector with a fancy name, but a simple meaning. If one element is enclosed in another, it is called a “child” element. If a child element has its own children, the child, its children, the children's

children and so on, are all called “descendants” of the first element. (You may still see the older, CSS 2 term, “contextual selector” used for descendant selectors.) In our project site, we have the menu built as Listing 1-7 shows.

Listing 1-7

```
<div id="nav" align='left'>
    <font ...> Home </font> ...
    <a ...> Timeline </a> ...
    <a ...> Map </a> </a> ...
    <a ...> Galileo </a> ...
    . . .
</div> <!-- nav →
```

Note that the `<a>` elements are all descendants of a `<div>` element. A CSS selector to choose such elements is:

```
div a { /* declarations here */ }
```

Warning, `div a` (separated by a space) selects the `a` elements that are descendants of a `div` element. `div, a` (with a comma) selects all `div` and all `a` elements. Watch for those commas!

Some CSS declarations have shortcuts. You could set border properties, as Listing 1-8 shows:

Listing 1-8

```
selector {
    border-color: blue;
    border-style: solid;
    border-width: 3px;
}
```

You can also save some typing and get the same result the `border` shortcut, this way:

```
selector { border: blue solid 3px; }
```

(The property values in a shortcut may be in any order. `solid` is a border style. It is not a color or length.)

This is one of those things that is more trouble to explain than it is to do. We like things that are easy to do. Giving all the links inside the nav div an outset border can be done with this line:

```
#nav a { border: blue outset 2px; }
```

That selects all the a elements that are descendants of the element whose ID is `nav` and sets their borders. You'll put this into practice in your project, which comes next.

Project

Are you a veteran of *Pro HTML*, the first volume in this series? If you are, the project work here adds styles to your sample project. If you are not, you will have to create (or use an existing) sample, HTML-only project Either way, be sure you save your project before you begin adding styles. You will want to go back later to compare the styled to the unstyled version.

If you have a project from *Pro HTML,* proceed immediately to Go Online 1f:

Online: Knowits > CSS II > Online > 1 > 1f

If you came here from another background, you'll need to create a sample project. This is not as hard as it might sound. Visit our sample, here:

Online: Knowits > CSS II > Project

You'll need a similar information-only website for the project examples here. For starters, create a home page with links to a few subordinate pages. Let's assume you are a fan of Jane Austen's novels. Start with a Jane Austen page and then create a page for each of her novels. The

novel pages can be as simple as an `<h1>` heading with the novel title, and either your cogent essay re Fanny Price as the heroine of *Mansfield Park* or, if you are pressed for time, some *lorem ipsum.*

The important part, for our purpose, is that each page have a menu of links to the other pages at the top. See the project in the HTML volume for what yours should look like:

Online: Knowits > HTML I > Project

As we go along, add features to your project to roughly match our sample, before styles (HTML only) and then style them.

With your sample project, and it's very simple navigation links, you are ready to go on to add an external style sheet, as shown in Go Online 1f.

Online: Knowits > CSS II > Online > 1 > 1f

Quiz

Choose the word or phrase that best completes each sentence.

1) CSS stands for
   a) Corinthian Styling System.
   b) Cascading Style System.
   c) Cascading Style Sheets.

2) CSS was invented
   a) when the World-Wide Web was five years old.
   b) when the Internet bubble burst.
   c) when Google created the Chrome browser.
   d) to create HTML element attributes.

3) CSS rules have
   a) two parts, a property name and a value.
   b) two parts, a custom and a valuation.
   c) a head and a body.
   d) none of the above.

4) CSS rules may be placed
   a) inline, out-of-line or in style sheets.
   b) inline or in embedded or external style sheets.
   c) in cascading or flat-water styles.

5) Style sheets may group
   a) selectors.
   b) declarations.
   c) both of the above.

6) Class selectors
   a) may be separated by commas.
   b) segregate HTML tags by the tags' “value.”
   c) apply only to HTML 4 and 4.01.
   d) are used to identify specific HTML elements.

7) ID selectors
   a) are preceded with dollar signs in the style sheet.
   b) are preceded with pound signs in the style sheet.
   c) are preceded with hash marks in the style sheet.

8) Pseudo-class selectors
   a) may always be determined in advance.
   b) may only be determined during operation.
   c) may be determined by the mouse position.

9) Descendant selectors
   a) eliminate the elements that “inherit” from a parent.
   b) apply to elements that are not direct child elements.
   c) do not include sibling elements.
   d) include sibling elements.

10) HTML tag selectors
   a) cannot be descendant selectors.
   b) must have matching ID values.
   c) may be part of descendant selectors.
   d) must follow class selectors.

2 The Box Model

Since CSS 1, our style sheets have been built on boxes. Every tag that creates something you see on your screens creates a box. Every box may be styled with CSS, whether it be a paragraph of text, an image or a heading. When you added padding and borders in Chapter 1 you were working with the CSS box model. We'll take a quick look at how this came about, and then dive in.

We'll present dozens of new properties, but you'll find them so nicely organized, so regular, that your memory will not be challenged.

History

By 1994, Lie and Bos had a browser with support for their CHSS (Cascading HTML Style Sheets) language. The “H” would be dropped when it was seen that there was no reason to limit the language to just supporting HTML. (One fears this was a triumph of ambition over focus.)

The W3C (headed by Berners-Lee) decided that, in general, this was a good idea and formally charged the HTML working group with working on the project. Lie and Bos were the technical leaders, so it is no surprise that CSS emerged (and not competing specifications, such as Netscape's JSSS—JavaScript Style Sheets).

Moving quickly, as the proposal did not have the disadvantage of numerous implementations and users, the first CSS standard, CSS 1, was completed in late 1996.

When is a standard not a standard? When it is not followed. Browser vendors were slow to adopt CSS. Those that did adopt it did so only partially. The poor frontend engineer who wanted to use style sheets could not count on pages looking the same in any two browsers (either different vendor's browsers, or different versions of a single vendor's browser).

It would be years before style sheets were a practical reality in the frontend engineers' world, but CSS 1 is still very much with us today. Its properties for styling fonts, for example, are the ones that we use every day. Among its best achievements is the definition of the CSS box model, which is this chapter's subject.

Box Areas

Almost every HTML tag creates a box. The few that don't are excluded for sensible reasons. Our `<link>` and `<style>` tags in the `<head>` are examples of elements without boxes—the viewer cannot see them. In the `<body>`, the `<br>` tag does not have a box. (The text into which it inserts a break has boxes. You can see the text.)

Boxes have three or four areas (clarification coming shortly) that you work with regularly. Figure 2-1 shows the basic idea.

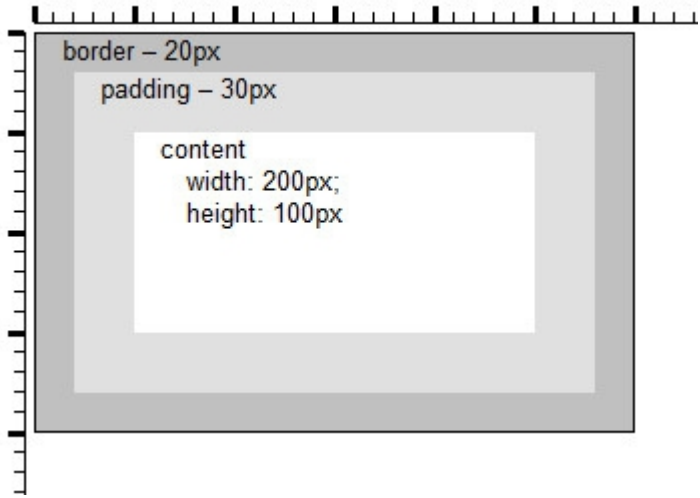


Figure 2-1

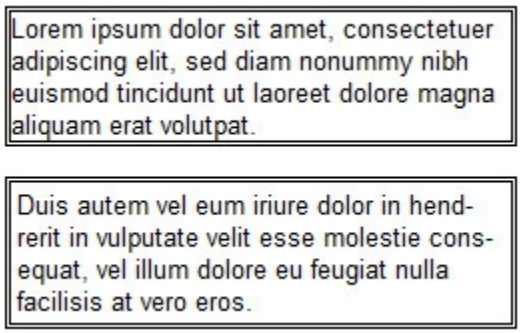
Content

The content area is the container for text, images, headings or whatever you need. Note that content may or may not fit. Commonly, scroll bars will appear as needed. Note that Figure 2-1 shows 200×100 pixels of content. This is the CSS size of the box, though the outside of its borders measure 300×200 pixels.

Padding

Boxes, especially boxes with borders, need a bit of padding around their content. Figure 2-2 shows text without padding and with padding.

Figure 2-2



The second box has three pixels of padding all around. You see a dramatic difference between the left-hand sides of the two boxes.

Note that the first box is 250 pixels wide. To make them look the same, the second box is 244 pixels wide (plus three pixels padding, left and right).

Border

The boxes in Figure 2-2 have `border-style: double` borders. You've already seen the `border-width` and `border-color` properties in use, as well as the `border` shorthand property. The boxes in Figure 2-2 are `border: black double 3px`.

This is a good time to look at all the values of the `border-style` property, in code, shown in Go Online 2a:

Online: Knowits > CSS II > Online > 2 > 2a

Be aware that browsers, especially MSIE, take liberties with your colors as they darken/lighten to show the effect of shadows. As with HTML, view the results in all the browsers you want to support.

Margin

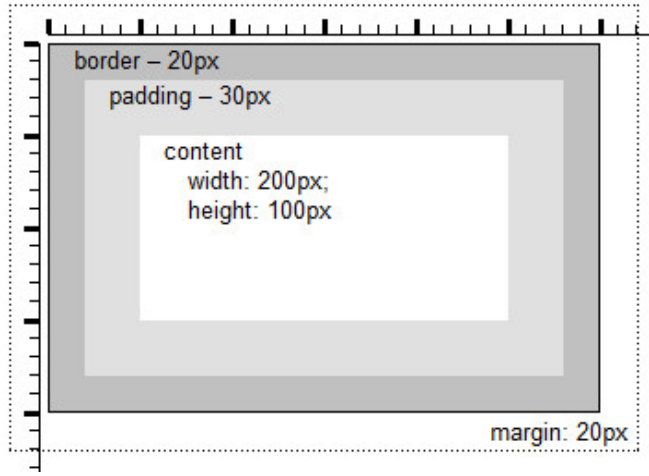


Figure 2-2

Figure 2-2 illustrates the margin that is the fourth of our "three or four" box areas. You may consider margins a box area, or you may claim they are clearly outside the box. We will agree with you either way. Margins cover nothing, letting the background show. You use them to provide spacing between your elements.

Horizontal Margins

Horizontal margins do not collapse. If the right side of one box has a 20-pixel margin, and the left-side of an adjacent box has a 10-pixel margin you get 30 pixels between the boxes.

Vertical Margins

Vertical margins are collapsed. A 20-pixel margin above or below a 10-pixel margin will combine, leaving only a 20-pixel margin.

Properties for Four Sides

So far we have used declarations like `padding: 3px`. This is the common usage, specifying padding on all four sides. We could also specify padding on the left side with `padding-left`. You can use `-top`, `-right` and `-bottom` to specify the other sides.

This should be good news if you were wondering how to remember so many style properties. You knew one, `padding`, and now you know five: `padding`, `padding-left`, `padding-top`, `padding-right` and `padding-bottom`.

Similarly, there are five possibilities for `margin`. Better yet, you have `borderX` (where X is `-top`, `-right`, `-bottom`, `-left` or omitted) and `borderXY` (where Y is one of `-color`, `-style`, `-width` or omitted). That gives you 20 different properties for just the `border`.

You can also specify one or four values for these properties: `border-color: red;` or `border-color: red blue cyan yellow;`. One color applies to all borders. Four colors applies clockwise starting from the top.

Specifying two or three colors also has defined meanings which we will not explain. (You can google if you are curious.) We tell you this as you may sometimes see other CSS experts showing off by applying this knowledge. Please don't become one of them. Specifying each border by name, when you want different borders on different sides, is a much better practice.

You might want to view our cautionary page re mixing different widths in a single border: `http://martinrinehart.com/frontend-engineering/artists/other/border-radius-limits.html`

Stick to a single border width, all around.

Position

In this chapter we will introduce the positioning of your boxes, as if it were a simple matter. Sometimes this is all you need to know. In Chapter 3 we go into positioning in more detail, as it is not, in reality, a simple matter.

Containers

First, everything is positioned relative to something else. Most commonly, boxes are positioned in containing boxes. At the outermost level, your HTML has a root element, `<html>`, that probably contains `<head>` and `<body>` elements. The `<body>` contains the visible elements. You may create `<div>`s that contain text and images. The text and images are positioned within their `<div>`s. The `<div>`s are positioned within the `<body>` (or within other elements, commonly other `<div>`s).

X, Y Positioning

The X and Y axes of the screen are not those you remember from math classes. X is horizontal, increasing from zero on the left. Y is vertical, increasing downward from zero at the top.

The main way to position elements is by setting their `left` (x) and `top` (y) properties. These are commonly set in pixels or percentages. Percentages are fractions of the container's size. It may be convenient to set `right` (x) and `bottom` (y) properties. Remember that results are not guaranteed if you provide styles for both (`left` and `right`, or both `top` and `bottom`).

The next examples illustrate the main ideas of x and y positioning. Begin by trying to answer this question: Where is `top:0px; left:0px` going to put one box, contained in another? (Assume they both have `borders` and `padding`.) We suggest you try Go Online 2b.

Online: Knowits > CSS II > Online > 2 > 2b

Next, suppose you wanted a contained box to live in the corner of a container box. Assume that you mean to place the contained box's border exactly over the container's border in one corner, as Figure 2-3 shows.

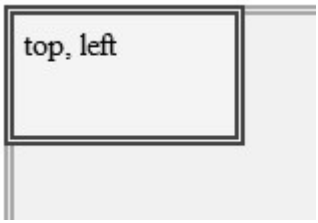


Figure 2-3

You want to understand the box model so completely that you can get this correct in every corner. Here are two hints. First, the position values (`left`, `top`, etc.) may be negative,. Second, `right: nnpx` means "inside the right of the container by nn pixels" (the mirror image of `left`). Go Online 2c makes this clear.

Online: Knowits > CSS II > Online > 2 > 2c

Z-Index Positioning

If you use just HTML it is the browser's job to ensure that items are laid out so they do not overlap. If you use CSS, it is easy to create overlapping boxes. When boxes overlap, which one should be on top (closer to the viewer, hiding items behind it)?

The simple rule is items are painted in the order they appear in the markup. If you don't want this order, you have two alternatives. One is to use the `z-index` property. The `z-index` is a number, an integer that

you assign. It defaults to zero. You may use negative numbers. Boxes are drawn on the screen in `z-index` order.

To be more exact, boxes in the same "stacking context" are drawn in `z-index` order or, if z indices are the same, in the order they appeared in the markup. Each container element establishes its own stacking context. An element with `z-index:10` may be hidden behind an element with `z-index:3` if they are not in the same stacking context.

If you are beginning to think this is not simple, we refer you to the actual CSS 2.1 specification (subsection 9.9.1, linked on the Companion Page) for the full rules. They are frightfully complex. We said there were two alternatives.

The second alternative is to simply rearrange elements in the markup. Elements that you want in front of others should be specified after all the elements they might overlap. If your markup is in back-to-front order, the `z-index` need never be specified. This almost always works. Use comments in the markup as necessary to explain otherwise strange orders.

See Chapter 3 for some of the many exceptions to common ordering on the z axis.

Size

In HTML, block-level elements (`<div>`, `<p>`, `<h?>`, etc.) are as wide as the container and as tall as the content requires. Inline elements are as wide as need be, wrapping to successive lines as required. CSS lets you get in the way.

If you specify height for a block-level element, the content may not fit. (An issue on the desktop, a major concern on tablets and phones.) If you specify width and height, inline elements may not fit. CSS lets you specify the layout, and gives you the power to cause problems.

Width and Height

You can specify the width and/or height with the `width` and `height` properties. See the discussion below, “Position and Size Values,” for the units you can use.

Minimums and Maximums

In addition to specifying a normal width with the `width` property, you can specify a `min-width` and/or a `max-width`. These will be critical for pages that may be viewed on anything from a phone to a desktop monitor. Go Online 2d shows you how to simulate different devices by dragging your browser's side in and out.

Online: Knowits > CSS II > Online > 2 > 2d

Position and Size Values

Percentages

If you set sizes in percentages (e. g., `width: 50%`) your sizes will be relative to your container element. The ultimate container for everything that you see is `<body>`, which will be, at least initially, sized based on the device size of the viewing hardware (phone, tablet, laptop, etc.).

Pixels

Pixels, a contraction of “picture elements” are the actual dots on the screen, until you start building with HTML and CSS. Paint a div, in your mind's eye, 100 pixels square. (There are 72 pixels per inch, just under 30 per cm, on many monitors.) The pixel density of many phones and tablets is higher; a true 100-pixel square's size will depend on the device pixel density.

But assume you are looking at a 100 pixel square on a 72 pixels-per-inch monitor. Now zoom in 20%. What happens? The physical pixels don't change. Your “logical” pixels are now each 1.2 physical pixels in size. Your browser is working hard to map logical pixels, specified in CSS, into physical pixels. Remember that you have lots of viewers and every one of them can zoom in or out, to suit their devices and eye sights. So how big is a pixel?

The simple answer is, you don't know. You can't know unless you know about the zoom level and the hardware.

We use pixel lengths all the time, in the simple expectation that our viewers have already set their hardware/browser combinations to be comfortable with the average pixels they view in cyberspace. That these may or may not be physical pixels doesn't worry us. 100 pixels on a desktop is longer than 100 pixels on a phone, but this is often just right as phones are usually viewed closer to the eye than desktop monitors.

A hundred pixels is specified `100px`.

Standard Length Measures

If we don't know how big a pixel is, we certainly don't know how long an inch or a centimeter is. We never use standard (non-cyberspace) lengths, but they are `in`, `cm` and `mm` if you want to try them. In theory these are the precisely specified lengths. In practice, your viewers' zoom settings change them. (You cannot zoom a physical ruler, thank goodness.)

Typographic Units

There are four old-fashioned typographic units available. The `em` and the `ex` were, in the printer's world, the width of an “M” and the height of an “x” in the font in use. The point, `pt`, and the pica, `pc`, are old English print units, 1/72nd of an inch per point, 12 points per pica. The type size in this book is 11 points. (11 and 11.5 are common sizes for books.)

Since a point is 1/72nd of an inch and a monitor may show 72 pixels per inch, a point is equal to a pixel, right? Well, that convenient identity

disappears with different device pixel densities, zoom levels and so on. We set font sizes in points and ems.

An em (in CSS, `em`) is a very useful size. It has been redefined to mean "relative to the current font size." You can use it with decimal values, so `1.2em` is 20% larger than the current base font size. If you set `font-size:12pt` (a good size for desktop use, generally too large for phones) `1.2em` is `14.4pt`. `14pt` is a useful `<h3>` heading size.

One convention suggests that you set the `font-size` of the `<body>` in `pt` and then set all other font sizes in `em`. That way changing the body font size changes all type sizes on your page. If your entire site uses a single CSS file, one change will do for every page in your site.

Another school (see the link to Bert Bos's paper on the Companion page) suggests that you don't set the body font size and then use ems to set all other sizes relative to the body font. The body font will be the browser default, as adjusted by the user (presumably) to match hardware capabilities and eyesight.

Yet another opinion holds that so many sites use 12 point type your viewer's browser is adjusted to suit that size. If you don't specify 12 points for the `<body>` you are defeating, not cooperating with, your viewers preferences.

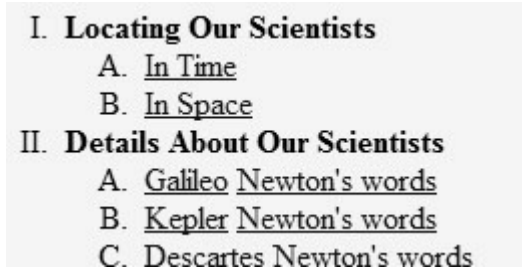
As time goes on we've become more partial to the "12 points for the body and everything else in ems" approach. Whichever you choose, it will be wrong for some of your viewers, so pick one and stick to it.

You now know enough about the box model and positioning boxes to do very useful work, as you will in the project assignment. In the next chapter, we'll go into some of the less obvious details.

Project

In the bottom-left corner of our HTML project we added an extra set of navigation links, as an outline. Part of it is shown in Figure 2-4.

Figure 2-4



Having extra navigation alternatives is always a good idea, and we really wanted to add an example of an outline built with nested ordered lists. (If you didn't create an HTML project with this feature, add one to your own project. Linking anywhere or nowhere is fine as long as you do it in nested, ordered lists, as Figure 2-4 shows.)

We achieved both of our HTML project goals, but in the process we won no prizes for our good looks.

This chapter's project assignment is to use our style sheet to make this outline a great-looking feature. Go Online 2e shows how we did this for our sample project.

Online: Knowits > CSS II > Online > 2 > 2e

And to achieve another teaching goal, please use descendant selectors, not classes.

Quiz

Choose the word or phrase that best completes each sentence.

1) CSS1 was
   a) an official W3C Recommendation in 1996.
   b) never widely supported.
   c) the foundation for all future CSS.
   d) all of the above.

2) The CSS box model areas, inside to outside, are
   a) content, padding, border and margin.
   b) content, margin, border and padding.
   c) content, border, margin and padding.
   d) content, padding, margin and border.

3) CSS box width is measured from
   a) the outside of the box borders.
   b) the inside of the box borders.
   c) the outside of the box content area.
   d) the outside of the box margins.

For questions four through seven, assume that the participating elements have content, padding, borders and margins.

4) In the rule `#box { top: 0px }` the part of the rule that specifies a pixel location or locations is
   a) the selector.
   b) the property name.
   c) the property value.
   d) the declaration.

5) In the rule `#box { top: 0px }` `top` refers to
   a) the top of the content area of `#box`.
   b) the top of the padding area of `#box`.
   c) the top of the margin area of `#box`.
   d) the top of the padding of the container of `#box`.

6) In the rule `#box { top: 0px }` `0px` refers to
   a) the top of the content area of `#box`'s container.
   b) the top of the padding area of `#box`'s container.
   c) the top of the margin area of `#box`'s container.
   d) the top of the border of `#box`.

7) If the container were the `<body>` element, the answer to question six
   a) would not change.
   b) would be different.

8) Specifying length in pixels
   a) works well because pixels are always the same size.
   b) does not work because pixels are not always the same size.
   c) works well, although pixels are not always the same size.

9) For a `<span>` element, the difference between `display:inline-block` and `display:block` is that
   a) the `inline-block` elements are aligned in a neat column.
   b) the `inline-block` elements are placed side-by-side across the page, space permitting.
   c) the `inline-block` elements are aligned, right justified.

10) For a `<span>` element, the difference between `display:static` and `display:inline` is
   a) none (there is no difference).
   b) `static` displays cannot be moved.
   c) `static` displays are full-width, like `block` displays.

3 Positioning

You started to use positioning when you wrote your first HTML. The `<h1>` heading was on a line by itself. The text and images that came after were “inline” elements, sharing space (if they fit on the line) below the headings. In this chapter we'll explain “normal flow” (the layout that HTML will do without further instructions from your CSS styles) and then show you how to take full control of your pages' layouts.

First, though, a quick look at where we've been.

History

CSS 1 became a W3C Recommendation in December, 1996. The W3C split its HTML group into three parts early in 1997. The part that we are

following is the CSS Working Group, which moved at warp speed (for public standards-making processes).

The first CSS 2 standards document was ready in November, 1997. It would become an official W3C Recommendation in May, 1998. As we noted before, it was a *de jure* standard, but it was not a *de facto* standard.

CSS 2 was, however, an important achievement and gave us much that we take for granted today. You've already seen absolute and relative positioning. (They are the subject of this chapter. You needed to use them already just to get your first CSS work completed.) CSS2 also introduced the still-complex subject of z-axis positioning.

CSS 2 provided the first support for media types, including print and aural media. We will refer to print (many users find printed copies helpful, even in today's mobile-saturated world). Aural and other assistive technologies are beyond the scope of this text, but be assured that the CSS you learn here is the CSS used by aural browsers, too. (You don't have `azimuth`—direction to the sound source—on a monitor. But it is just a CSS property used in CSS declarations in aural CSS style sheets.)

CSS 2.1, was the first standard to become a *de facto* standard, years after CSS2. Some CSS 2 ideas were dropped. (CSS 2 had font shadows but they were dropped in CSS 2.1. Do not mourn their loss; they return as a CSS 3 feature and you can use them today.) Most of CSS 2 lives on in today's CSS.

Now, on to positioning.

Normal Flow

As you know, HTML lays out pages for you. It is based on inline and block elements. Many authors just let HTML do the layout without thinking about it. Headings are placed above the text they head. Getting papers on the Web, especially academic papers, doesn't take much markup. From a CSS viewpoint, the layout HTML does is called “normal flow.”

Block Elements

In normal flow, the block elements (in HTML5, “block-level” elements) are the easy ones. A block element takes the full width of its container. It is placed below any prior content and above subsequent content. This works perfectly for headings in a text document, for example. (And it is usually just right for block quotes, for numbered and bullet lists and so on.)

Inline Elements

Inline elements are not so simple. First, it may not be obvious that every bit of your content is enclosed in tags separating the elements. You may have `<p>...</p>` elements, or you may have text with no visible enclosing tags. If you don't provide the element, CSS specifies an “anonymous” inline-level block element that is wrapped around the text. You cannot style anonymous elements (as there is no way to identify them: no tag, no class, no id).

If you mix elements with text, the individual bits of text are wrapped with anonymous elements. Consider:

```
Lorem <span>text in a span</span> ipsum
```

“Lorem “ (space as shown) is wrapped as an anonymous block. And so is “ ipsum”. In between, your `<span>` forms another, not anonymous, block.

Mostly, but not always, you can forget anonymous blocks and let your browsers handle them. For the “not always” part, try Go Online 3a:

Online: Knowits > CSS II > Online > 3 > 3a

It poses a question. Go Online 3b suggests an answer:

Online: Knowits > CSS II > Online > 3 > 3b

And Go Online 3c makes it conclusive:

Online: Knowits > CSS II > Online > 3 > 3c

(Tempted to skip? Where, in an otherwise empty div, will your browsers (plural) position the bottom of an `<img>`? It's on the quiz, and should be in your mental toolkit, too.)

Position Property Values

We cover most of the `position` property values here.

`fixed`

A “fixed” element does not scroll with the page. If you want a footer that is always at the bottom of the page, make it `position: fixed`. To be more precise, a fixed element is positioned within the viewport (the window through which the page is viewed) for non-paged media. Normal browser viewing is non-paged. For print media (definitely paged) the element's position is fixed within each output page.

Go Online 3d shows a `position: fixed` header and footer.

Online: Knowits > CSS II > Online > 3 > 3d

`absolute`

An “absolute” position is fixed within its container element. As the container moves, the `position: absolute` element moves with it. With percentage positioning, an absolutely positioned element may also move as its container is re sized. (Is the word “absolute” well-chosen?)

Fixed and absolutely positioned elements are lifted out of normal flow. Other elements are positioned as if they simply did not exist.

Go Online 3e shows a `position: absolute` header and footer.

Online: Knowits > CSS II > Online > 3 > 3e

`relative`

A "`relative`" position is a location relative to the element's position in normal flow. Positioning properties such as `left` are offsets from the position the browser would otherwise choose. This is often used to make fine adjustments. With `top:-2px` a relatively positioned element is raised a bit.

Go Online 3f shows a `position: relative` header and footer.

Online: Knowits > CSS II > Online > 3 > 3f

Go Online 3g shows a `position: fixed` header and footer, above and below a scrolling content area.

Online: Knowits > CSS II > Online > 3 > 3

Positioned Elements

Elements are considered "positioned" if they are `fixed`, `absolute` or `relative`. Elements are not "positioned" with `static` positions nor are they positioned if the `position` property is not specified. Important: only positioned elements respond to a specified size (`width` or `height`).

If you want to specify the size of an element, but let your browsers otherwise position it in normal flow, specify `position: relative` and do not specify `left`, `top`, `right` or `bottom`. Your `left` and `top` will default to zero; `right` and `bottom` will be undefined. Without `position: relative`, your `width` and/or `height` will be ignored and no error will be reported.

Important! When you are varying the `width` or `height` of an element and it refuses to respond that is a sure sign that the element is not positioned.

`static`

The `position: static` simply specifies that the element will be positioned in normal flow and is not "positioned" when it comes to having a specified size. We have no idea why this is called "static."

Float and Clear Properties

Floated elements may be placed on the left or right side of their containers. Other elements' inline content wraps around them. This is ideal for wrapping text around images. The `clear` property is used in elements that might be pushed out of the way by floated elements. It might more accurately be titled, `do-not-start-until-this-is-in-the-clear`.

`float`

Wrapping text around an image is very simple. Go Online 3h shows this common use.

Online: Knowits > CSS II > Online > 3 > 3h

That was the simple case: within a single container you were wrapping inline content around an inline element. Unfortunately, you may find your pages needing to float block-level elements, wrapping other inline- and block-level elements around them. This gets more complicated, and quickly.

When you float a block-level element, the element itself does not float. It still takes the full width of its container. The inline elements within the block-level element will wrap around the floated element. The shaded,

bordered `<p>` in Figure 3-1 is an example. Puppy's nose is in front of this `<p>`. The text within the `<p>` is floated to the right.

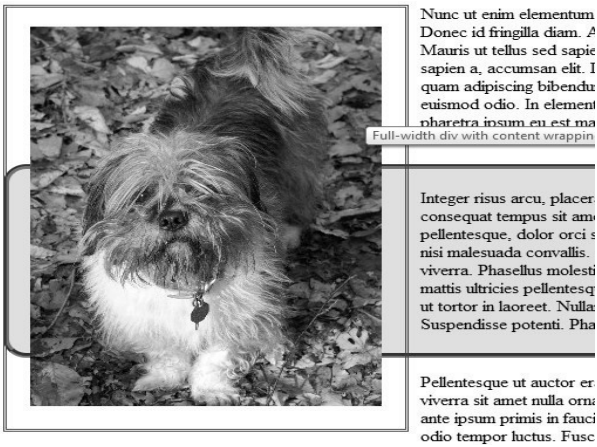


Figure 3-1

Go Online 3i should make this clear.

Online: Knowits > CSS II > Online > 3 > 3i

You can have multiple floated elements on either side. The first one in the markup floats on the side. The second one floats beside the first, and so on. On your own, add another photo to the page you made for Go Online 3i.

Warning: floating elements' height is not recognized by their containing elements. A container with no content other than floats will be zero pixels tall.

`clear`

The good news about the clear property is that it is very simple. Before CSS, it was an attribute of the HTML `<br>` tag, such as `<br clear='left'>`. Add a `clear: left` declaration to one of the paragraphs floating to the right in your last Go Online to see for yourself.

Display Properties

Setting the display property tells the browser's layout engine how you want an element shown. In scripting it is common to toggle a `div` between `none` and `block`, to hide and show the `div`. (`display: block` is the default for block-level elements, such as `div`s.)

`inline`

This is the default display for inline-level elements. Seldom specified in CSS, it would be used in scripting to turn an inline-level element on and off. Width, height and margins are not respected.

`block`

The default for block-level elements. Reminder: block-level elements default to the full width of their containers. Width and height are respected if the block is "positioned." Margins are respected.

`inline-block`

Changes a block-level element to display like an inline element. Width is reduced to fit the content. Alignment is between other inline elements (space permitting). Width, height and margins are respected.

PPK's Playground

The authoritative "Quirksmode" site, created by Peter-Paul Koch, a long-time frontend engineer and writer based in Amsterdam, includes what he calls a "Playground" where you can set display properties on nested elements. Go there and "play" with the settings until you are confident you understand their interactions.

`http://quirksmode.org/css/css2/display.html`

`list-item`

List items are special in that they have a number (ordered) or bullet (unordered) to the left of their content. You may never need to use this display value as it is the default for `<li>` elements.

`table`-xxx

Their are eight display properties in this group allowing you to replace HTML table-format attributes with CSS rules. Their use is straightforward. To lay out as a `<colgroup>` you would use `display: table-column-group`.

One you should bear in mind is `table-cell`. Vertical alignment works reliably within table cells, but not elsewhere. It is probably a very bad practice to format non-table elements as table cells. On the other hand, if your design requires vertical alignment, this may be the only choice. Sometimes you choose not what is best, but what is least bad.

`none`

Commonly used in scripting to turn elements on and off, but seldom used in plain CSS (except, possibly, to hide elements that your script will later turn on). There is one key difference between `display: none` and `visibility: hidden` that you should bear in mind.

Using `display: none` deletes your element from the layout, entirely. Using `visibility: hidden` makes your element invisible but places the other elements around it as if it were visible. (This generally leaves an objectionable “hole” in your page. Useful for very unique needs, only.)

Project

We took a good look at our timeline page to see if a little styling would be helpful. You should do the same. You may want to give it a shot

before you consider our opinions. These all apply to the timeline page, so an embedded style sheet in that page's `<head>` was our choice. Here are the improvements we made:

1. We added a border around the table. Pick your own styles.
2. We added some padding inside the table.
3. We created classes for each of our scientists.
4. We defined a `background-color` for each of our scientist classes (and removed the `bg-color=...` from the HTML).
5. We created a rule set combining all five of our scientist classes.
6. We used the rule set to add borders (with radii) to all our scientists' table entries.

Of these, removing the `bgcolor=...` from the HTML is probably most important. We now have the color for Galileo in exactly one place.

When we were happy, we took a new screenshot to replace the old one on the website's home page.

Quiz

Choose the word or phrase that best completes each sentence.

1) CSS2 became an official W3C Recommendation in
   a) 1996.
   b) 1997.
   c) 1998.
   d) 1999.

2) CSS2 was important as
   a) the first true CSS standard.
   b) the foundation for CSS 2.1.
   c) the foundation for CSS 2.1 and current CSS.

3) HTML text
   a) should always be enclosed in blocks.
   b) is enclosed in anonymous blocks.
   c) is enclosed in anonymous blocks if the markup does not enclose it in blocks.

4)Text blocks using different font sizes in normal flow are
   a) aligned on their baselines.
   b) are centered in their blocks.
   c) are an error.

5) Images, in normal flow, are
   a) centered in their lines.
   b) aligned with the top of adjacent text, if any.
   c) aligned with the baseline of adjacent text, if any.
   d) aligned with the default baseline of adjacent text, whether or not there is any adjacent text.

6) Elements that are not part of normal flow are
   a) `static`.
   b) `static` and `relative`.
   c) `fixed` and `absolute`.
   d) `fixed`, `absolute` and `display: none`.

7) A `float: left` element
   a) pushes following elements to its right.
   b) pushes following inline elements to its right.
   c) pushes following text to its right.

8) A `float: left` element following another `float: left` element
   a) is positioned on the container's left, following the earlier `float: left` element.
   b) is positioned to the right of the earlier `float: left` element, space permitting.
   c) is positioned based on the `clear` property of the earlier `float: left` element.

9) For a `<span>` element, the difference between `display:inline-block` and `display:block` is that
   a) the `inline-block` elements are aligned in a neat column.
   b) the `inline-block` elements are placed side-by-side across the page, space permitting.
   d) the `inline-block` elements are aligned, left justified.
   c) the `inline-block` elements are aligned, right justified.

10) To center text vertically
   a) use `vertical-align: center` in the text's element.
   b) use `vertical-align: middle` in the text's element.
   c) use `vertical-align: middle` in the text's containing element.
   d) use `vertical-align: middle` in a `table-cell` container.

END, Print Preview

A – Building a Carousel

Do you like challenges? Conquer this one and even veteran frontend engineers will be impressed. Here we'll use what you've learned to create a fully fluid image carousel. If you like, your images can be less than opaque so you can see the ones in back right through the ones in front. Take a peak at Go Online A1 to see what it looks like.

Online: Knowits > CSS II > Online > A1

A cube is an extruded square with a top and bottom added. We can also extrude all the other regular polygons: pentagons, hexagons and so on. (How many sides? How's your patience? As many as you like.)

There is a bit of tricky folding required to make fully fluid extruded polygons. The W3C specification omits percentage lengths in the

`translateZ()` transformation. You want your 3D objects to scale nicely in all three dimensions to fit as many devices as possible.

After you get your polygon shape, you'll add images to each side and set the object spinning on its Y axis (conveniently located in its center) The result is a treat!

Make your carousel a nice one. An animated carousel is a good example to show to potential employers or clients. The CSS is advanced and the commercial possibilities are endless. Go Online A1 shows you how.

Online: Knowits > CSS II > Online > A1

B – Basic Trigonometry

You probably studied the basics of sines, cosines and tangents when you were in high school. Most of us have had no occasion since to apply this knowledge. Your authors find that it crops up from time to time, but so infrequently that it requires looking up when it appears.

Suppose you want the user to select a time (to schedule an appointment) by clicking on a clock face. This makes a very quick, easy-to-use input widget but under the surface, some trigonometry is needed to translate the X, Y location of a mouse click into the time (angle of the clock hands).

And then there is the image carousel. We tried first with a spreadsheet replete with sines and cosines to compute X, Z coordinates (before we

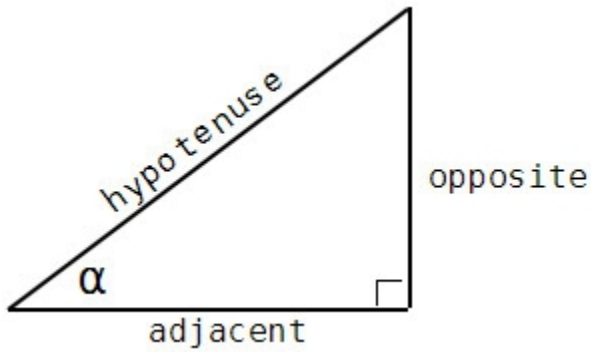
discovered that “folding” divs was easier and solved the essential problem of keeping the result fluid).

This is the quick review that we would need before grabbing our calculators.

Fundamental Ratios

Based on a right triangle, the three most common functions in trigonometry are the ratios of pairs of sides. The hypotenuse of the right triangle, the long side, is the denominator for sines and cosines. The sides at 90 degrees to each other are called “adjacent” (one side of the angle) and “opposite” (away from the angle). Figure B-1 shows these.

Figure B-1

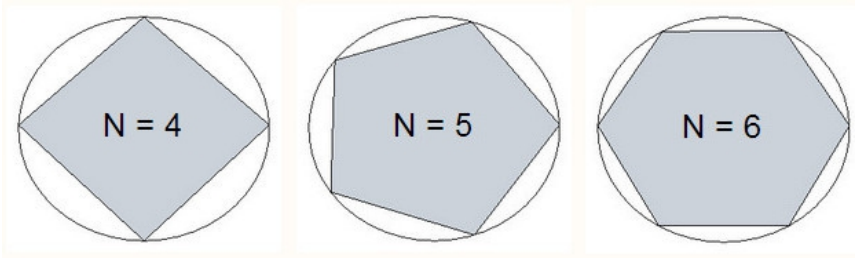



```
sine = opposite / hypotenuse
cosine = adjacent / hypotenuse
tangent = opposite / adjacent
```

Unit Circles

It is often true that embedding your right triangle in a “unit circle” (a circle that has a radius of one) saves you from a great deal of arithmetic. Figure B-2 shows regular polygons embedded in unit circles.

Figure B-2



The sines and cosines of your angle are also the lengths of the adjacent and opposite sides, as shown in Figure B-3.

Figure B-3

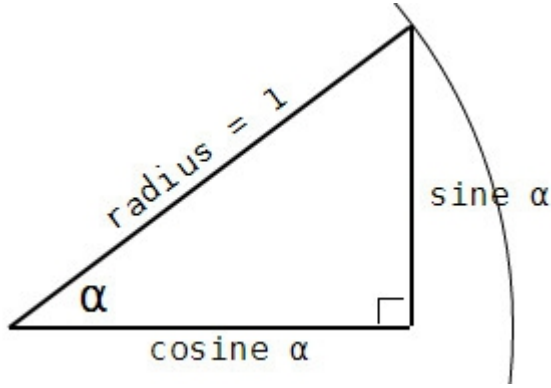
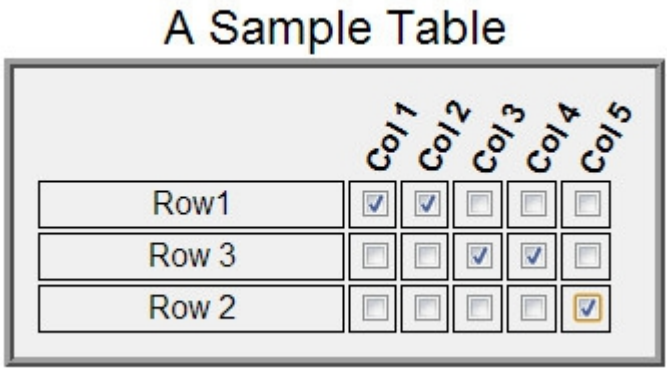


Table Headings

In laying out tables with HTML (any version of HTML prior to CSS3 transformations) we were limited to writing horizontal text above the columns. If the data in the columns did not need many pixels, we still had to have a column that was wide enough to hold the heading.

A simple solution to fitting more columns (or taking less space) is to rotate the column headings, as shown in Figure B-4.

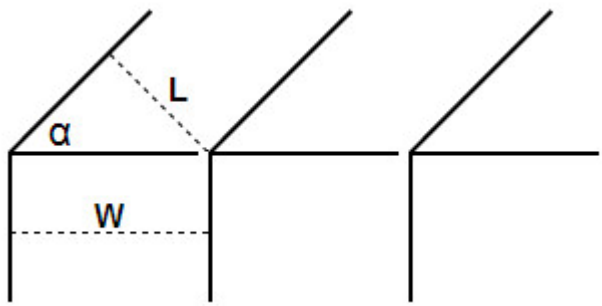
Figure B-4



The headings in Figure B-4 are rotated 60 degrees. More than this becomes very difficult to read. (Vertical text is extremely difficult to read. Test this the first time you rotate table headings.) How wide a column is needed? That depends on the height of the heading line and the angle of rotation. For this, we need some trigonometry.

If “L” is the line-height of the heading text, and “W” is the width of the column needed for the line-height, we can calculate the relationship between L and W as shown in Figure B-5.

Figure B-5



In Figure B-5, the hypotenuse of the triangle enclosing angle *alpha* is W long. The sine of *alpha,* the opposite divided by the hypotenuse, is L / W so we get:

```
sine α = L / W

L = W * sine α
```

or

```
W = L / sine α
```

You choose the more restrictive of the label line height or the column width and then use one of the above equations to find the unknown.

C – Regular Polygons

We are ready to apply our small subset of trigonometry to the problem of "drawing" regular polygons. For our carousel, we showed how you could apply CSS3 transforms to create a fluid 3D carousel with absolutely no math.

Online: Knowits > CSS II > Online > A1

However, as you go on to other forms, you will not always stumble on solutions that depend solely on "folding" your "paper" to achieve the result you seek.

We start by looking at regular polygons drawn in a unit circle. This leads quickly to a generalized method of deriving the X, Y coordinates of the

vertexes (angles) in the polygons, and the ratio of the side length to the circle's radii.

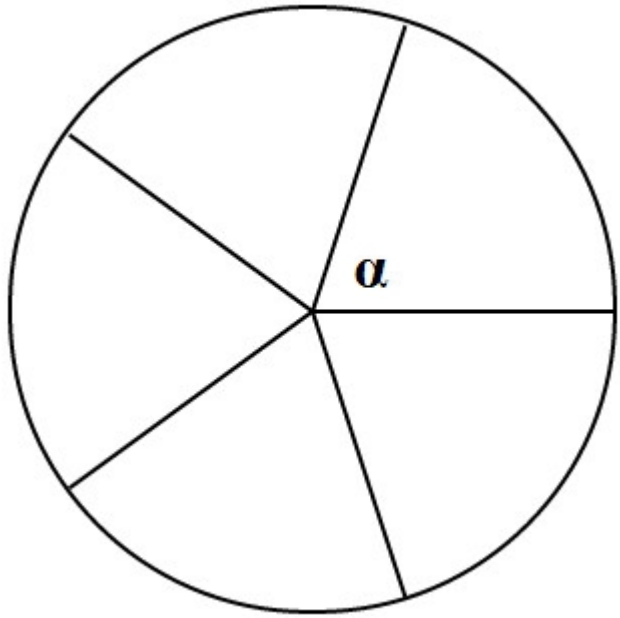
Polygon in Unit Circle

By now you have become accustomed to the CSS coordinate system: 0, 0 is at the top, left corner and values increase downward and to the right. We ask you now to return to the Cartesian coordinates you used in high school math: 0, 0 is in the center; X increases to the right and Y increases going up. (The CSS, and HTML, system will be used for most of your work, but if you advance to WebGL, making heavy use of modern 3D, you will put the Cartesian coordinates back into service. 0, 0, 0 will be the center of your cyber worlds.)

To draw a regular polygon, start with a circle. Lay out a radius line. Place this base radius at 3:00 o'clock to make many future steps easier. (That's the line from 0, 0 to 1, 0 if you have a unit circle in the center.)

Then you draw additional radii around the circle at equal angles (the angle is named *alpha* in the figures). The angle, for an `N`-sided polygon is `360 / N`. Figure C-1 shows a pentagon, `N` = 5, radii 72 degrees apart.

Figure C-1



With circle and spokes you complete the polygon by connecting the intersections of the spokes and the circle. Figure C-2 shows the result and names the angle between the radii and the side *beta*.

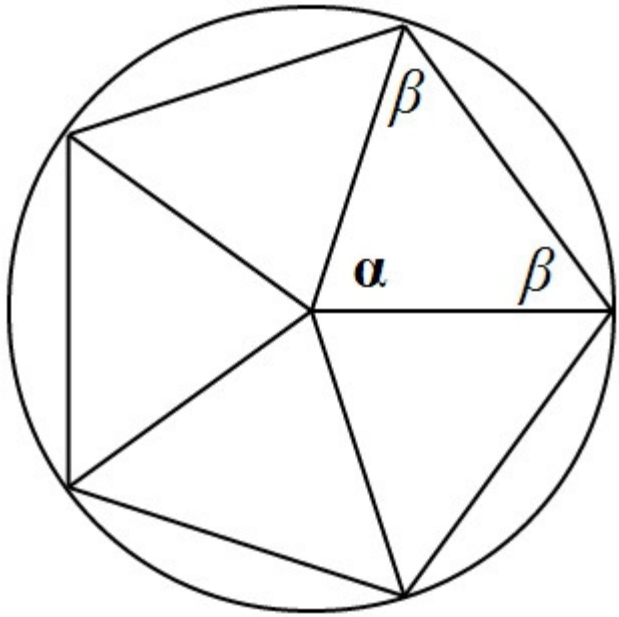
Note that there are 180 degrees in the sum of the degrees in each angle in a triangle. In our drawing:

```
α + β + β = 180
```

Alternatively,

```
β = (180 - α) / 2
```

Figure C-2



Remember that our unit circle radii are all one unit long. If we want a radius of 150 pixels, we simply declare a unit to be 150 pixels. In this example, we got a circle by starting with `height` and `width` of 300 pixels and then `border-radius` of 150 pixels.

Continuing with CSS, our radii are `div`s with a two-pixel border, top only. They are 150 pixels wide, 0 high. We created them all where the 3:00 o'clock radius remains. The second one was rotated `-72` degrees. The third one was rotated `-144` degrees (2 * -72) and so on.

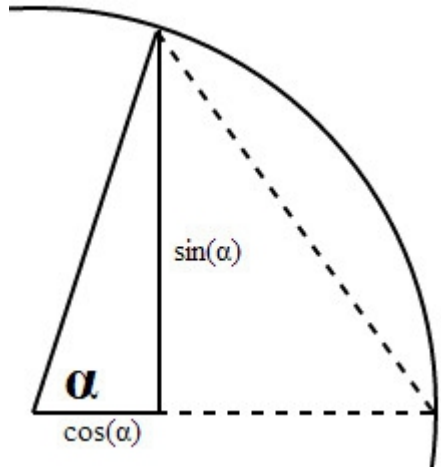
For sides, we attached another div as the child of each radius and rotated it as Go Online A-1 showed.

Vertex Coordinates

Now we are ready to get the coordinates we need to draw a regular polygon using CSS. (“Vertex” is the name for a point in the mathematics of computer graphics.) We begin by drawing a line from the second

corner (the intersection of the second radius with the circle) straight down to the first radius, as Figure C-3 shows.

Figure C-3



From there, the length of the inscribed triangle's side adjacent to *alpha* is the cosine of *alpha*. That is the value of X, in units. (Multiply by 150 for units 150 pixels long.) Similarly, the value of Y is the length of the opposite side of the inscribed triangle, which is the sine of *alpha*.

Now, a little spreadsheet (Figure C-4) lets us create all the answers we need. (Our spreadsheet works for a dozen-sided polygon. For a pentagon, we get the same coordinates for radius five that we got for radius zero, which is harmless.) Just plug in the number of sides, the length of a unit and you're done.

Figure C-4

	A	B	C	D	E
1					
2	Number of sides:		8		
3	Length of radius:		200		
4					
5		Alpha	45		
6		**Angle**	**Rads**	**X**	**Y**
7	0	0	0.000	200.0	0.0
8	1	45	0.785	141.4	141.4
9	2	90	1.571	0.0	200.0
10	3	135	2.356	-141.4	141.4
11	4	180	3.142	-200.0	0.0
12	5	225	3.927	-141.4	-141.4
13	6	270	4.712	0.0	-200.0
14	7	315	5.498	141.4	-141.4
15	8	360	6.283	200.0	0.0
16	9	405	7.069	141.4	141.4
17	10	450	7.854	0.0	200.0
18	11	495	8.639	-141.4	141.4
19	12	540	9.425	-200.0	0.0
20					

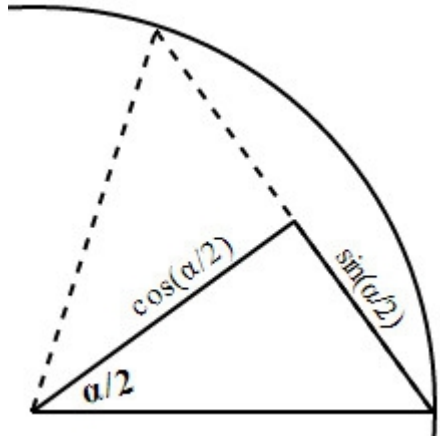
We've shown the results for an 8-sided polygon, 200 pixel radius. This is an easy one to test because you can check it without a calculator. At zero degrees (3:00 o'clock, radius zero) the vertex is (200, 0). At 45 degrees, the vertex is (141.4, 141.4). (You recognized the square root of two, we hope.)

Now there is only one problem left to solve: how long is a side of the polygon for a given radius?

Length of a Side

Figure C-5 shows another radius, bisecting angle *alpha*. Like the last inscribed triangle, the hypotenuse of this inscribed triangle is again one radius long. So again we can read the lengths directly from the diagram.

Figure C-5



As the length of a half side is the sine of half *alpha,*

```
side length = 2 * sin(α/2)
```

(That is the length in units. Multiply by the length of the unit for the length in pixels.)

For a pentagon:

N = 5

α = 360 / N = 72

sine (α / 2) = sine (36) ~= 0.5878

side length = 0.5878 * 2 = 1.1756

Quiz Answers

I	1c,	2a,	3d,	4b,	5c,	6a,	7c,	8c,	9d,	10c
II	1c,	2a,	3c,	4d,	5c,	6b,	7b,	8c,	9b,	10a
III	1c,	2c,	3c,	4a,	5d,	6d,	7b,	8b,	9b,	10d
IV	1b,	2d,	3c,	4d,	5b,	6c,	7d,	8b,	9a,	10d
V	1c,	2c,	3c,	4b,	5d,	6c,	7d,	8a,	9b,	10c
VI	1a,	2d,	3c,	4b,	5a,	6a,	7a,	8d,	9d,	10d
VII	1b,	2b,	3a,	4c,	5a,	6b,	7a,	8d,	9c,	10b
VIII	1a,	2b,	3b,	4a,	5b,	6d,	7a,	8c,	9c,	10a
IX	1d,	2a,	3a,	4d,	5d,	6d,	7c,	8a,	9d,	10b
X	1d,	2c,	3d,	4c,	5a,	6a,	7d,	8b,	9b,	10a