CSS3 Learning Notes

Monday, May 05, 2014 12:51

1. **Anatomy of a rule (or rule set)**

```
selector {
        property: value;
}
```

   a. A property
   b. A value
   c. A Declaration: A property: value pair is called a declaration.
   d. Declaration block
   e. Keywords: eg. different color names (red, green, blue, etc) and different border styles (dashed, dotted, solid)
   f. CSS units: eg. 1px
   g. Functional notation: functional notation is used to denote colors, attributes, and URIs in CSS3.

```
First, using hexadecimal:
blockquote {
        background: #FF0000;
}
Second, using a keyword:
blockquote {
        background: red;
}
Third, using functional notation:
blockquote {
        background: rgb(255,0,0);
}
```

   h. Selectors
   i. Combinators
   j. At-rules: eg. @import, @page, @media

2. **CSS shorthand**

   One way to bloat your style sheet (although sometimes it can be useful) is to use multiple declarations when one will do. There are a number of properties that can be turned into CSS shorthand, saving both time and file size. We'll look at margin and padding but similar rules apply to the various background, border, font, list-style, and outline properties.

3. **CSS Reset**

   a. Reset Reasoning

      The basic reason is that all browsers have presentation defaults, but no browsers have the same defaults. (Okay, no two browser families—most Gecko-based browsers do have the same defaults.) For example, some browsers indent unordered and ordered lists with left margins, whereas others use left padding. In past years, we tackled these inconsistencies on a case-by-case basis; for example, making sure to always set both left padding and left margin on lists.

      But there are all kinds of inconsistencies, some more subtle than others. Headings have slightly different top and bottom margins, indentation distances are different, and so on. Even something as basic as the default line height varies from one browser to another—which can have profound effects on element heights, vertical alignments, and overall feel.

   b. Using normalize.css

      i. Different between normalize and reset

         1) CSS resets aim to remove all built-in browser styling. Standard elements like H1-6, p, strong, em, et cetera end up looking exactly alike, having no decoration at all. You're then supposed to add all decoration yourself.
         2) Normalize CSS aims to make built-in browser styling consistent across browsers. Elements like H1-6 will appear bold, larger et cetera in a consistent way across browsers. You're then supposed to add only the difference in decoration your design needs.
         3) Normalize.css preserves useful defaults rather than "unstyling" everything. For example, elements like sup or sub "just work" after including normalize.css (and are actually made more robust) whereas they are visually indistinguishable from normal text after including reset.css. So, normal ize.css does not impose a visual starting point (homogeny) upon you. This may not be to everyone's taste. The best thing to do is experiment with both and see which ge ls with your preferences.
         4) Normalize.css corrects some common bugs that are out of scope for reset.css. It has a wider scope than reset.css, and also prov ides bug fixes for common problems like: display settings for HTML5 elements, the lack of font inheritance by form elements, correcting font -size rendering for pre, SVG overflow in IE9, and the button styling bug in iOS.
         5) Normalize.css doesn't clutter your dev tools. A common irritation when using reset.css is the large inheritance chain that is displayed in browser CSS debugging tools. This is not such an issue with normalize.css because of the targeted stylings.
         6) Normalize.css is more modular. The project is broken down into relatively independent sections, making it easy for you to pot entially remove sections (like the form normalizations) if you know they will never be needed by your website.
         7) Normalize.css has better documentation. The normalize.css code is documented inline as well as more comprehensively in the Gi tHub Wiki. This means you can find out what each line of code is doing, why it was included, what the differences are between browsers, and more easily run your own tests. The project aims to help educate people on how browsers render elements by default, and make it easier for them to be involved in submitting improveme nts.

4. **Eight rules of thumb to remember when crafting your stylesheets**

   i. Think in terms of components, not pages.
   ii. Think about types of things, not individual things.
   iii. Prefer classes to IDs.
   iv. Create composable classes.
   v. Use descendent selectors to avoid redundant classes.
   vi. Keep your selectors as short as possible.
   vii. Order your CSS rules loosely by specificity.
   viii. Prefer percentages for internal layout dimensions.

5. **Vendor prefixes**

Prefix	Browser/Company
-khtml-	Konqueror browser
-ms-	Microsoft
-moz-	Mozilla (Gecko browsers)
-moz-	Opera
-webkit-	Safari and Chrome (plus other WebKit browsers)

   i. **Prefix-free:** If you feel daunted by the thought of writing out declarations multiple times and can't promise to keep to your pledge, there are tools out there to help. The best of which is –prefix-free by Lea Verou, which is a small JavaScript file you can include in your site.

6. **CSS Selectors**

i. **The following selectors were introduced in CSS1:**
   1) **Type (e.g., p {...}, blockquote {...})**
   2) **Descendant combinator (e.g., blockquote p {...})**
   3) **ID (e.g., #content {...} on <article id="content">)**
   4) **Class (e.g., .hentry {...} on < article class="hentry">)**
   5) **Link pseudo-class (e.g., a:link {...} or a:visited {...})**
   6) **User action pseudo-class (e.g., a:active {...})**
   7) **:first-line pseudo-element (e.g., p:first-line {...})**
   8) **:first-letter pseudo-element (e.g., p:first-letter {...})**

ii. Another 11 selectors were added in CSS 2.1.
   1) Universal (e.g., * {...})
   2) User action pseudo-class (e.g., a:hover {...} and a:focus {...})
   3) The :lang() pseudo-class (e.g., article:lang(fr) {...})
   4) Structural pseudo-class (e.g., p:first-child {...})
   5) The :before and :after pseudo-elements (e.g., blockquote:before {...} or a:after{...})
   6) Child combinator (e.g., h2 > p {...})
   7) Adjacent sibling combinator (e.g., h2 + p {...})
   8) Attribute selectors (e.g., input [required] {...})
   9) Attribute selectors; exactly equal (e.g., input [type="checkbox"] {...})
   10) Substring attribute selectors; one equal to string (e.g., input [class~="long-field"] {...})
   11) Substring attribute selectors; hyphen separated beginning with string (e.g., input [lang|="en"] {...})

iii. The selectors added in CSS3 that we'll be looking at in this chapter are as follows:
   1) General sibling combinator (e.g., h1 ~ pre {...})
   2) Substring attribute selectors; string starts with (e.g., a[href^="http://"] {...})
   3) Substring attribute selectors; string ends with (e.g., a[href$=".pdf"] {...})
   4) Substring attribute selectors; string contains (e.g., a[href*="twitter"] {...})
   5) The :target() pseudo-class (e.g., section:target {...})
   6) Structural pseudo-classes; :nth-child (e.g., tr:nth-child(even) td {...})
   7) Structural pseudo-classes; :nth—last-child (e.g., tr:nth—last-child(-n+5) td {...})
   8) Structural pseudo-classes; :last-child (e.g., ul li:last-child {...})
   9) Structural pseudo-classes; :only-child (e.g., ul li:only-child {...})
   10) Structural pseudo-classes; :first-of-type (e.g., p:first-of-type {...})
   11) Structural pseudo-classes; :last-of-type (e.g., p:last-of-type {...})
   12) Structural pseudo-classes; :nth-of-type (e.g., li:nth-of-type(3n) {...})
   13) Structural pseudo-classes; :nth-last-of-type (e.g., li:nth-last-of-type(1) {...})
   14) Structural pseudo-classes; :only-of-type (e.g., article img:only-of-type {...})
   15) Structural pseudo-classes; :empty (e.g., aside:empty {...})
   16) Structural pseudo-classes; :root (e.g., :root {...})
   17) UI element states pseudo-classes; :disabled and :enabled (e.g., input:disabled {...})
   18) UI element states pseudo-classes; :checked (e.g., input[type="checkbox"]:checked {...})
   19) Negation pseudo-classes; :not (e.g., abbr:not([title]} {...})

7. **CSS Layout (Details see the Chapter 9 A Layout for Every Occasion of "Beginning HTML5 and CSS3")**
   i. **The Box Model: content, padding, border, margin**

ii. **The width and height calculation algorithm**
   1) Total width = margin-left + border-left-width + padding-left + width + padding-right + border-right-width + margin-right
   2) Total height = margin-top + border-top-width + padding-top + height + padding-bottom + border-bottom-width + margin-bottom

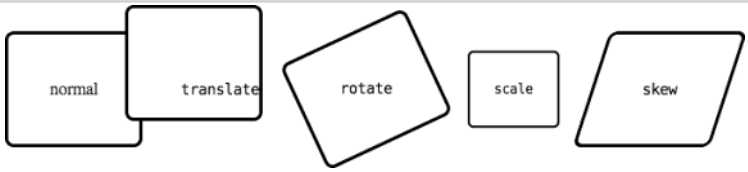
8. The Benefits of CSS3

a. Streamlining Design
   more and more designers are beginning to design "in the browser.
b. Reduced workarounds and hacks
   i. Specifying rich web fonts: No more creating type in an image editor and serving it via image replacement or using a tool like sIFR.
   ii. Creating round corners automatically: No more reliance on images for creating boxes with rounded corners. It can all be handled automatically with border-radius.
   iii. Opacity and alpha channels: Create transparency without images using opacity or rgba.
   iv. Creating gradients on the fly: No more creating repeating images for gradients or patterns. Simply use CSS gradients (radial or linear).
   v. Reducing heavy scripting for simple animations: No need to use JavaScript to create simple transitions or transforms. It can now be handled in CSS.
   vi. Cutting additional markup: No need to write additional markup to create columns.
   vii. Less classitis: No need to add additional extraneous classes when elements can be targeted using new CSS3 selectors.

9. Transforms, Transitions, and Animation

a. Simple example

```
.translate {transform: translate(-24px, -24px);}
.rotate {transform: rotate(-205deg);}
.scale {transform: scale(.75);}
.skew {transform: skewX(-18deg);}
```



b. Transform properties and functions
   i. Transform: This property takes one or more space-separated transform functions (listed below) to apply to an element, for example transform: translate(3em, -24px) scale(.8);. Transform functions can take negative values, and the default is none. The transform functions include the following:
      - translate(): Moves the element from its transform-origin along the X, Y,and/or Z-axes. This can be written as translate(tX), translate(tX, tY), and translate3D(tX, tY, tZ) (percentage except tZ, lengths). There's also the 2D translateX(tX) and translateY(tY), and the 3D translateZ(tZ).
      - rotate(): Rotates the element around its transform-origin in two-dimensional space, with 0 being the top of the element, and positive rotation being clockwise (angles). There are also the rotateX(rX), rotateY(rY), and rotateZ(rZ)3 transformation properties to rotate around an individual axis. Finally, there's rotate3D(vX, vY, vZ, angle) to rotate an element in three-dimensional space around the direction vector of vX, vY, and vZ (unitless numbers) by angle (angles).
      - scale(): Changes the size of the element, with scale(1) being the default. It can be written as scale(s), scale(sX, sY), and scale3D(sX, sY, sZ) (unitless numbers). There's also the 2D transforms scaleX(sX) and scaleY(sY), and the 3D transform scaleZ(sZ).
      - skew(): Skews the element along the X (and, if two numbers are specified, Y) axis. It can be written as skew(tX) and skew(tX, tY) (angles). There's also skewX() and skewY().
      - matrix(): This transform property takes a transformation matrix that you know all about if you have some algebra chops. matrix() takes the form of matrix(a, b, c, d, e, f) (unitless numbers). matrix3D() takes a 4×4 transformation matrix in column-major order. The 2D transform matrix() maps to matrix3D(a, b, 0, 0, c, d, 0, 0, 0, 0, 1, 0, e, f, 0, 1) (unitless numbers). If you have the required giant brain, this lets you do (pretty much) all other 2D and 3D transforms at once.
      - perspective(): Provides perspective to 3D transforms and controls the amount of foreshadowing (lengths)—think fish-eye lenses in photography. The value must be greater than zero, with about 2000px appearing normal, 1000px being moderately distorted, and 500px being heavily distorted. The difference with the perspective property is that the transform function affects the element itself, whereas the perspective property affects the element's children. Note that perspective() only affects transform functions after it in the transform rule.
   ii. perspective: This works the same as the perspective transform function, giving 3D transformed elements a feeling of depth. It affects the element's children, keeping them in the same 3D space.
   iii. perspective-origin: This sets the origin for perspective like transform-origin does for transform. It takes the same values and keywords as transform-origin: keywords, lengths, and percentages. By default this is perspective-origin: 50% 50%;. It affects the children of the element it's applied to, and the default is none.
   iv. transform-origin: Sets the point on the X, Y, and/or Z-axes around which the transform(s) will be performed. This can be written transform-origin: X;, transform-origin: X Y;, and transform-origin: X Y Z;. We can use the keywords left, center, and right for the X-axis, and top, center, and bottom for the Y-axis. We can also use lengths and percentages for X and Y, but only lengths for Z. Finally, for a 2D transform-origin you can use offsets by listing three or four values, which take the form of two pairs of a keyword followed by a percentage or length. For three values a missing percentage or length is treated as 0. By default transform-origin is the center of the element, which is transform-origin: 50% 50%; for a 2D transform and transform-origin: 50% 50% 0; for a 3D transform.
   v. transform-style: For 3D transforms this can be flat (the default) or preserve-3d. flat keeps all children of the transformed element in 2D—in the same plane. preserve-3d child elements transform in 3D, with the distance in front of or behind the parent element controlled by the Zaxis.
   vi. backface-visibility: For 3D transforms this controls whether the back side of an element is visible (the default) or hidden.

c. CSS transitions and CSS animations: compare and contrast
   i. CSS transitions can be triggered by a CSS change in state and JavaScript. CSS animations play by default once declared, although you can also trigger them by a CSS change in state9 and JavaScript.
   ii. CSS transitions apply a transition to an existing instant change. CSS animations add styles to an element and animate using them.
   iii. CSS transitions occur between two intrinsic styles,10 the element's intrinsic style before and after the transition is triggered (such as non-:hover and :hover values). CSS animations animate from the element's intrinsic state and between (multiple) keyframes. By default, the element will return to its intrinsic state when the animation ends.
   iv. CSS transitions are simple, with wider browser support. CSS animations are more powerful and complex, with less browser support.