CSS: Cascading Style Sheets

The Complete Web Developer in 2018

The Complete Web Developer in 2018
Zero to Mastery
Andrei Neagoie
Lecture Notes by Stephanie

CSS Cheat Sheet

Reference:
*https://www.w3schools.com/cssref/css_selectors.asp
*https://css-tricks.com/almanac/

Cascading Style Sheets
>>> the order of CSS rules matter.

.class
#id (similar to class but can only be used once)
*   (all elements, not used often, usually at top of css sheet)
element
element1, element2 (element1 and element2)
element1 element2   (all elem2 inside elem1)
element1 > element2 (all elem2 that have parent elem1)
element1 + element2 (any elem2 exactly after elem1)
:hover   (style on mouse hover)
:last-child
:first-child
!important (not recommended)

What selectors win out in the cascade depends on:
-Specificity
-Importance
-Source Order

CSS - Cascading Style Sheet

- CSS file "cascades"- it uses the last applicable input for a given selector
- there are three ways to css styles: separate css file, in-line styling, and <style>

- To link html file to css style sheet, use: <link rel="stylesheet" type="text/css" href="">
rel is relation, specifies what relationship
type is media type
href is link to the css file

- can also use in-line styles within HTML file, for example: <header style="background-color: green; color:red">

- within HTML file, within the <head>, can create style tags: <style>

- see common css code in file named *27.4 style.css*

- use RGBA instead of hex to incorporate opacity

sublime shortcuts - css

- after creating css file, to link to it in the html file, go to head section of html file and use link tag: type "<link" and press tab to autofill the rest

https://cssguidelin.es/

High-level advice and guidelines for writing sane, manageable, scalable CSS (V. Practical, see full guide online)

- two (2) space indents, no tabs;
- 80 character wide columns;
- Table of Contents

```
/*-------------------------------------*\
  #SECTION-TITLE
\*-------------------------------------*/
```

section title with hashtag for searches
each title should be preceded by f*ive (5) carriage return*

```
/*-------------------------------------*\
  #A-SECTION
\*-------------------------------------*/

.selector { }
```

```
/*-------------------------------------*\
  #ANOTHER-SECTION
\*-------------------------------------*/

/**
 * Comment
 */

.foo, .foo--bar,
.baz {
  display: block;
  background-color: green;
  color: red;
}
```

each declaration indented by two (2) spaces; NO TABS

All strings in classes are delimited with a hyphen (-), like so:

```
.page-head { }
.sub-content { }
```

BEM splits components' classes into three groups:
Block: The sole root of the component.
Element: A component part of the Block.
Modifier: A variant or extension of the Block.

To take an analogy (note, not an example):

.person { }
.person__head { }
.person--tall { }
Elements are delimited with two (2) underscores (__), and Modifiers are delimited by two (2) hyphens (--).

Here we can see that .person {} is the Block; it is the sole root of a discrete entity. .person__head {} is an Element; it is a smaller part of the .person {} Block. Finally, .person--tall {} is a Modifier; it is a specific variant of the .person {} Block.

Selector Intent
An unambiguous, explicit selector with good Selector Intent. We are explicitly selecting the right thing for exactly the right reason.

Poor Selector Intent is one of the biggest reasons for headaches on CSS projects. Writing rules that are far too greedy—and that apply very specific treatments via very far reaching selectors—causes unexpected side effects and leads to very tangled stylesheets, with selectors overstepping their intentions and impacting and interfering with otherwise unrelated rulesets.

CSS cannot be encapsulated, it is inherently leaky, but we can mitigate some of these effects by not writing such globally-operating **selectors: your selectors should be as explicit and well reasoned as your reason for wanting to select something.**

Reusability
We want the option to be able to move, recycle, duplicate, and syndicate components across our projects.

Location Independence
it is in our interests not to style things based on where they are, but on what they are. A component shouldn't have to live in a certain place to look a certain way.

Portability

Quasi-Qualified Selectors
Use comments to make selector read as specific without actually being specific.

Naming
By using slightly more ambiguous names, we can increase our ability to reuse these components in different circumstances.

```
 * Runs the risk of becoming out of date; not very maintainable.
.blue {
  color: blue;
}

* Depends on location in order to be rendered properly.
.header span {
  color: blue;
}

* Too specific; limits our ability to reuse.
.header-color {
  color: blue;
}

/**
 * Nicely abstracted, very portable, doesn't risk becoming out of date.
 */
.highlight-color {
  color: blue;
}
```

Selector performance
the longer a selector is (i.e. the more component parts) the slower it is

Select what you want explicitly, rather than relying on circumstance or coincidence. Good Selector Intent will rein in the reach and leak of your styles.
Write selectors for reusability, so that you can work more efficiently and reduce waste and repetition.
Do not nest selectors unnecessarily, because this will increase specificity and affect where else you can use your styles.
Do not qualify selectors unnecessarily, as this will impact the number of different elements you can apply styles to.
Keep selectors as short as possible, in order to keep specificity down and performance up.
if a selector will work without it being nested then do not nest it.

Keep specificity low at all times

Rules are the children of principles.

Object-orientation is a programming paradigm that breaks larger programs up into smaller, in(ter)dependent objects that all have their own roles and responsibilities

Modularity
Object Oriented CSS (OOCSS)
OOCSS deals with the separation of UIs into structure and skin: breaking UI components into their underlying structural forms, and layering their cosmetic forms on separately
- Whenever you are building a UI component, try and see if you can break it into two parts: one for structural styles (paddings, layout, etc.) and another for skin (colours, typefaces, etc.).

[s]oftware entities (classes, modules, functions, etc.) should be **open for extension, but closed for modification.**

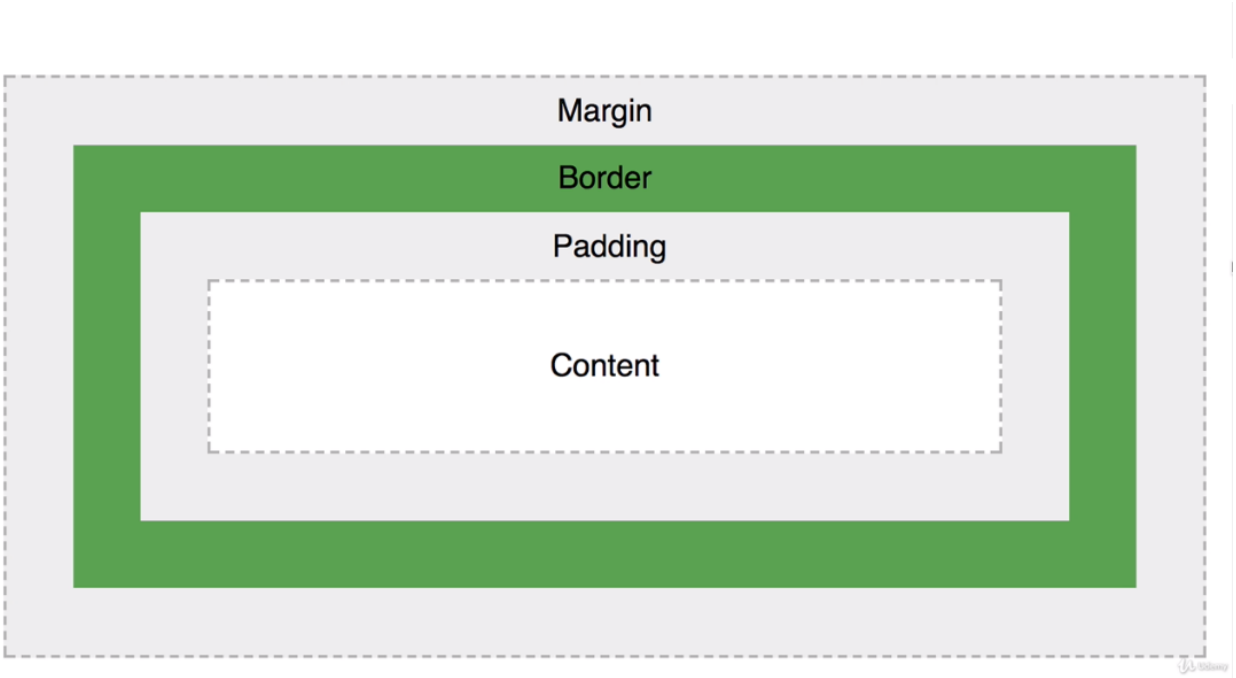
Composition over Inheritance
Now that we're used to spotting abstractions and creating single responsibilities, we should be in a great position to start composing more complex composites from a series of much smaller component parts. Nicole Sullivan likens this to using Lego; tiny, single responsibility pieces which can be combined in a number of different quantities and permutations to create a multitude of very different looking results.

This idea of building through composition is not a new one, and is often referred to as composition over inheritance. This principle suggests that larger systems should be composed from much smaller, individual parts, rather than inheriting behaviour from a much larger, monolithic object. This should keep your code decoupled—nothing inherently relies on anything else.

Composition is a very valuable principle for an architecture to make use of, particularly considering the move toward component-based UIs. It will mean you can more easily recycle and reuse functionality, as well rapidly construct larger parts of UI from a known set of composable objects. Think back to our error message example in the Single Responsibility Principle section; we created a complete UI component by composing a number of much smaller and unrelated objects.

The Separation of Concerns
code should be broken up into distinct sections, such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program. […] A program that embodies SoC well is called a modular program.

CSS Box Model

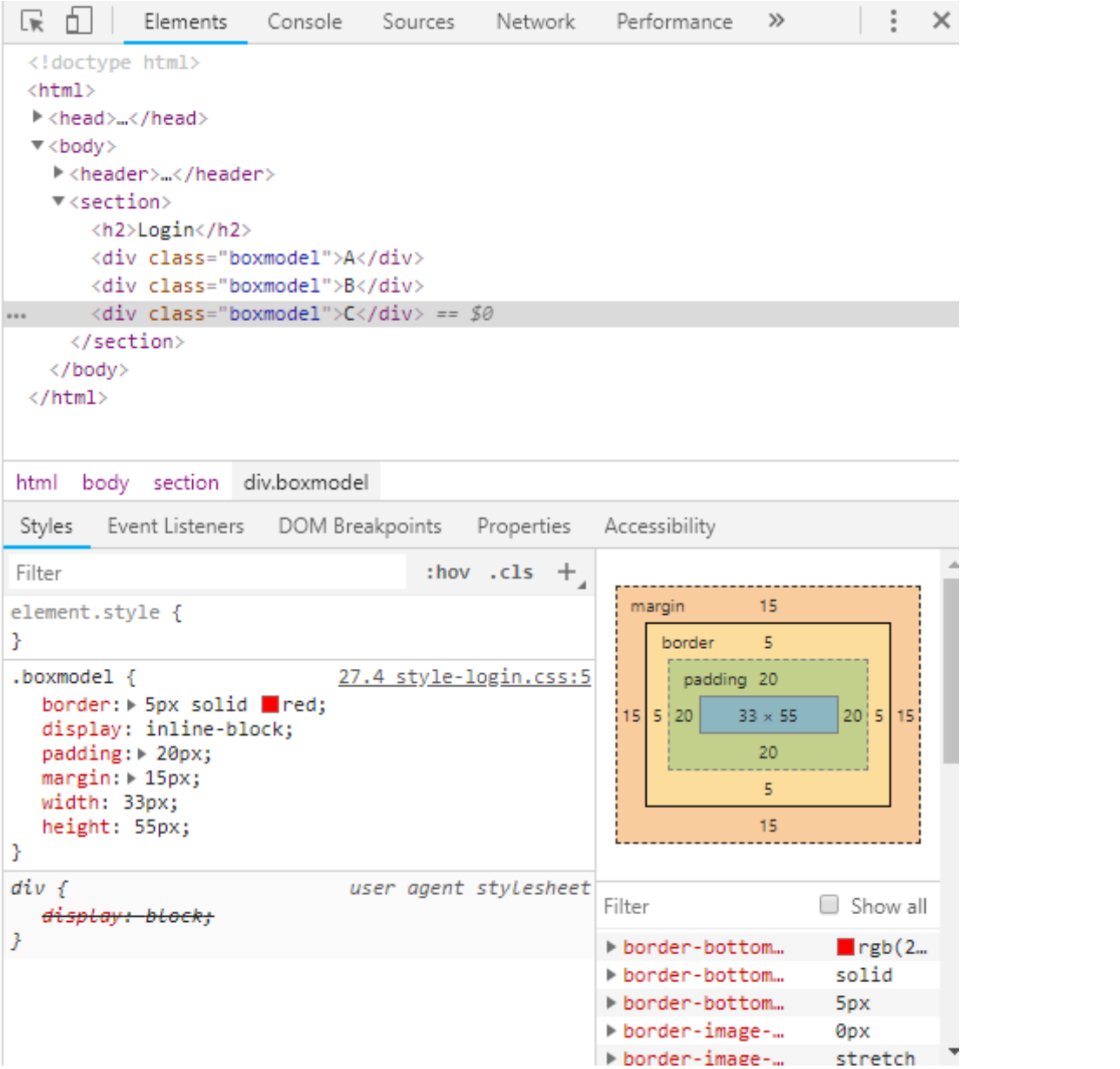


- Home
- About
- Login

Login

A B C

```css
h2{
  color: blue;
}

.boxmodel {
  border: 5px solid red;
  display: inline-block;

  padding: 20px;
  /*all around*/

  margin: 15px;
  /*all around*/

  width: 33px; /*content width*/
  height: 55px; /*content height*/
  
}
```

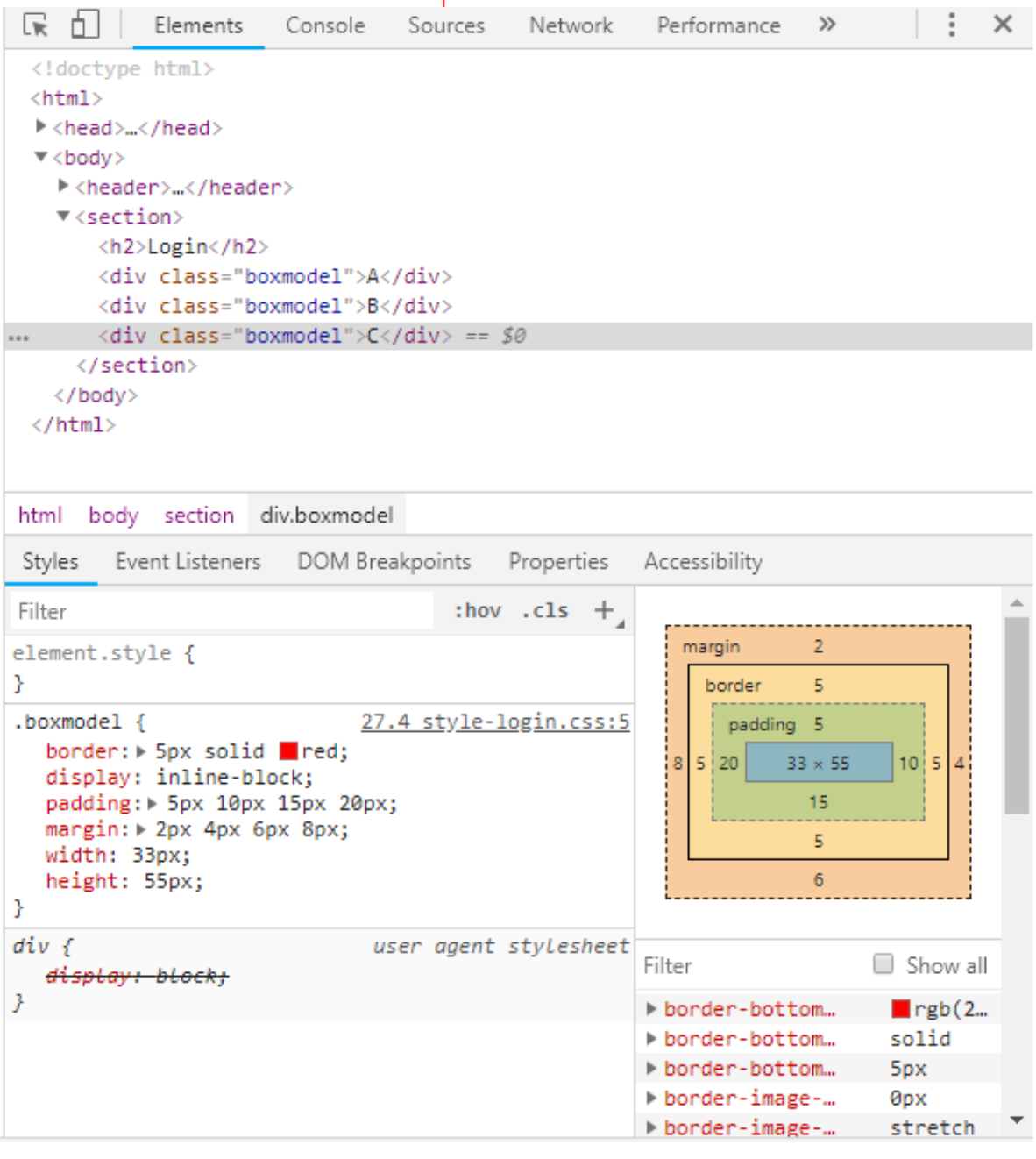


- Home
- About
- Login

Login

A B C

```css
h2{
  color: blue;
}

.boxmodel {
  border: 5px solid red;
  display: inline-block;

  padding: 5px 10px 15px 20px;
  /*top, right, bottom, left*/

  margin: 2px 4px 6px 8px;
  /*top, right, bottom, left*/

  width: 33px; /*content width*/
  height: 55px; /*content height*/

}
```

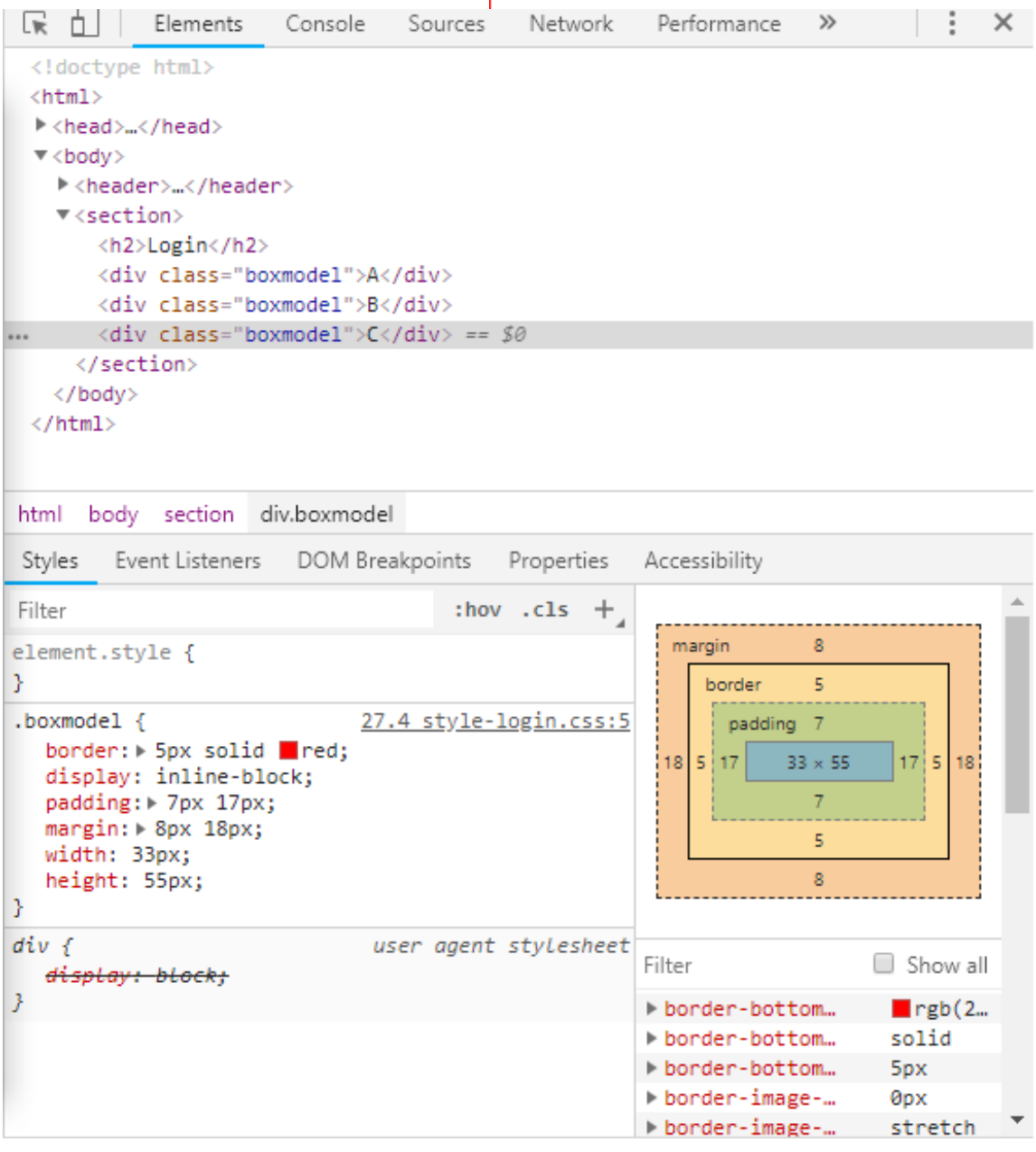


- Home
- About
- Login

Login

A B C

```
h2{
  color: blue;
}

.boxmodel {
  border: 5px solid red;
  display: inline-block;

  padding: 7px 17px;
  /*top/bottom, left/right*/

  margin: 8px 18px;
  /*top/bottom, left/right*/

  width: 33px; /*content width*/
  height: 55px; /*content height*/

}
```



```
/**
 * CONTENTS
 *
 * SETTINGS
 * Global..............Globally-available variables and config.
 *
 * TOOLS
 * Mixins..............Useful mixins.
 *
 * GENERIC
 * Normalize.css........A level playing field.
 * Box-sizing...........Better default `box-sizing`.
 *
 * BASE
 * Headings.............H1–H6 styles.
 *
 * OBJECTS
 * Wrappers.............Wrapping and constraining elements.
 *
 * COMPONENTS
 * Page-head............The main page header.
 * Page-foot............The main page footer.
 * Buttons..............Button elements.
 *
 * TRUMPS
 * Text.................Text helpers.
 */

 /**
 * I am a long-form comment. I describe, in detail, the CSS that follows. I am
 * such a long comment that I easily break the 80 character limit, so I am
 * broken across several lines.
 */




/*-------------------------------------*\
  #A-SECTION
\*-------------------------------------*/

.selector { }
```

```
/*-------------------------------------*\
  #ANOTHER-SECTION
\*-------------------------------------*/

/**
 * This is basically how things should be formatted.
 */

.another-selector { }

.icon {
  display: inline-block;
  width:  16px;
  height: 16px;
  background-image: url(/img/sprite.svg);
}

/*single line css*/
.icon--home     { background-position:   0     0  ; }
.icon--person   { background-position: -16px   0  ; }
.icon--files    { background-position:   0   -16px; }
.icon--settings { background-position: -16px -16px; }


/*alignment*/
.foo {
  -webkit-border-radius: 3px;
     -moz-border-radius: 3px;
          border-radius: 3px;
}

/*alignment*/
.bar {
  position: absolute;
  top:    0;
  right:  0;
  bottom: 0;
  left:   0;
  margin-right: -10px;
  margin-left:  -10px;
  padding-right: 10px;
  padding-left:  10px;
}
```

```
/*-------------------------------------*\
  #HTML-STYLING
\*-------------------------------------*/

/*2 spaces between class attributes in HTML*/
<div class="foo  bar">

/*group related classes with brackets*/
<div class="[ box  box--highlight ]  [ bio  bio--long ]">


/*see single line spacing below*/
<ul class="primary-nav">

  <li class="primary-nav__item">
    <a href="/" class="primary-nav__link">Home</a>
  </li>

  <li class="primary-nav__item  primary-nav__trigger">
    <a href="/about" class="primary-nav__link">About</a>

    <ul class="primary-nav__sub-nav">
      <li><a href="/about/products">Products</a></li>
      <li><a href="/about/company">Company</a></li>
    </ul>

  </li>

  <li class="primary-nav__item">
    <a href="/contact" class="primary-nav__link">Contact</a>
  </li>

</ul>




/*-------------------------------------*\
  #MAIN-PAGE-HEAD
\*-------------------------------------*/

/*This section is about how you comment like there's no tomorrow */

/**
```

```
 * The site's main page-head can have two different states:
 *
 * 1) Regular page-head with no backgrounds or extra treatments; it just
 *    contains the logo and nav.
 * 2) A masthead that has a fluid-height (becoming fixed after a certain point)
 *    which has a large background image, and some supporting text.
 *
 * The regular page-head is incredibly simple, but the masthead version has some
 * slightly intermingled dependency with the wrapper that lives inside it.
 */

 /**
 * Extend `.btn {}` in _components.buttons.scss.
 */

.btn { }

/**
 * These rules extend `.btn {}` in _objects.buttons.scss.
 */

.btn--positive { }

.btn--negative { }


/**
 * Large site headers act more like mastheads. They have a faux-fluid-height
 * which is controlled by the wrapping element inside it.
 *
 * 1. Mastheads will typically have dark backgrounds, so we need to make sure
 *    the contrast is okay. This value is subject to change as the background
 *    image changes.
 * 2. We need to delegate a lot of the masthead's layout to its wrapper element
 *    rather than the masthead itself: it is to this wrapper that most things
 *    are positioned.
 * 3. The wrapper needs positioning context for us to lay our nav and masthead
 *    text in.
 * 4. Faux-fluid-height technique: simply create the illusion of fluid height by
 *    creating space via a percentage padding, and then position everything over
 *    the top of that. This percentage gives us a 16:9 ratio.
 * 5. When the viewport is at 758px wide, our 16:9 ratio means that the masthead
 *    is currently rendered at 480px high. Let's…
 * 6. …seamlessly snip off the fluid feature at this height, and…
 * 7. …fix the height at 480px. This means we should see no jumps in height
 *    as the masthead moves from fluid to fixed. This actual value takes into
 *    account the padding and the top border on the header itself.
```

```
 */

.page-head--masthead {
  margin-bottom: 0;
  background: url(/img/css/masthead.jpg) center center #2e2620;
  @include vendor(background-size, cover);
  color: $color-masthead; /* [1] */
  border-top-color: $color-masthead;
  border-bottom-width: 0;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1) inset;

  @include media-query(lap-and-up) {
    background-image: url(/img/css/masthead-medium.jpg);
  }

  @include media-query(desk) {
    background-image: url(/img/css/masthead-large.jpg);
  }

  > .wrapper { /* [2] */
    position: relative; /* [3] */
    padding-top: 56.25%; /* [4] */

    @media screen and (min-width: 758px) { /* [5] */
      padding-top: 0; /* [6] */
      height: $header-max-height - double($spacing-unit) - $header-border-
width; /* [7] */
    }

  }

}
```

```
/*-------------------------------------*\
  #NAMING-CONVENTIONS
\*-------------------------------------*/

/*%ll strings in classes are delimited with a hyphen*/
.page-head { }
.sub-content { }

/*BEM splits components' classes into three groups:*/
/*Block:    The sole root of the component.*/
/*Element:  A component part of the Block.*/
/*Modifier: A variant or extension of the Block.*/

.person { }
.person__head { }
.person--tall { }

.room { }
  .room__door { }

.room--kitchen { }

.person { }
  .person__head { }
  .person__eye--blue { }
  .person__face--handsome { }/*a handsome face on a regular person

  regular face on a handsome person*/
  .person--handsome .person__face { }



/*For websites...*/
.page { }

.content { }

.sub-content { }

.footer { }
  .footer__copyright { }
```

```
/*-------------------------------------*\
  #NAMING-CONVENTIONS-HTML
\*-------------------------------------*/

/*How are the classes box and profile related to each other? How are the classes
 profile and avatar related to each other? Are they related at all? Should you
 be using pro-user alongside bio? Will the classes image and profile live in the
  same part of the CSS? Can you use avatar anywhere else?*/
/*DONT DO THIS*/
<div class="box  profile  pro-user">

  <img class="avatar  image" />

  <p class="bio">...</p>

</div>

/*Below we can clearly see which classes are and are not related to each other,
 and how; we know what classes we can't use outside of the scope of this
  component; and we know which classes we may be free to reuse elsewhere.*/
/*DO THIS*/
<div class="box  profile  profile--is-pro-user">

  <img class="avatar  profile__image" />

  <p class="profile__bio">...</p>

</div>




/*-------------------------------------*\
  #CSS-SELECTORS
\*-------------------------------------*/

/*----SELECTOR INTENT----*/

/*For example, if you are wanting to style your website's main navigation */
/*menu, a selector like this would be incredibly unwise:*/
/*DONT!*/
header ul { }

/*This selector's intent is to style any ul inside any header element,
whereas our intent was to style the site's main navigation. This is poor
Selector Intent: you can have any number of header elements on a page, and
```

```
they in turn can house any number of uls, so a selector like this runs the
risk of applying very specific styling to a very wide number of elements.
This will result in having to write more CSS to undo the greedy nature of
such a selector.*/

/*DO!*/
.site-nav { }
/*An unambiguous, explicit selector with good Selector Intent. We are
explicitly selecting the right thing for exactly the right reason.*/


/*----REUSABILITY----*/


/*----LOCATION INDEPENDENCE----*/

.promo a { }  /*DONT!*/
.btn { }      /*DO! Can be used anywhere*/


/*----PORTABILITTY----*/

/**
 * Text-level errors.
 */
.error-text {
  color: red;
  font-weight: bold;
}

/**
 * Elements that contain errors.
 */
.error-box {
  padding: 10px;
  border: 1px solid;
}


/*----QUASI-QUALIFIED SELECTORS----*/

ul.nav { }      /*DONT!*/
/*ul*/.nav { }  /*DO!*/
```

```
/*----NAMING----*/

.site-nav      /*NO*/
.primary-nav   /*YES*/


.footer-links  /*NO*/
.sub-links     /*YES*/


 /** Runs the risk of becoming out of date; not very maintainable.*/
.blue {
  color: blue;
}

 /** Depends on location in order to be rendered properly.*/
.header span {
  color: blue;
}

 /** Too specific; limits our ability to reuse.*/
.header-color {
  color: blue;
}

 /** Nicely abstracted, very portable, doesn't risk becoming out of date.*/
.highlight-color {
  color: blue;
}

.home-page-panel  /*NO*/
.masthead         /*YES*/

.site-nav       /*NO*/
.primary-nav    /*YES*/

.btn-login    /*NO*/
.btn-primary  /*YES*/
```