Full-Stack developer

Roee Angle

August 2020

Contents

1 HTML/CSS

1.1 HTML Fundamentals

HTML is the standard markup language for Web pages. With HTML you can create your own Website. lets see some of the markup:

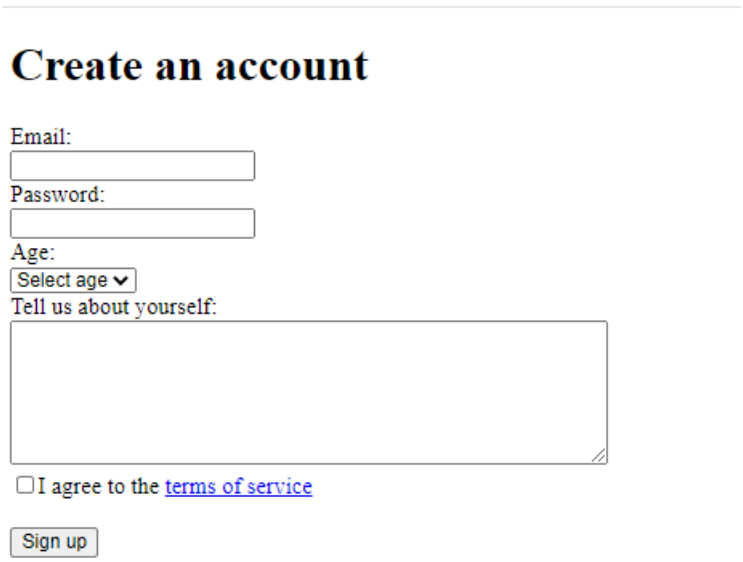
- <!DOCTYPE> It is an "information" to the browser about what document type to expect.
- <Head> element is a container for metadata (data about data) and is placed between the <html> tag and the <body> tag.
- <Body> This element contains all the contents of an HTML document, such as headings, paragraphs, images, hyperlinks, tables, lists, etc.
- <Meta> Defines metadata about an HTML document.
- <Title> Defines the title of the document.

- `<h1/2/3/4/5/6>` Headings.
- `<p>` Paragraph.
- `<br>` New line.
- `<hr>` Unerline.
- `<img>` Image.
- `<a>` Anchor tag.
- `<ul> + <li>` List.
- `<table> + <thead> + <tr> + <tbody> <td>` Table.
- `<div>` Division or a section in an HTML document.
- `<span>` This is a inline container used to mark up a part of a text, or a part of a document.
- `<form>` Is used to create an HTML form for user input.

1.1.1 Simple example for the `<form>` tag:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>Create an account</h1>
    <div>
        <label for="Email">Email:</label><br>
        <input type="email" id="Email">
    </div>
    <div>
        <label for="Password">Password:</label><br>
        <input type="password" id="Password">
    </div>
    <div>
        <label for="Age">Age:</label><br>
        <select name="age" id="age">
            <option value="-1">Select age</option>
            <option value="0-15">0-15</option>
            <option value="16-30">16-30</option>
            <option value="31-50">31-50</option>
            <option value="51+">51+</option>
        </select>
    </div>
    <div>
        <label for="message">Tell us about yourself:</label><br>
        <textarea name="message" id="message" cols="50" rows="6"></textarea>
    </div>
    <div>
        <input type="checkbox" id="terms">I agree to the <a href="#">terms of service</a>

    </div>
    <br>
    <div>
        <button>
            Sign up
        </button>
    </div>
</body>
</html>
```



1.2 CSS Fundamentals

1.2.1 Basic CSS

Without the stylesheet, the webpage will look ugly. CSS is a language that describes the style of an HTML document. CSS describes how HTML elements should be displayed. For each tag, we can determine for him a class or an id. This helps us to style the specific tag. We can style inside the HTML file or in a separate file.

```
/* Body styling */
body {
  background-color: #333;
}

/* . is for classes */
.primary-heading {
  color: blue;
}

/* # is for IDs */
#welcome {
  background-color: #f4f4f4;
}
```


```
/* Body styling */
body {
  background-color: #333;
}

/* . is for classes */
.primary-heading {
  color: blue;
}

/* # is for IDs */
#welcome {
  background-color: #f4f4f4;
}
```


1.2.2 Fonts

The CSS font properties define the font family, boldness, size, and the style of a text.

- `font-size:` Text size
- `line-height:` The space between lines
- `font-weight:` Text weight
- `font-style:` change the text style

1.2.3 Background

- `background-color:` Change the background color

- `background-image:` Change the background image
- `background-repeat:` Repeat the background image
- `background-size:` Specifies the size of the background images.
- `background-attachment:` Sets whether a background image scrolls with the rest of the page, or is fixed.

1.2.4 Border

This property border the tag with a border color, border width and border style.

- `border-width:` Border width
- `border-color:` The color of the border
- `border-style:` How the border look like
- `border:` 1 line for the three properties.

1.2.5 Box Model, Margin and Padding:

First, lets remember that each browser have there own default properties. So first thing first is to reset the css margin padding and box model.

```
/* CSS Reset */
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}
```

- `padding:` Used to generate space around an element's content, inside of any defined borders.
- `margin:` Used to create space around elements, outside of any defined borders.
- `box-sizing:` Defines how the width and height of an element are calculated: should they include padding and borders, or not.

1.2.6 Float and alignment:

The float property can be used to align an entire block element to the left or right such that other content flows around it. You can use the text-align property to define whether the content of a block element is aligned to the left, right, in the center or justified to both margins.

1.2.7 Inline, Block and Inline-Block Display

The display property is one of the most commonly used features of CSS development. Our web page treats every HTML element as a box, and with the display property, we determine how these boxes will be shown, or whether to show or hide them.

Inline elements Take only as much space as they need Displayed side by side Don't accept width or height properties, and top-bottom margin Can be a parent of other inline elements

Inline-block As we can understand from its name, display: inline-block declaration shows both the characteristics of inline and block-level elements.

Block vs. Inline

Have you ever noticed that some HTML tags like `<div>, <p>, <ul>` take full-width of space and each starts with a new line, whereas other HTML tags like `<span>, <img>` or `<a>` don't need a new line and can be placed side by side? This is because of the different display behaviors: Block or inline.

- `inline:` The element generates one or more inline element boxes that do not generate line breaks before or after themselves. In normal flow, the next element will be on the same line if there is space
- `block:` The element generates a block element box, generating line breaks both before and after the element when in the normal flow.
- `inline-block:` The element generates a block element box that will be flowed with surrounding content as if it were a single inline box.

1.2.8 Position

The position CSS property sets how an element is positioned in a document. The top, right, bottom, and left properties determine the final location of positioned elements.

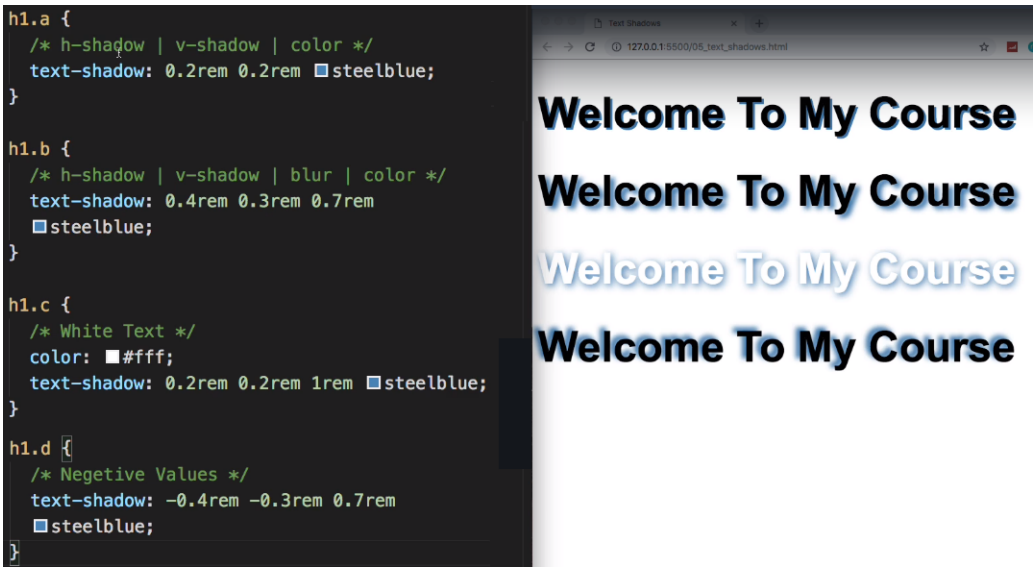
Static	Not effected by tblr(top, bottom, left, right) properties/values
Relative	tblr values cause element to be moved from its normal position
Absolute	Positioned relative to its parent element that is positioned “relative”
Fixed	Positioned relative to the viewport
Sticky	Positioned based on scroll position

1.2.9 Box-shadow and Text-shadow

Box-shadow: The box-shadow CSS property adds shadow effects around an element’s frame. You can set multiple effects separated by commas. A box shadow is described by X and Y offsets relative to the element, blur and spread radius, and color.

```
/* offset-x | offset-y | color */
box-shadow: 10px 10px teal;
/* offset-x | offset-y | blur-radius |
color */
box-shadow: 5px 5px 20px teal;
/* Negative values */
box-shadow: -5px -5px 20px teal;
/* offset-x | offset-y | blur-radius |
spread-radius | color */
box-shadow: 3px 3px 10px 1px rgba(0,0,0,
0.3);
/* inset | offset-x | offset-y | color */
box-shadow: inset -3px -3px teal;
/* Multiple Shadows */
box-shadow: 3px 3px 10px teal, -3px -3px
10px olive;
```

Text-shadow The text-shadow CSS property adds shadows to text. It accepts a comma-separated list of shadows to be applied to the text and any of its decorations. Each shadow is described by some combination of X and Y offsets from the element, blur radius, and color..



1.2.10 Responsive design

Responsive web design makes your web page look good on all devices. The layout changes based on the size and capabilities of the device. For example, on a phone users would see content shown in a single column view; a tablet might show the same content in two columns.

```
@media (max-width: 400px) {
    html { background: red; }
}
@media (min-width: 401px) and (max-width: 800px) {
    html { background: green; }
}
@media (min-width: 801px) {
    html { background: blue; }
}
```

1.2.11 CSS Units

Many CSS properties like width, margin, padding, font-size etc. take length. CSS has a way to express length in multiple units. Length is a combination of a number and unit with no whitespace. E.g. 5px, 0.9em etc.

- `rem` "r" stands for "root": "root em" -, which is equal to the font size fixed to the root element
- `vh and vw` Many responsive web design techniques rely heavily on percentage rules.
- `vmin and vmax` These units are related to the maximum or minimum value of vh and vw.
- `em` the width of a capital letter M of the font-size of the current element. Font sizes are inherited from parent elements.

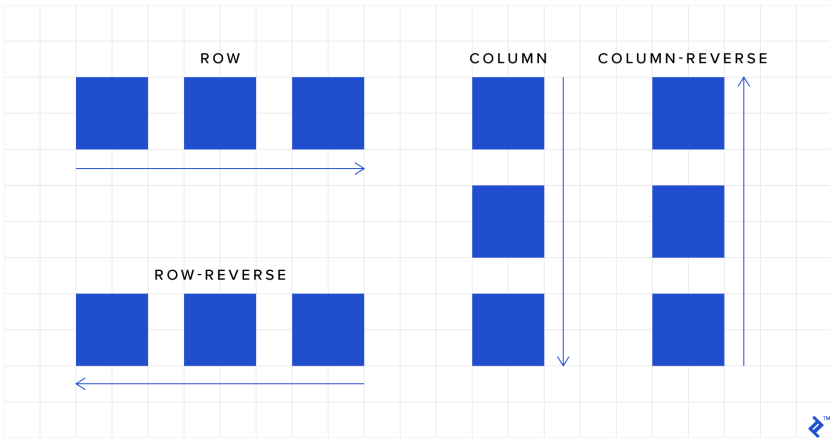
1.2.12 CSS Flex-Box

By default, HTML block-level elements stack, so if you want to align them in a row, you need to rely on CSS properties like float, or manipulate the display property with table-ish or inline-block settings. display is one of the most basic properties in CSS and is quite important in the context of Flexbox, as it is used to define a flex wrapper.

There are two possible flex wrapper values: flex and inline-flex.

The difference between the two is that a display: flex wrapper acts like a block element while a display: inline-flex wrapper acts like an inline-block. Also, inline-flex elements grow if there is not enough space to contain the children. But other than those differences, the behavior of the two would be the same.

```
.container{
    display:flex | inline-flex;
    flex-direction: row || row-reverse || column || column-reverse;
}
```



Flex-wrap The flex-wrap property defines whether the flex items are forced in a single line or can be flowed into multiple lines. If set to multiple lines, it also defines the cross-axis which determines the direction new lines are stacked in.

```
.container{
    flex-wrap: nowrap || wrap || wrap-reverse;
}
```

- `nowrap (default):` single-line which may cause the container to overflow
- `wrap:` multi-lines, direction is defined by flex-direction
- `wrap-reverse:` multi-lines, opposite to direction defined by flex-direction

Flex-flow You can combine flex-direction and flex-wrap properties into a single property: flex-flow.

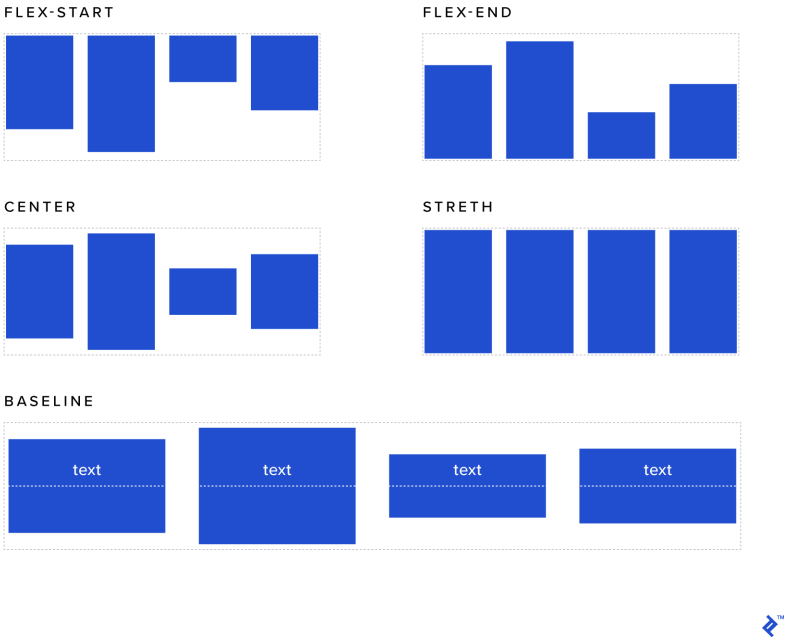
```
.container{
    /*flex-flow: {flex-direction} {flex-wrap};*/
    flex-flow: flex-direction flex-wrap|initial|inherit;
}
```

Justify content This property is used to control the horizontal alignment of the child elements

- `flex-start (default):` Elements are aligned to the left (similar to inline elements with text-align: left).
- `flex-end:` Elements are aligned to the right (similar to inline elements with text-align: right).
- `center:` Elements are centered (similar to inline elements with text-align: center).
- `space-around (where the magic begins):` Every element will be rendered with the same amount of space around each item. Note that the space between two sequential child elements will be double the space between the outermost elements and the sides of the wrapper.
- `space-between:` Just like space-around, except the elements will be separated by the same distance and there will be no space near either edge of the wrapper.

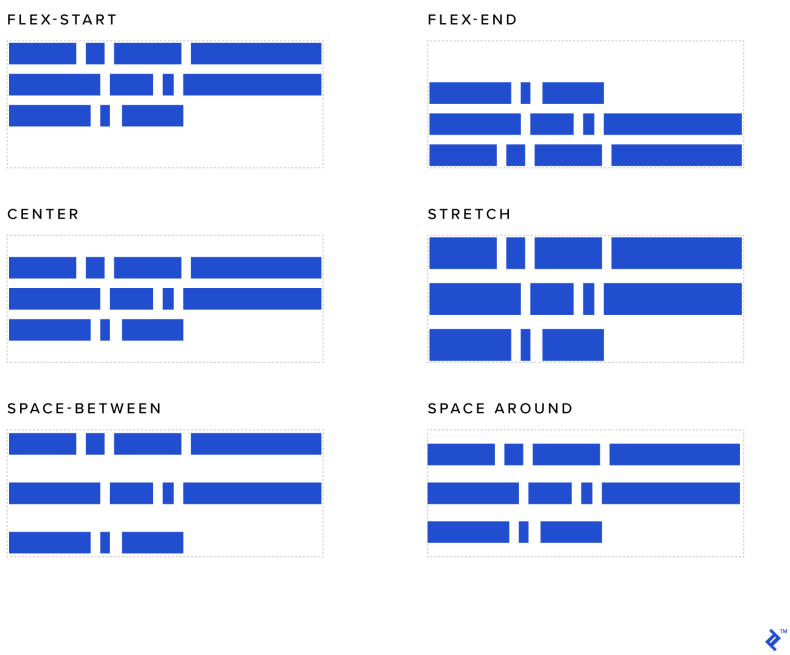
Align items This property is similar to justify-content but the context of its effects is the rows instead of the wrapper itself

- `flex-start:` Elements are vertically aligned to the top of the wrapper.
- `flex-end:` Elements are vertically aligned to the bottom of the wrapper.
- `center:` Elements are centered vertically within the wrapper (at last, a safe practice to achieve this).
- `stretch (default):` Forces the elements to occupy the full height (when applied to a row) and the full width (when applied to a column) of the wrapper.
- `baseline:` Vertically aligns the elements to their actual baselines



Align content This property is similar to justify-content and align-items but it works in the vertical axis and the context is the entire wrapper. To see its effects, you will need more than one row

- `flex-start:` Rows are vertically aligned to the top.
- `flex-end:` Rows are vertically aligned to the bottom.
- `center:` In general, this property stretches the elements to utilize the entire vertical height of the wrapper. However, if you have set some specific height of the elements, that height will be honored and the remaining vertical space (in that row, below that element) will be empty.
- `space-around:` Every row will be rendered with the same amount of space around itself vertically (i.e., below and above itself). Note that the space between two rows will, therefore, be double the space between the top and bottom rows and the edges of the wrapper.
- `space-between:` Just like space-around, except the elements will be separated by the same distance and there will be no space on the top and bottom edges of the wrapper.



Flex grow This property sets the relative proportion of the available space that the element should be using. The value should be an integer, where 0 is the default. Let's say you have two different elements into the same flex wrapper. If both have a flex-grow value of 1, they will grow equally to share the available space. But if one a flex-grow value of 1 and the other a flex-grow value of 2, as shown in the example below, this one with a flex-grow value of 2 will grow to take twice as much space as the first.

Flex shrink Similar to flex-grow, this property sets whether the element is "shrinkable" or not with an integer value. Similar to flex-grow, flex-shrink specifies the shrink factor of a flex item.

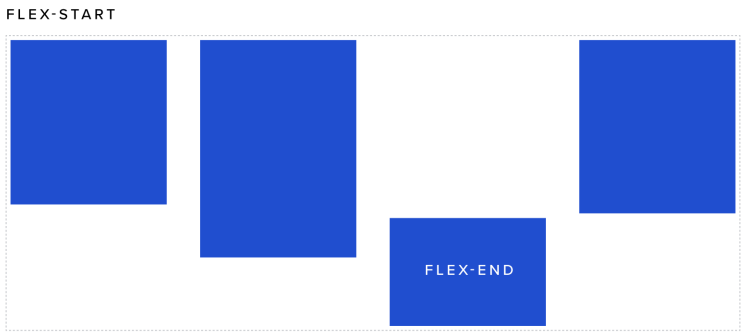
Flex basis This property is used to define the initial size of an element before available space is distributed and elements are adjusted accordingly.

Flex You can combine flex-grow, flex-shrink, and flex-basis properties into a single property: flex as follows:

```
.container{
    /*flex: {flex-grow} {flex-shrink} {flex-basis};*/
    flex: 0 1 auto;
}
```

Align self This property is similar to align-items but the effect is applied individually to each element. The possible values are:

- `flex-start:` Vertically aligns the element to the top of the wrapper.
- `flex-end:` Vertically aligns the element to the bottom of the wrapper.
- `center:` Vertically centers the element within the wrapper (at last a simple way to achieve this!).
- `stretch:` Stretches the element to occupy the full height of the wrapper (when applied to a row) or the full width of the wrapper (when applied to a column).
- `baseline:` Aligns the elements according to their actual baselines.



1.2.13 CSS Grid-Box

Grid layouts are fundamental to the design of websites, and the CSS Grid module is the most powerful and easiest tool for creating it. Grid layout works on a grid system. The grid is an intersecting set of horizontal and vertical lines that create a sizing and positioning coordinate system for the grid container's contents. To create a grid, you simply set an element to display: grid. This automatically makes all of that element's direct descendants grid items. You can now use the various grid properties to adjust their size and positioning as required.

```
<!DOCTYPE html>
<title>Example</title>
<style>
#grid {
  display: grid;
  grid-template-rows: 1fr 1fr 1fr;
  grid-template-columns: 1fr 1fr 1fr;
  grid-gap: 2vw;
  }
#grid > div {
  font-size: 5vw;
  padding: .5em;
  background: gold;
  text-align: center;
}
</style>
<div id="grid">
  <div>1</div>
  <div>2</div>
  <div>3</div>
  <div>4</div>
  <div>5</div>
  <div>6</div>
  <div>7</div>
  <div>8</div>
  <div>9</div>
</div>
```

- `display: grid` Turns the element into a grid container. This is all that's required in order to create a grid. We now have a grid container and grid items. The grid value generates a block-level grid container box.

- `grid-template-rows: 1fr 1fr 1fr` Explicitly sets the rows of the grid. Each value represents the size of the row.
- `grid-template-columns: 1fr 1fr 1fr` VSame as above except it defines the columns of the grid.
- `grid-gap: 2vw` Sets the gutter.
- `grid-template-rows: repeat(5, 1fr);` Same as grid-template-rows: 1fr 1fr 1fr 1fr 1fr;
- `grid-row: x / y` specifies a grid item's size and location within the grid row by contributing a line, a span, or nothing (automatic) to its grid placement, thereby specifying the inline-start and inline-end edge of its grid area.
- `grid-column: x / y` specifies a grid item's size and location within a grid column by contributing a line, a span, or nothing (automatic) to its grid placement, thereby specifying the inline-start and inline-end edge of its grid area.

1.2.14 Responsive Web Design

Responsive web design makes your web page look good on all devices. A website should display equally well in everything from widescreen monitors to mobile phones. It's an approach to web design and development that eliminates the distinction between the mobile-friendly version of your website and its desktop counterpart. Responsive design is accomplished through CSS "media queries". Think of media queries as a way to conditionally apply CSS rules. They tell the browser that it should ignore or apply specific rules depending on the user's device.

Media query Media queries let us present the same HTML content as distinct CSS layouts. So, instead of maintaining one website for smartphones and an entirely unrelated site for laptops/desktops, we can use the same HTML markup (and web server) for both of them.

```
/* Mobile Styles */
  @media only screen and (max-width: 400px) {
    body {
      background-color: #F09A9D; /* Red */
    }
  }

  /* Tablet Styles */
  @media only screen and (min-width: 401px) and (max-width: 960px) {
    body {
      background-color: #F5CF8E; /* Yellow */
    }
  }

  /* Desktop Styles */
  @media only screen and (min-width: 961px) {
    body {
      background-color: #B2D6FF; /* Blue */
    }
```

When you resize your browser, you should see three different background colors: blue when it's greater than 960px wide, yellow when it's between 401px and 960px, and red when it's less than 400px.

Relative CSS units At the core of responsive web design are relative CSS units. These are units that get their value from some other external value.

1.2.15 Advance selectors

Adjacent Sibling Selector It selects all the elements that are adjacent siblings of specified elements. It selects the second element if it immediately follows the first element.

```
            h4+ul{
        border: 4px solid red;
}
```

Attribute Selector It selects a particular type of inputs.

```
input[type="checkbox"]{
        background:orange;
}
a[href="http://www.google.com"]{
background:yellow;
}
```

nth-of-type Selector It selects an element from its position and types.

```
    div:nth-of-type(5){
        background:purple;
}
```

Direct Child Selector It selects any element matching the second element that is a direct child of an element matching the first element. The element matched by the second selector must be the immediate children of the elements matched by the first selector.

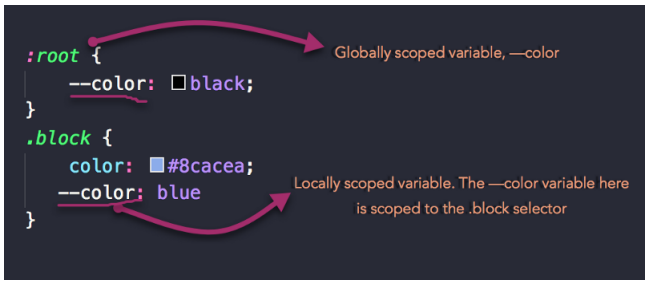
```
    p > div {
        background-color: DodgerBlue;
}
```

1.2.16 CSS variables

Variables are one of the major reasons CSS preprocessors exist at all. The ability to set a variable for something like a color, use that variable throughout the CSS you write, and know that it will be consistent, DRY, and easy to change is useful. :root –main-color: red

```
    :root {  --main-color: red}
    #div1 {
      background-color: var(--main-color);
    }
}
```

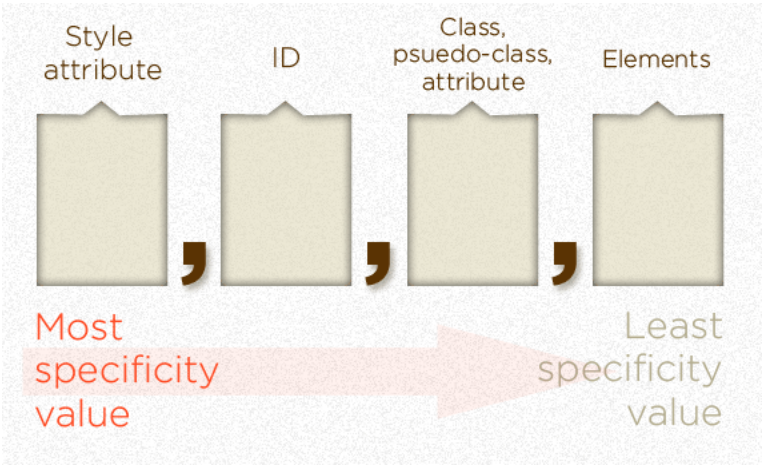
The :root selector allows you to target the highest-level element in the DOM, or document tree. So, variables declared in this way are kind of scoped to the global scope.



1.2.17 CSS Specificity

What is Specificity? If there are two or more conflicting CSS rules that point to the same element, the browser follows some rules to determine which one is most specific and therefore wins out. Think of specificity as a score/rank that determines which style declarations are ultimately applied to an element. Every selector has its place in the specificity hierarchy. There are four categories which define the specificity level of a selector:
Inline styles - An inline style is attached directly to the element to be styled. Example: `<h1 style="color: #ffffff;">`.
IDs - An ID is a unique identifier for the page elements, such as `#navbar`.
Classes, attributes and pseudo-classes - This category includes .classes, [attributes] and pseudo-classes such as :hover, :focus etc.

Elements and pseudo-elements - This category includes element names and pseudo-elements, such as h1, div, :before and :after.



In other words:
If the element has inline styling, that automatically1 wins (1,0,0,0 points) For each ID value, apply 0,1,0,0 points For each class value (or pseudo-class or attribute selector), apply 0,0,1,0 points For each element reference, apply 0,0,0,1 point You can generally read the values as if they were just a number, like 1,0,0,0 is "1000", and so clearly wins over a specificity of 0,1,0,0 or "100". The commas are there to remind us that this isn't really a "base 10" system, in that you could technically have a specificity value of like 0,1,13,4 – and that "13" doesn't spill over like a base 10 system would.

2 Javascript

2.1 Data types

There are eight basic data types in JavaScript. Here, we'll cover them in general and in the next chapters we'll talk about each of them in detail.

Number The number type represents both integer and floating point numbers.

```
let n = 123;
n = 12.345;
```

Besides regular numbers, there are so-called "special numeric values" which also belong to this data type: Infinity, -Infinity and NaN.

String A string in JavaScript must be surrounded by quotes.

```
     let str = "Hello";
let str2 = 'Single quotes are ok too';
let phrase = `can embed another \${str}`;
```

Backticks are "extended functionality" quotes. They allow us to embed variables and expressions into a string by wrapping them in $...

Boolean The boolean type has only two values: true and false.

```
let nameFieldChecked = true; // yes, name field is checked
  let ageFieldChecked = false; // no, age field is not checked
```

Null The special null value does not belong to any of the types described above. It forms a separate type of its own which contains only the null value:

```
let age = null;
```

In JavaScript, null is not a "reference to a non-existing object" or a "null pointer" like in some other languages. It's just a special value which represents "nothing", "empty" or "value unknown".

Undefined The special value undefined also stands apart. It makes a type of its own, just like null.

```
let age; // shows "undefined"
```

Null VS Undefined Normally, one uses null to assign an "empty" or "unknown" value to a variable, while undefined is reserved as a default initial value for unassigned things.

JavaScript Objects In JavaScript, an object is a collection of properties, defined as a key-value pair. Each property has a key and a value. The property key can be a string and the property value can be any valid value.

To create an object, you use the object literal syntax. For example, the following snippet creates an empty object:

```
let empty = {};
```

To create an object with properties, you use the key : value syntax. For example, the following snippet creates a person object:

```
let person = {
     firstName: 'John',
     lastName: 'Doe'
 };
```

To access a property of an object, you use one of two notations: the dot notation and array-like notation.

1) The dot notation (.) The following illustrates how to use the dot notation to access a property of an object: For example, to access the firstName property of the person object, you use the following expression:

```
person.firstName
```

2) Array-like notation ([]) The following illustrates how to access the value of an object's property via the array-like notation:

```
    let person = {
    firstName: 'John',
    lastName: 'Doe'
};

console.log(person['firstName']);
console.log(person['lastName']);
```

Add a new property to an object

Unlike objects in other programming languages such as Java and C#, you can add a property to an object after creating it.

The following statement adds the age property to the person object and assigns 25 to it:

```
person.age = 25;
```

Check if a property exists

To check if a property exists in an object, you use the in operator:

```
    let employee = {
    firstName: 'Peter',
    lastName: 'Doe',
    employeeId: 1
};

console.log('ssn' in employee); //false
console.log('employeeId' in employee);//true
```

Iterate over properties of an object using for...in loop

```
let website = {
title: 'JavaScript Tutorial',
url: 'https://www.javascripttutorial.net',
tags: ['es6', 'javascript', 'node.js']
};

for (const key in website) {
    console.log(website[key]);
}
```

console output:

```
// "JavaScript Tutorial"
 //"https://www.javascripttutorial.net"
 //["es6", "javascript", "node.js"]
```

It's also possible to call hasOwnProperty through the object itself:

```
        if (obj.hasOwnProperty(prop)) {
}
```

But this will fail if the object has an unrelated field with the same name:

```
var obj = { foo: 42, hasOwnProperty: 'lol' };
obj.hasOwnProperty('foo');  // TypeError: hasOwnProperty is not a function
```

That's why it's safer to call it through Object.prototype instead:

```
var obj = { foo: 42, hasOwnProperty: 'lol' };
Object.prototype.hasOwnProperty.call(obj, 'foo');  // true
```

Iterate over properties of an object using forEach & Object.keys

```
Object.keys(website).forEach(e => console.log(`key=${e}  value=${website[e]}`));
```

console output:

```
//"key=title  value=JavaScript Tutorial"
//"key=url  value=https://www.javascripttutorial.net"
//"key=tags  value=es6,javascript,node.js"
```

Iterate over properties of an object using map & Object.entries

```
 const anObj = { 100: 'a', 2: 'b', 7: 'c' };
 console.log(Object.entries(anObj)); // [ ['2', 'b'],['7', 'c'],['100', 'a'] ]

 Object.entries(anObj).map(obj => {
   const key   = obj[0];
   const value = obj[1];

   // do whatever you want with those values.
});
 Object.entries(obj).forEach(([key, value]) => {
  console.log(`${key} ${value}`); // "a 5", "b 7", "c 9"
});
```

Map Prior to ES6, when you need to map keys to values, you often use an object, because an object allows you to map a key to a value of any type. By definition, a Map object holds key-value pairs where values of any type can be used as either keys or values. In addition, a Map object remembers the original insertion order of the keys.

```
let map = new Map([iterable]);
```

The Map() accepts an optional iterable object whose elements are key-value pairs.

- `clear()` – removes all elements from the map object.
- `delete(key)` – removes an element specified by the key. It returns if the element is in the map, or false if it does not.
- `entries()` – returns a new Iterator object that contains an array of [key, value] for each element in the map object. The order of objects in the map is the same as the insertion order.
- `forEach(callback[, thisArg])`– invokes a callback for each key-value pair in the map in the insertion order. The optional thisArg parameter sets the this value for each callback.
- `get(key)`– returns the value associated with the key. If the key does not exist, it returns undefined.
- `has(key)` – returns true if a value associated with the key exists, otherwise, return false.
- `keys()` – returns a new Iterator that contains the keys for elements in insertion order.
- `set(key, value)`– sets the value for the key in the map object. It returns the map object itself therefore you can chain this method with other methods.
- `values()` - returns a new iterator object that contains values for each element in insertion order.

```
    let userRoles = new Map();
    console.log(typeof(userRoles)); // object
    console.log(userRoles instanceof Map); // true
    let john = {name: 'John Doe'},
    lily = {name: 'Lily Bush'},
    peter = {name: 'Peter Drucker'};
    let userRoles = new Map([
    [john, 'admin'],
    [lily, 'editor'],
    [peter, 'subscriber']
]);
for (let role of userRoles.values()) {
    console.log(role);
}
// admin
// editor
// subscriber
//Add elements to a Map
for (let elem of userRoles.entries()) {
    console.log(`${elem[0].name}: ${elem[1]}`);
}

// John Doe: admin
// Lily Bush: editor
// Peter Drucker: subscriber
userRoles.set('100','200')
//Get an element from a map by key
console.log(userRoles.get('100'));//200
console.log(userRoles.get('john')); //admin

var roles = [...userRoles.values()];
console.log(roles); //[ 'admin', 'editor', 'subscriber' ]
```

Array Arrays provide a lot of methods. To make things easier, in this chapter they are split into groups. Add/remove items We already know methods that add and remove items from the beginning or the end:

- `arr.push(...items)` – adds items to the end,
- `arr.pop()` – removes an element specified by the key. It returns if the element is in the map, or false if it does not.

- `arr.shift()` – extracts an item from the beginning,
- `arr.unshift(...items)` – adds items to the beginning.

```
 //arr.splice(index[, deleteCount, elem1, ..., elemN])
 let arr = ["I", "study", "JavaScript"];
  arr.splice(1, 1); // from index 1 remove 1 element
  alert( arr ); // ["I", "JavaScript"]
  //arr.slice([start], [end])
  let john = ["Roee" , "Angel","The","king"];
  let arr = john.slice(0,2)
  console.log(arr) //["Roee", "Angel"]

//  let result = arr.find(function(item, index, array) {
// if true is returned, item is returned and iteration is stopped
//item is the element. index is its index. array is the array itself.
// for falsy scenario returns undefined });
let users = [
    {id: 1, name: "John"},
    {id: 2, name: "Pete"},
    {id: 3, name: "Mary"}
  ];

  let user = users.find(item => item.id == 1);

  alert(user.name); // John

  //filter
  let users = [
    {id: 1, name: "John"},
    {id: 2, name: "Pete"},
    {id: 3, name: "Mary"}
  ];

  // returns array of the first two users
  let someUsers = users.filter(item => item.id < 3);

  alert(someUsers.length); // 2

  //map
  let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
  alert(lengths); // 5,7,6
```

2.2 Functions

There is a lot of build-in function inside Object, Windows, Array, etc. A function is a parametric block of code defined once and called multiple times later. In JavaScript, a function is composed and influenced by many components:

- JavaScript code that forms the function body
- The list of parameters
- The variables accessible from the lexical scope
- The returned value
- The context this when the function is invoked
- Named or an anonymous function
- The variable that holds the function object
- Arguments object (or missing in an arrow function)

Function declaration

A function is a parametric block of code defined once and called multiple times later. In JavaScript a function is composed and influenced by many components:

```
function name(parameter1, parameter2, parameter3)
{
    // what the function does3
}
```

As you can see, it consists of the function keyword plus a name. The function's parameters are in the brackets and you have curly brackets around what the function performs. You can create your own, but to make your life easier – there are also a number of default functions.

```
        // function declaration
function isEven(num) {
  return num % 2 === 0;
}
isEven(24); // => true
isEven(11); // => false
```

function isEven(num) {...} is a function declaration that defines isEven function, which determines if a number is even.

The function declaration creates a variable in the current scope with the identifier equal to the function name. This variable holds the function object.

The function variable is hoisted up to the top of the current scope, which means that the function can be invoked before the declaration (see this chapter for more details).

The created function is named, which means that the name property of the function object holds its name. It is useful when viewing the call stack: in debugging or error messages reading.

As you can see, it consists of the function keyword plus a name. The function's parameters are in the brackets, and you have curly brackets around what the function performs. You can create your own, but to make your life easier – there are also several default functions.

Function expression

A JavaScript function can also be defined using an expression.

A function expression can be stored in a variable:

```
const doStuff = function() {}
```

We often see anonymous functions used with ES6 syntax like so:

```
const doStuff = () => {}
```

IIFE The name — immediately invoked function expressions — pretty much says it all here. When a function is created at the same time it is called, you can use an IIFE, which looks like this:

```
(function() => {})()
//or
(() => {})()
```

Callbacks A function passed to another function is often referred to as a "callback" in JavaScript.