Cascading Style Sheets

Part 3

Responsive UI

Overview

- With the growth in mobile Internet usage comes the question of how to build websites suitable for all users
- The response to this question has become responsive web design, also known as RWD
- Responsive web design is the practice of building a website suitable to work on every device and every screen size, no matter how large or small, mobile or desktop
  - *The responsive web design term was coined & largely developed by Ethan Marcotte (http://ethanmarcotte.com)*

Responsive Sites Vs Mobile Sites

- Responsive websites are not be mixed with Mobile websites
- While *responsive sites* use the same code base (HTML) for desktop users or mobile users, *mobile sites* are developed separately & solely for mobile users
- Responsive sites continually and fluidly change based on different factors like the browser size, etc
- jQuery Mobile is popular a toolkit to develop mobile sites

Responsive Design

- Responsive design is broken down into three main components
  - flexible layouts
  - media queries
  - flexible media

Flexible Layouts

- Flexible layouts is a layout with a flexible grid, capable of dynamically resizing to any width
- Flexible grids are built using relative length units, most commonly percentages or *em* units
- These relative lengths are then used to declare common grid property values such as width, margin, or padding
- Flexible layouts do not advocate the use of fixed measurement units, such as pixels or inches, as the viewport height and width continually change from device to device

Flexible Grid

- To compute the proportions of a flexible layout using relative values, we can use the formula:
  - Target width ÷ context width = result width

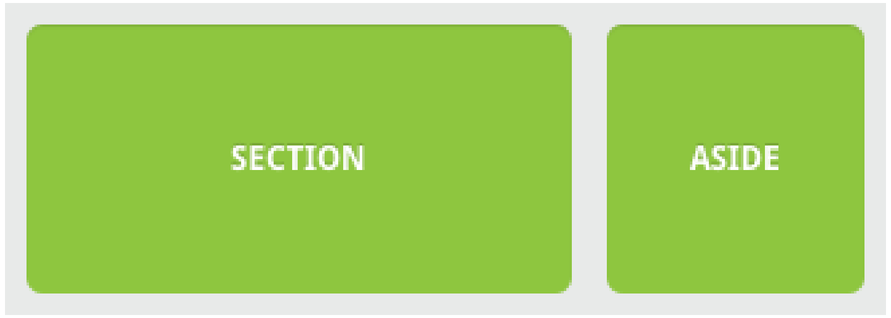
- For example, consider we have a two-column layout:
  - The HTML:

```
<div class="container">
  <section>...</section>
  <aside>...</aside>
</div>
```

Flexible Grid…

- The CSS:

```
.container {
    width: 660px;
}
section {
    float: left;
    margin: 10px;
    width: 420px;
}
aside {
    float: right;
    margin: 10px;
    width: 200px;
}
```



Flexible Grid…

- Using the flexible grid formula, we can take off the fixed unit values with relative units:

```
.container {
    max-width: 660px;
}
section {
    float: left;
    margin: 1.51515151%;   /*  10px ÷ 660px = .01515151 */
    width: 63.63636363%;   /* 420px ÷ 660px = .63636363 */
}
aside {
    float: right;
    margin: 1.51515151%;   /*  10px ÷ 660px = .01515151 */
    width: 30.30303030%;   /* 200px ÷ 660px = .30303030 */
}
```

Media Queries

- Media queries are an extension to media types commonly found when targeting and including styles

- Media queries provide the ability to specify different styles for individual browser and device circumstances, the width of the viewport or device orientation

- Media queries can be used
  - Using **@media** rule inside an existing stylesheet
  - by linking to a separate style sheet from within the HTML document

Initializing Media Queries

- Within HTML:

```html
<!-- Separate CSS File -->
<link href="style.css" rel="stylesheet" media="all and (max-
   width: 1024px)">
```

- Within Stylesheet:

```css
/* @media Rule */
@media all and (max-width: 1024px) {...}
```

  - Generally, it is recommend to use the @media rule inside of an existing style sheet to avoid any additional HTTP requests

Initializing Media Queries…

- Each media query may include a *media type* followed by one or more *expressions*
- Common media types include
  - all
  - screen
  - print
  - tv
  - braille
- If a media type is not specified, the media query will default the media type to screen

Media Query Logical Operators

- There are three logical operators available for use within media queries

  – **and** = allows extra conditions to be added:

```
@media all and (min-width: 800px) and (max-width: 1024px)
    {…}
```

  – **not** = negates the query:

```
@media not screen and (max-width: 320px) {…}
```

  - Applies to any device that is *not* screen type and width is less than 320px

Media Query Logical Operators…

- only = specifies a specific condition:

@media **only** screen **and** **(**orientation**:** portrait**)** **{…}**

  - Selects only screens in a portrait orientation

Media Features

- Media features identify what attributes or properties will be targeted within the media query expression

- Media features include
  - Height & Width Media Features
  - Orientation Media Feature
  - Aspect Ratio Media Features
  - Pixel Ratio Media Features
  - Resolution Media Feature

Height & Width Media Features

- The height and width may be found by using **height**, **width**, **device-height** and **device-width**
  - Each of these features may also be prefixed with the *min* or *max* qualifiers, like min-width or max-device-width
  - The height and width features are based off the height and width of the browser window
  - The device-height and device-width are based off the height and width of the output device, which may be larger than the actual rendering area
  - Example:

```
@media all and (min-width: 320px) and (max-width: 780px)
   {...}
```

Orientation Media Feature

- The orientation media feature determines if a device is in the *landscape* or *portrait* orientation

- Example:

```
@media all and (orientation: landscape) {...}
```

Aspect Ratio Media Features

- The **aspect-ratio** and **device-aspect-ratio** features specifies the width/height pixel ratio of the targeted rendering area or output device
  - The *min* and *max* prefixes are available, identifying a ratio above or below that of which is stated

- Example:

```
@media all and (min-device-aspect-ratio: 16/9) {...}
```

  - The first integer identifies the width in pixels while the second integer identifies the height in pixels

Pixel Ratio Media Features

- The **pixel-ratio** and **device-pixel-ratio** feature is for identifying high definition devices, including retina displays
  - The *min* and *max* prefixes are available, identifying a ratio above or below that of which is stated

- Example:

```
@media only screen and (-webkit-min-device-pixel-ratio: 1.3)
   {...}
```

  - Note the vendor specific property prefix

Resolution Media Features

- The **resolution** media feature specifies the resolution of the output device in pixel density
  - Pixel density is also known as dots per inch or DPI
  - Also accepts *min* and *max* prefixes

- Example:

```
@media print and (min-resolution: 300dpi) {…}
```

  - The values can also be specified in units of
    - Dots per pixel (dppx)
    - Dots per centimeter (dpcm)

Media Query Browser Support

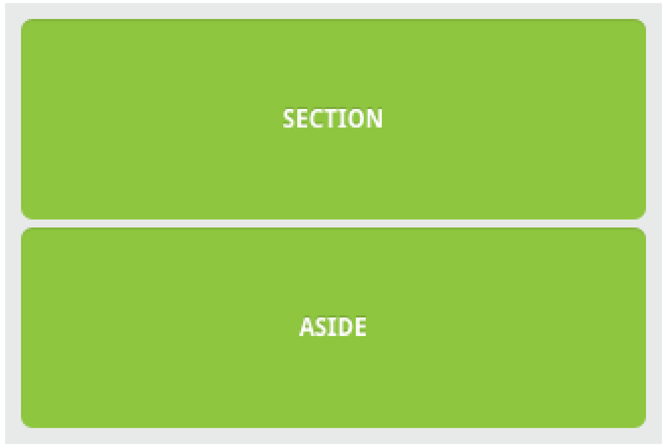
- Media queries do not work with legacy browsers

- Some of the workaround available are:
  - Respond.js = a JavaScript based solution that adds min/max-width query feature to legacy browsers
  - CSS3-MediaQueries.js = a JavaScript solution offers support for a larger array of more complex media queries

Using Media Queries

- Using media queries we will now rewrite the flexible layout we built previously
  - One of the problems with the layout is that with smaller viewports, the aside next to section is uselessly small
  - Adding a media query for viewports under 420 pixels wide we can change the layout by turning off the floats and changing the widths of the section and aside

```
@media all and (max-width: 420px) {
	section, aside {
		float: none;
		width: auto;
	}
}
```



Mobile First

- The *mobile first* approach is about using styles targeted at smaller viewports as the default styles for a website, then use media queries to add styles as the viewport grows
  - The operating belief is that a user on a mobile device should't have to load the styles for a desktop computer, only to have them over written with mobile styles later and wastage of precious bandwidth

- A breakup of mobile first media queries might look like:

```
/* Default styles first then media queries */
@media screen and (min-width: 400px)  {...}
@media screen and (min-width: 600px)  {...}
@media screen and (min-width: 1000px) {...}
```

Mobile First…

- Also, downloading unnecessary media assets can be stopped by using media queries

```
/* Default media */
body { background: #ddd; }

/* Media for larger devices */
@media screen and (min-width: 800px) {
   body { background: url("bg.png") 0 0 no-repeat; }
}
```

Viewport

- The **viewport** is a meta tag that tells the browser how to behave when it renders the webpage
  - Tell it how big the viewport will be
  - Set the zoom level
  - Prevent zooming by user, etc

- In HTML, add it as a meta tag with the syntax:

```
<meta name="viewport" content="specifications...">
```

Viewport Width & Height

- Using the *viewport* meta tag with either width or height values will define the width or height of the viewport

```
<meta name="viewport" content="width=320">
```

  - Sets the width to 320px

- Keyword values *device-width* and *device-height* inherits the device's default width and height value

```
<meta name="viewport" content="width=device-width">
```

Viewport Scale

- To control the zooming level on a mobile device use one of the properties:
  - minimum-scale : can take values from 0 to 10
  - maximum-scale : can take values from 0 to 10
  - initial-scale : can take values from 0 to 10
  - user-scalable : can take values ‘no’ or ‘yes’. No will disable any zooming by the user

```
<meta name="viewport" content="initial-scale=1">
```

  - In portrait orientation, defines the ratio between device height and the viewport size
  - In landscape mode it would be the ratio between the device width and the viewport size

Viewport Resolution

- The **target-densitydpi** specifies the resolution to the device
- The target-densitydpi viewport accepts value:
  - device-dpi
  - high-dpi
  - medium-dpi
  - low-dpi
  - an actual DPI number

```
<meta name="viewport" content="target-densitydpi=device-
dpi">
```

Combining Viewport Values

- The viewport meta tag accepts individual values as well as multiple values, allowing multiple viewport properties to be set at once
  - Setting multiple values requires comma separating them within the content attribute value
  - One of the recommended viewport values is outlined below, using both the width and initial-scale properties

```
<meta name="viewport" content="width=device-width,
   initial-scale=1">
```

Flexible Media

- Media size does not change as the viewports change
  - Images, videos, and other media types need to be scalable, changing their size as the size of the viewport changes

- One quick way to make media scalable is by using the max-width property with a value of 100%
  - As the viewport gets smaller any media will scale down according to its containers width:

```
img, video, canvas {
    max-width: 100%;
}
```

LESS CSS

CSS

- CSS is very simple and straight forward
- However, it has its own drawback: **CSS is static**

CSS with LESS

- LESS is a dynamic stylesheet language
  - It extends CSS with variables, mixins, operations, functions, etc., forcing CSS to behave more like a programming language
- LESS is a CSS preprocessor
  - CSS preprocessing is a method of writing the stylesheets in an extended language
  - Then compiling the code to pure CSS so that its readable by browsers
- LESS can be run on the client or server side (with Node.js)
- The other popular preprocessors are Sass & Stylus

Advantages with LESS

- Since LESS allows usage of variables, functions, etc., the primary benefits are
  - Less code to write (for developers / designers)
  - Easier maintenance of your stylesheets

- The LESS syntax is almost identical to CSS, hence, it is easier and faster to learn

How does LESS work?

- Instead of writing CSS stylesheets, you write LESS stylesheets
- The LESS stylesheet files have the extension “less”

- The LESS stylesheets are compiled by LESS compilers to generate CSS

- The LESS compiler is written in JavaScript
- To compile LESS on server side, use Node.js
- To compile LESS on client side, use the less.js compiler provided at lesscss.org

Getting Started

- Download the LESS compiler from lesscss.org
  - We will use the LESS compiler on client side

- Include the LESS stylesheet using the <link>

  ```
  <link rel="stylesheet/less" type="text/css" href="style.less" />
  ```

  - The rel should be set as **stylesheet/less**
  - Note the extension of LESS stylesheet file is .less

- Include the LESS JavaScript compiler

  ```
  <script type="text/javascript" src="less-1.7.0.min.js"></script>
  ```

  - The JavaScript compiler should always be included after the LESS stylesheet inclusion

Commenting

- Multiline comments in LESS are similar to comments in CSS

```
/**
 * multi-line comments
 * are similar to as in CSS
 */
```

- Single-line comments are “silent”, they don’t show up in the compiled CSS output

```
  // LESS single comment example
```

Variables

- Variables are declared and used with @ symbol
  - Variables have name and a value assigned to it
  - Once declared, you can refer to them any where in your .less file

```
@name: value;                        // syntax
@color: red;                         // example
background-color: @color;            // usage
background-color: red;               // output
```

Variables Example

```
@primaryColor: #c80000;
@border-thick: 3px;
@border-thin: 1px;

.class1 {
    color: @primaryColor;
}

.class2 {
    border: @border-thick solid @primaryColor;
    margin: 12px 0;
}
```

Variable Scope

- What is the output background color of both the elements?

```
@color: #c80000;
.class1 {
   background-color: @color;
}

.class2 {
   @color: #006600;
   background-color: @color;
}
```

- What is the scope of variables now?

Mixins

- Mixins are used to group CSS instructions as reusable classes
  - They are similar to what functions are in programming languages

```
@primaryColor: #c80000;
@radius: 6px;
.RoundBorders {
    border: 3px solid @primaryColor;
    border-radius: @radius;
    -moz-border-radius: @radius;
    -webkit-border-radius: @radius;
}

.class1, .sidebar-block {
    color: #333;
    .RoundBorders;
}
```

  - The mixin name can be a class-name or an id-name, but, usually a class-name

Nested Rules

- In CSS, hierarchical rulesets are written separately
  - Which leads to long selectors that repeat parts over and over

```
header {…}
header nav {…}
header nav ul {…}
header nav ul li {…}
header nav ul li a {…}
```

LESS Nested Rules

- In LESS you can simply nest selectors inside other selectors
  - This makes inheritance clear and stylesheets shorter

- Example

```
#header {
   color: black;
   .navigation { font-size: 12px; }
   .logo { width: 300px; }
}
```

Nested Rules…

- **&** operator represents the parent's selector
  - Its used when you want a nested selector to be concatenated to its parent selector

- Example:

```
.class1 {
    color: black;
    a { text-decoration: none;
      &:hover { text-decoration: underline; }
      &:visited { color: grey; }
    }
}
```

Math Operators

- Any number, color or variable can be operated on, using the math operators
  - Less understands the difference between colors and units
  - If a unit is used in an operation, it will use that unit for the final output
- Example 1:

```
.class1 { background-color: #c80000; }
.class2 { background-color: #c80000 / 4; } // math operator
```

- Example 2:

```
@sidebarWidth: 400px;
@sidebarColor: #FFCC00;
#sidebar {
    color: @sidebarColor + #FFDD00;
    width: @sidebarWidth / 2;
    margin: @sidebarWidth / 2 * 0.05;
}
```

Math Functions

- LESS provides a bunch of handy math functions
  - round(1.67); // returns 2
  - ceil(2.4);   // returns 3
  - floor(2.6); // returns 2
  - percentage(0.5); // returns 50%

- The 960gs example:

```
@baseWidth: 960px;
@mainWidth: round(@baseWidth / 1.618);
@sidebarWidth: round(@baseWidth * 0.382);

#main {  width: @mainWidth; }

#sidebar { width: @sidebarWidth; }
```

Variable Interpolation

- Interpolation is a convenient way to insert the value of a variable right into a string literal
- Variables can be embedded inside strings with the @{name} construct

- Example:

```
// Variable
@images: "../img";

// Usage
body { background: url("@{images}/white-sand.png"); }

// Compiles to
body { background: url("../img/white-sand.png"); }
```

Variable Interpolation Example

- Example:

```
// variable
@mySelector: banner;

// Usage
.@{mySelector} { margin: 0 auto; }

// compiles to
.banner { margin: 0 auto; }
```

What is happening here?

- Consider this example:

```
@size: 20px/80px;
body {
    // desired output is
    // font: 20px/80px sans-serif;
    font: @size sans-serif;
} // what is the output now?
```

  – Is the content visible?
  – Does the content exist in the page? (See source & verify)

Escaping

- ~ operator is used to escape the values
  - To escape a value prefix the ~ operator just before the string

- Example:

```
// escape the value with ~ & quotes around
@size: ~"20px/80px";

body {
    font: @size sans-serif;
}
```

Mixins

- Recall that mixins are used to group CSS instructions as reusable classes
  - They are similar to what functions are in programming languages

```
@primaryColor: #c80000;
@radius: 6px;
.RoundBorders {
    border: 3px solid @primaryColor;
    border-radius: @radius;
    -moz-border-radius: @radius;
    -webkit-border-radius: @radius;
}

.class1, .sidebar-block {
    .RoundBorders;
}
```

Mixin Arguments

- Mixins can take arguments

```
.RoundBorders (@radius) {
    border-radius: @radius;
     -moz-border-radius: @radius;
     -webkit-border-radius: @radius;
}

.class1 {
    border: 3px solid #c80000;
    .RoundBorders(6px);
}

.submit {
    .RoundBorders(9px);
}
```

Mixin Multiple Arguments

- Multiple arguments can be passed to mixins as comma separated list of variables or values

```
.Border (@thickness, @style, @color) {
    border: @thickness @style @color;
}

.class1 {
    .Border(3px, solid, #c80000);
}

.class2 {
    .Border(6px, dotted, #666);
}
```

Mixin Argument Default Value

- The arguments of Mixins can have default value too
  - When the value is passed during mixin call, it uses the passed value
  - Else it will use the default value

```
.RoundBorders (@radius: 6px) { // default value set
    border-radius: @radius;
    -moz-border-radius: @radius;
    -webkit-border-radius: @radius;
}

.class1 {
    border: 3px solid #c80000;
    .RoundBorders; // uses default value
}

.submit {
    .RoundBorders(9px); // uses passed value
}
```

Mixin @arguments

- All the arguments passed to a mixin is stored in a special variable called @arguments
  - It can be used to quickly assign all the values

```
.BoxShadow(@x: 0, @y: 0, @blur: 1px, @color: #000) {
    box-shadow: @arguments; // contains passed args or default values
    -moz-box-shadow: @arguments;
    -webkit-box-shadow: @arguments;
}

.class1 {
    .BoxShadow(2px, 5px);
}
```

Polymorphic Mixin

- Mixins with the same name can be declared differently to give different outputs
  - Similar to polymorphic methods / functions
  - It works by checking all the available mixins of that name, in the current scope, to see if they match according to what was passed to the mixin and how it was declared

  - The simplest case of matching is done with the count of arguments that a mixin takes
  - Example:

Polymorphic Mixin…

```
.mixin { font-weight: bold; }

.mixin(@a) { background-color: @a; }

.mixin(@a, @b) { padding: @a; margin: @b; }


.class1 { .mixin(yellow); }

.class2 { .mixin(8px, 12px); }

.class3 {
    .mixin;
    .mixin(yellow);
    .mixin(8px, 8px);
}
```

Polymorphic Mixin…

- Another way of controlling whether a mixin matches, is by specifying a value in place of an argument name when declaring the mixin

```
.mixin(darkness, @color: #990066) {
    color: @color;
    font-weight: bold;
}

.mixin(lightness, @color: #ff9966) {
    color: @color;
    font-weight: normal;
}

.class1, .class2 {
    .mixin(darkness);
}
```

Mixin Guards

- Another way of restricting when a mixin is mixed in, is by using guards
- A guard is a special expression that is associated with a mixin declaration that is evaluated during the mixin process
- It must evaluate to true before the mixin can be used
- **when** keyword is used to begin describing a list of guard expressions

Mixin Guards…

- Syntax

```
.mixinClass( @variable ) when ( conditions )  {
    prop1: value1;
    prop1: value1;
}

.implementationClass {
    .mixinClass( arguments );
}
```

Mixin Guards Example

- Example

```
.mixin(@size) when (@size > 10) {
    color: red;
    font-weight:bold;
}

.mixin(@size) when (@size <= 10) {
    color: green;
    font-weight: normal;
}

.class1 { .mixin(5); } // font is green & normal

.class2 { .mixin(15); } // font is red & bold
```

- The comparison operators that can be used are >, >=, =, =<, <

Mixin Guards Type Checking

- To match mixins based on value type, you can use the is* functions
  - Checks based on the type:
    - iscolor
    - isnumber
    - isstring
    - isurl
  - Check if a value is in a specific unit in addition to being a number:
    - ispixel
    - ispercentage
    - isem

Mixin Guards Type Checking Example

- Example

```
.mixin (@a) when (iscolor(@a)) {
   color: @a;
}

.mixin (@a) when (ispixel(@a)) {
   width: @a;
}

.class1 { .mixin(black); } // passing a color

.class2 { .mixin(960px); } // passing pixel val
```

Mixin Guards Multiple Conditions

- Guards can be separated with a comma – if any of the guards evaluates to true, it's considered a match:

```
.mixin (@a) when (@a > 10), (@a < -10) { ... }
```

- Use the **and** keyword so that all of the guards must match in order to trigger the mixin:

```
.mixin (@a) when (isnumber(@a)) and (@a > 0) { ... }
```

- Use the not keyword to negate conditions:

```
.mixin (@b) when not (@b > 0) { ... }
```

Conditional Mixin

- The default function may be used to make a mixin match depending on other mixing matches
  - similar to else or default statements

- Example:

```
.mixin(@size) when (@size > 10) { color: red; }

.mixin(@size) when ( default() ) { color: green; }

.class1 { .mixin(15); }
.class2 { .mixin(5); } // default() is applied
```

Creating Loops with Mixins

- Mixins are recursive (can call itself) and this feature can be used to create loop structures
- Example:

```
.mixin(@n, @i: 1) when (@n >= @i) {
    .column-@{i} {
      width: (@i * 100px / @n),
    }
    .mixin(@n, (@i + 1)); // calling itself, iterator
}

.mixin(4); // generates .column-1, .column-2, etc
```

Performance & Optimization

Performance

- The performance of a web application has a direct impact on the user experience

- In this section, we are going to look at:
  - Impact of using different methods in animations effect the browser performance
  - How the CSS code can be optimized to make the overall application performance better

How Browsers Work?

- Any browser performs a list of duties:
  - Making HTTP calls
  - Receiving HTTP responses
  - Caching files (received from responses)
  - Interpretting / parsing HTML, CSS & JavaScript
  - Generating required output

- A browser primarily uses the CPU to interpret / parse HTML, CSS & JavaScript
  - Interpreting & generating HTML is usually a one-time task
  - With CSS & JavaScript, there are repetitive tasks involved

CSS Parsing

- To parse CSS, a browser does
  - Recalculate styles = calculate the styles that apply to the elements
  - Layout = generate the geometry and position for each element
  - Paint Setup and Paint = fill out the pixels for each element into layers
  - Composite Layers = draw the layers out to screen
- The browser uses a thread (main thread) to handle tasks like running JavaScript, calculate styles, layout, painting and compositing
- Certain tasks are offloaded from the main thread to the GPU to handle compositing

Handling Animations

- The compositing process is where all the individual textures are uploaded to the GPU, then drawn out to create the final picture output on the screen
  - It all happens behind the scenes before a single frame of content is presented

- When an element is animated, the browser has to
  - Re-evaluate the page, frame by frame, to determine if and how the element has affected the flow of nearby elements & the overall page
  - Then recalculate the positions and geometries of those elements
  - This process is called reflow

Handling Animations…

- The browser also looks for areas that need to be **re-repainted** due to changes in an element’s color or background properties
- Every time a repaint or reflow happens, the browser has to composite those areas all over again
- These reflows and repaints can be “expensive,” making our animations and transitions appear jarred
- The key to optimized animation performance is keeping the number of areas being reflowed and repainted as low as possible

Testing Animations & Transitions

- Its always important to test the performance of an animation or transition in order to see exactly how much work the browser is doing

- Using the **Timeline** panel and rendering settings of *Chrome DevTools*, you can test transition/animation methods to determine which one will always give the smoothest results

Test Animation

- Consider this: an image has to be moved from its current position to a new position and return it back to original position

- This task can be achieved by the keyframes animation. But there are two approaches that you can take to animate it:
  1. Use position relative / absolute & manipulate the top & left
  2. Use transform: translate(x, y)

Animating Layout Properties

- When elements change
  - The browser may need to do a layout, which involves calculating the geometry (position and size) of all the elements affected by the change
  - For example, if you change the width of the <body> element any of its children may be affected
  - Changes further down the tree can sometimes result in layout calculations all the way back up to the top.
  - The larger the tree, the longer it takes to perform layout calculations

- So avoid animating properties that trigger layout
  - Like width, height, padding, margin, display, border, position, font size & weight, text-align, etc

Animating Paint Properties

- Changing an element may also trigger painting
  - Other elements *besides* the one that changed may also need to be painted
- Styles that affect paint:
  - Color, border-style, background, visibility, text-decoration, border-radius, box-shadow, etc
- On mobile devices this is particularly expensive because CPUs are significantly less powerful than on desktop
  - Painting work takes longer
  - And the bandwidth between the CPU and GPU is limited, so texture uploads take a long time

Optimizing Animations

- Browsers create a new thread / layer for any element which has a CSS transition or animation & hand it to the GPU for processing

- Going back to our test animation, in approach 1, we use position property along with top & left to move the element
- Position, top & left, all trigger layout operations, which is expensive

- The better solution is to use translate on the element
  - As it does not trigger layout operations
  - As it is passed on to the GPU, the CPU is not overloaded

Imperative Vs Declarative Anims

- Developers often have to decide if they will animate with JavaScript (imperative) or CSS (declarative)

- JavaScript animations:
  - The main disadvantage is that it runs in the browser’s main thread
  - The main thread is already busy with other JavaScript, style calculations, layout and painting
  - This substantially increases the chance of missing animation frames
  - However, JavaScript also gives a lot of control: starting, pausing, reversing, interrupting, cancelling. Some effects, can only be achieved in JavaScript

Imperative Vs Declarative Anims…

- CSS animations: The alternative approach to write transitions and animations
  - The primary advantage is that the browser can optimize the animation
  - It can create layers if necessary, and run some operations off the main thread
  - But CSS animations lack the expressive power of JavaScript animations
  - It is very difficult to combine animations in a meaningful way, which means authoring animations gets complex and error-prone

General CSS Optimization

- CSS can also be optimized for overall performance with:
  - Prefer using classes to ids
  - Optimize CSS selectors
    - For ID selectors, select with *#wrapper* rather than *div#wrapper*
  - Use CSS shorthand
  - Reduce number of line breaks
    - prefer single line declarations
    - Remove blank lines
    - Skip last semi-colon in declarations
  - Use simple comments
  - Change 0px to 0
  - Remove unused classes
  - Group similar styles

General CSS Optimization…

- CSS can also be optimized for overall performance with:
  - Use CSS Sprites
  - Aggregate & combine CSS files into one
  - Make CSS an external file
  - Prefer putting CSS in <head>
  - Use <link> instead of @import
  - Use compression tools

CSS3 Image Filters

CSS3 Image Filters

- CSS3 provides a list of effects that can be applied on images:
  - Greyscale
  - Blur
  - Saturate
  - Sepia
  - Hue Rotate
  - Invert
  - Brightness
  - Contrast

Using Filters

- Syntax:

```
img {
    filter: type(value);
}
```

- Vendor specific syntax:

```
img {
    filter: type(value);
    -webkit-filter: type(value);
    -moz-filter: type(value);
    -ms-filter: type(value);
    -o-filter: type(value);
}
```

  – Presently, only webkit has full support for the filters

Grayscale

- Grayscale filter converts the colour in images to a shade of gray

```
img {
    filter:grayscale(.7);
    -webkit-filter: grayscale(.7);
}
```

  - The argument value can be either decimal or percentage
  - 100% or 1.0 will make the image full greyscaled
  - 0% or 0 will add no effect of greyscale to the image

Blur

- The blur() filter blurs the image

```
img {
    filter: blur(5px);
    -webkit-filter: blur(5px);
}
```

  - Blur is measured in pixels
  - More the pixels, more the blur

Saturate

- Saturate adds colour saturation to the colours in images

```
img {
    filter: saturate(500%);
    -webkit-filter: saturate (500%);
}
```

  - The argument value can be either decimal or percentage
  - 100% gives normal saturation
  - Values more than 100% will add more saturation (brightens colors)
  - Values less than 100% will reduce saturation (move towards gray)

Sepia

- Sepia adds a sepia tint to the images
  - Sepia tint is a type of monochromatic photographic image in which pictures appear in shades of reddish brown as opposed to grayscale in black & white pictures

```
img {
    filter: sepia(100%);
    -webkit-filter: sepia(100%);
}
```

  - The argument value can be either decimal or percentage
  - 100% or 1.0 is full sepia
  - 0% or 1 is no sepia at all

Hue Rotate

- The hue rotate property changes the colour around to be completely different depending of the degree value provided
  - Think of it like a colour spectrum wheel, the colour that it's starting at will take the degree value of the hue rotate and use the new colour instead

```
img {
    filter: hue-rotate(180deg);
    -webkit-filter: hue-rotate(180deg);
}
```

  - The value is given in degrees
  - Normal is 0 degrees
  - With a value of 360 degrees the spectrum goes full circle and will be exactly the same

Invert

- The invert effect inverts the colors in the image
  - Gives the same effect as the negatives in photographic films

```
img {
   filter: invert(1);
   -webkit-filter: invert(1);
}
```

  - The argument value can be either decimal or percentage
  - 100% or 1.0 will give full invert
  - 0% or 0 will be normal

Brightness

- Brightness effect changes the brightness applied to the image

```
img {
   filter: brightness(200%);
   -webkit-filter: brightness(200%);
}
```

  - The argument value can be either decimal or percentage
  - 100% is default image brightness
  - 0% will make the image black
  - Values higher than 100% will move the colors towards white

Contrast

- Contrast changes the difference between the lightest and darkest colors of the image

```
img {
   filter: contrast(1.5);
   -webkit-filter: contrast(1.5);
}
```

  - The argument value can be either decimal or percentage
  - Default image contrast is 1
  - To make it darker use a value less than 1, to make it brighter you change the value more than 1.

Image Reflection in Webkit

- In Webkit browsers, reflections can be created using the **-webkit-box-reflect** property

- Syntax:

  -webkit-box-reflect: <direction> <offset> <mask-box-image>;

  – Direction can have a value that is
    - above - The reflection appears above the border box
    - below - The reflection appears below the border box
    - left - The reflection appears to the left of the border box
    - right - The reflection appears to the right of the border box

Image Reflection in Webkit…

- Syntax:

  -webkit-box-reflect: <direction> <offset> <mask-box-image>;

  - Offset defines the distance of the reflection from the border of the box
  - Mask box image is used to overlay the box image reflection

Image Reflection Example

- The HTML:

```
<div class="reflect">
    <img src="seasurf.jpg" alt="surf">
</div>
```

- The CSS:

```
.reflect {
    -webkit-box-reflect: below 10px; /* direction, offset */
}
```

  - Creates image reflection below with the offset 10px

Image Reflection Example…

- Adding gradient mask:

```
.reflect {
    -webkit-box-reflect: below 10px

    -webkit-gradient(linear, left top, left bottom,
    from(transparent), color-stop(20%, transparent),
    to(rgba(255, 255, 255, 0.5)));
}
```

Summary