CSS: the language of web design

At times we use ☝ and 👇 to make it clear whether an explanation belongs to the code snippet above or below the text. The ‼ sign is added to code examples you should run yourself. When you see a 🐛, we offer advice on how to debug your code with the browser's and VSC's tooling - these hints are solely to help you with your programming project and not exam material! Lastly, paragraphs with a 🚩 are just for your information and not exam material.

An automatically generated PDF of this transcript is available here.

Table of Contents

- Required & recommended readings and activities
- Learning goals
- Code examples
- A word of warning
- Context
- Take-aways of the required reading
- Pseudo-classes
  - ‼ nth-child(X) and nth-of-type(X)
  - ‼ root
  - ‼ hover and active
  - ‼ enabled and disabled
  - ‼ not
  - ‼ Selector combinations
- Pseudo-elements
  - ‼ first-letter and first-line
  - ‼ before and after
- ‼ Data attributes
- Element positioning
  - ‼ Float
  - ‼ Position
- CSS media queries
- Animations and transitions
  - CSS vs. JavaScript animations
  - Animations
    - ‼ Animations: TU DELFT neon sign
    - ‼ Animations: Delft weather app
  - ‼ Transitions
- Browser-specific prefixes
- Self-check

Required & recommended readings and activities

- Required readings: //TODO: update the required readings for CSS
- Recommended activities:
  - 📺 Watch this talk by Rachel Andrew to learn more about the long (and painful) history of CSS.
  - 📺 Watch this talk by Mandy Michael on text effects in CSS. Data attributes and pseudo-elements are a powerful combination!
  - 🎶 Listen to this podcast by Wes Bos and Scott Tolinski on the present and future of CSS.
  - If you are looking for styling challenges, take a look at the weekly CodePen Challenges.
  - At https://flukeout.github.io/ you find a small game that allows you to practice CSS selectors.
- Recommended readings:
  - 📕 Chapters 1-4 and chapter 13 of The Book of CSS3: A developer's guide to the future of web design by Peter Gasston (2nd edition, 2014).
  - To make sense of CSS positioning, take a look here.
  - If you want to look behind the scenes, read Mozilla's blog post on their CSS engine Quantum CSS.
  - After studying this lecture you have the basic knowledge to consider a career as a Front-End Developer
- Relevant scientific publications:
  - Meyerovich, L.A. and Bodik, R., 2010. Fast and parallel webpage layout In Proceedings of the 19th international conference on World wide web, pp. 711-720.
  - Panchekha P. and Torlak E., 2016. Automated reasoning for web page layout. In Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 181-194.

Learning goals

- Position and style HTML elements according to a given design of a web application.
- Employ pseudo-classes and pseudo-elements.
- Employ CSS variables and data access/creation facilities.
- Write CSS media queries.
- Create CSS-based animations.

Code examples

At https://codepen.io/charlottehase/full/abzYoQp you find a demo that we developed to show off some of CSS' animation capabilities. Throughout this transcript we will refer to it from time to time---we encourage you to fork it and explore the code. It contains a lot of comments that should help you get started.

This demo contains only a single image (the TU DELFT EWI building), everything else is designed with CSS. Since some elements appear hundreds of times (e.g., every rain drop is a separate `<span>`, every cloud is a `<div>`) the demo also shows off how to set up an element's style in JavaScript.

More generally, if you want to find inspirations about what is possible with a bit of HTML, CSS (and JavaScript), it is worthwhile to take a look at CodePen and what it has to offer.

As a second example application we often refer to a todo application. Here, we do not offer an entire application but present how specific parts of the application can be styled.

Lastly, we also consider a TU DELFT neon sign as an example for text-heavy animations. The entire source code is presented here.

A word of warning

This tweet 👇 sums up a lot of CSS experiences - some things are easy with CSS (animations come to mind), while others, which intuitively should not be that hard to accomplish (e.g. the vertical alignment of elements before CSS grid came along), will cost you a lot of time.

12:45 PM · Nov 30, 2017

7.3K 2.7K people are Tweeting about this

Another important bit of information comes from Addy Osmani:

Addy Osmani
@addyosmani

border: 1px solid red; is the console.log of CSS

10:00 AM · Jun 24, 2020

10.8K 2.1K people are Tweeting about this

Context

The CSS - or **Cascading Style Sheets** - language describes how elements in the document object model (DOM) should be rendered. As other web technologies, CSS can be considered a victim of the browser wars - which led to years of inactivity in the standard's refinement:

- **CSS1** became a W3C recommendation in 1996. It had support for fonts, colors, alignment, margins, ids and classes.
- Two years later, **CSS2** became a W3C recommendation, adding support for media queries and element positioning (among others). The W3C was ready to get started on CSS3. However, the browser support for CSS2 remained inconsistent; the W3C decided to focus on fixing inconsistencies with CSS2.1.
- In 2011, **CSS2.1** became a W3C recommendation which fixed errors and added support for features already widely implemented in major browsers.
- Work on **CSS3** began already in 1998, was put on hold, and is now back in full swing with browser vendors pushing to implement the newest standards. CSS became so complex and agreeing on a standard for the entire language became so tedious that after CSS2 the monolithic nature of the standard was given up. Instead, we now have **CSS modules**, which each progress at an individual pace. The current state of those modules is available at https://www.w3.org/Style/CSS/current-work. Many of those modules are in *Working Draft* status, while some have already reached level 4 - the *level number* indicates how many revisions of the module have taken place.

Any CSS module developed after CSS2.1 can be considered as CSS3. There will not be a monolithic CSS4, instead the different modules of the CSS specification will each continue to develop at their own pace.

Be aware, that it is not possible to make use of the very latest CSS features and to expect them to work across all major browsers. Instead, which features to use should be based on:

- the intended user base (will tell us something about the most popular browsers in use);
- the mode of usage (smartphone vs. touch screen vs ...);
- the type of web application (are 3D animations necessary, etc.).

When building web applications that should work across a range of browsers (old and new), you will often come across the terms **shims**, **fallbacks** and **polyfills**. These are all terms used to provide HTML5 (which includes CSS3) functionalities in browsers that do not natively support them.

Take-aways of the required reading

Having worked through the required reading, you should be able to style HTML elements with CSS.

You have also learned about three types of style sheets:

- the browser's style sheet;
- the author's style sheet;
- the user's style sheet.

The user's style sheet overrides the other two and the author's style sheet overrides the browser's.

Style sheets are processed in order: later declarations override earlier ones if they are on the same or a higher specificity level.

You know that in the following code snippet 👇:

```
body {
  background-color: #ffff00;
}
```

`body` is a **selector**, `background-color` is a **property** and `#ffff00` is a **value**. You know the difference between a `class` and an `id` attribute and how to use both.

Note: In this example ☝ the color value is represented by its hex value. Throughout this transcript, we often use color names - there are more than 140 color names that all modern browsers recognize (a nicely formatted list on is available on Wikipedia). Besides hex values, colors can also be represented by their `rgba` values (red, green, blue, alpha). We suggest convertingcolors.com as one option to convert between different color formats.

Lastly, you know about CSS resets and `!important` which overrides all other declarations.

In this lecture, we move beyond these basics and introduce a number of more advanced CSS concepts.

!! We strongly suggest to work with the toy examples presented here and not only render them yourself in the browser, but also make changes to the CSS to see the impact of those changes. Find out what your browser's Style Editor and Inspector can do!

Pseudo-classes

A **pseudo-class** is a keyword added to a **selector** that indicates *a particular state or type* of the corresponding element. Pseudo-classes allow styling according to (among others) **document external** factors such as mouse movements and user browsing history as well as **element external** factors such as the placement of the element within the entire document structure.

They are used as follows 👇:

```
selector:pseudo-class {
    property: value;
    property: value;
}
```

Imagine you have a list of twenty todo items and you want to alternate the todo items' background color (to make it easier to read them). Two simple ways to go about this are:

- You can *hardcode* the CSS rule of every element, ending up with twenty rules. This is not maintainable.
- You write two CSS rules (one per background color), assign each to a class and then alternate in the todo items class assignment. Better than the first option, but the redundancy remains. And what happens if you want to include a todo in the middle of the existing list? All subsequent todos would have to be assigned a different class to maintain the alternating colors.

Ideally, we only create two CSS rules and solve the rest (alternate assignment of background colors) with pseudo-classes. And that's exactly what we can do with the two pseudo-classes we introduce next.

!! nth-child(X) and nth-of-type(X)

- `nth-child(X)` is any element that is the Xth **child element** of its parent;
- `nth-of-type(X)` is any element that is the Xth **sibling** of its type.

In both cases, `X` can be an integer or formula, e.g `2n+1`, where `n` represents a number starting at 0 and incrementing. Instead of `2n` and `2n+1` we can also use `even` and `odd` as shortcuts. If we want to start counting elements in reverse order, we can use the analogous `nth-last-child(X)` and `nth-last-of-type(X)` pseudo-classes.

If we are aiming at the first and/or last child or sibling element, we also have additional shortcut pseudo-classes:

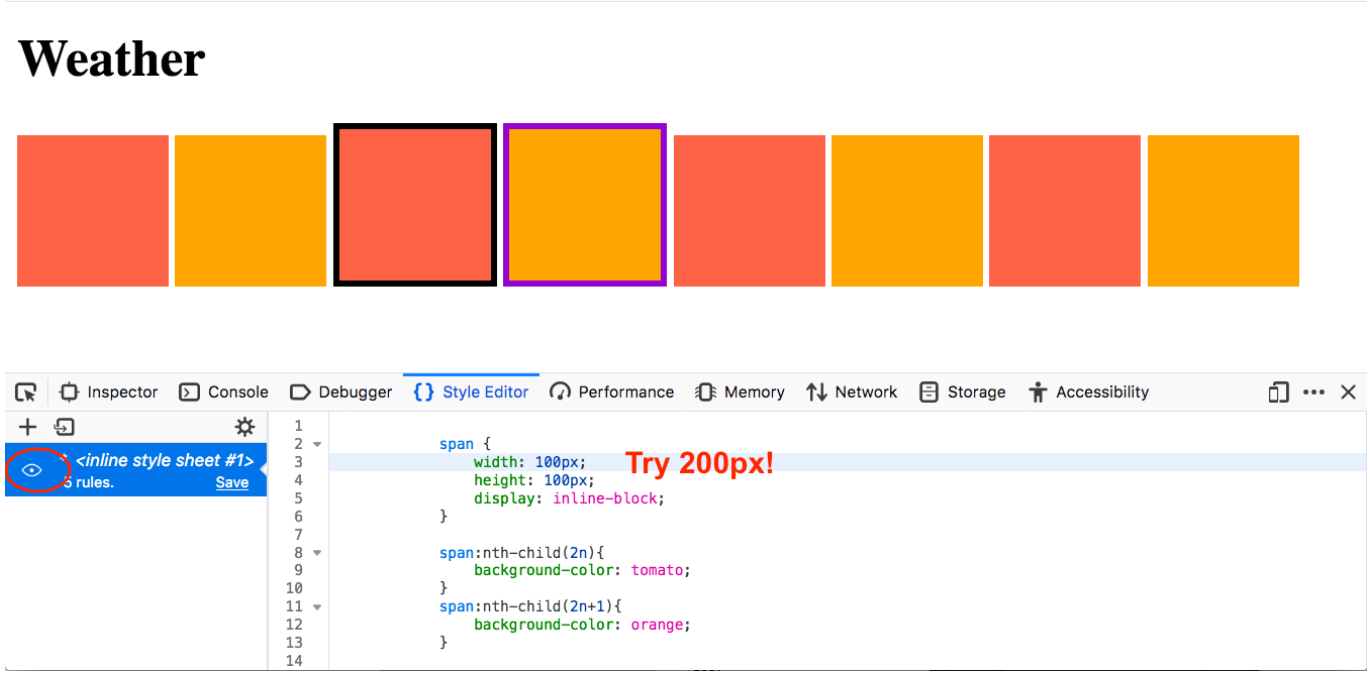
Pseudo-class	Equivalent to
`:first-child`	`:nth-child(1)`

Pseudo-class	Equivalent to
`:last-child`	`:nth-last-child(1)`
`:first-of-type`	`:nth-of-type(1)`
`:last-of-type`	`:nth-last-of-type(1)`

In the following example both pseudo-classes (`nth-child` and `nth-of-type`) are showcased 👇:

```
<!DOCTYPE html>
<html>
    <head>
        <style>
            span {
                width: 100px;
                height: 100px;
                display: inline-block;
            }

            span:nth-child(2n){
                background-color: tomato;
            }
            span:nth-child(2n+1){
                background-color: orange;
            }

            span:nth-child(4){
                border: 4px solid black;
            }

            span:nth-of-type(4){
                border: 4px solid darkviolet;
            }
        </style>
    </head>
    <body>
        <h1>Weather</h1>
        <span id="c1"></span>
        <span id="c2"></span>
        <span id="c3"></span>
        <span id="c4"></span>
        <span id="c5"></span>
        <span id="c6"></span>
        <span id="c7"></span>
        <span id="c8"></span>
        </ul>
    </body>
</html>
```

If we open Firefox's Style Editor we now see the following:



This piece of code ☝️ is rendered with alternating background colors as was our goal. We also see here clearly how `nth-child` and `nth-of-type` differ:

- the third `<span>` (`id=c3`) element is the fourth child of its parent `<body>` and thus is rendered with a black border;
- the fourth `<span>` (`id=c4`) element is the fourth sibling among the group of `<span>` elements and thus rendered with a violet border.

If we were to remove the `<h1>` element in our code snippet, `nth-child(4)` and `nth-of-type(4)` would point to exactly the same `<span>` element.

Firefox's Style Editor allows us to not only inspect the CSS, but to also:

- switch off the CSS (with a click on the "eye" symbol on the left);
- **change the CSS** in the editor and view the immediate effect of those changes.

🪛 The last point is vital for efficient CSS styling: instead of coding, saving and opening your HTML file in the browser, you can prototype your CSS rules directly in the browser. Try it out for yourself! Just do not forget to then copy the new rules to your original CSS file as those changes are lost when the Style Editor is closed. In addition, Visual Studio Code has also good code commentary for CSS: as an example, if you add the line `float: left` right after `display: inline-block` in the current code example, VSC will warn you that this CSS rule has no effect and will provide the reason why.

Lastly, you can use this pseudo-class to change the color of half the clouds in our Delft weather app: simply append to the end of the CSS the following code snippet (you will learn what `::after` and `::before` refers to in a bit) to make half the clouds red:

```
.cloud:nth-child(2n){
  background-color: red;
}
.cloud:nth-child(2n)::after, .cloud:nth-child(2n)::before {
```

```
    background-color: red;
}
```

☝🏽 Here, `nth-of-type` could have also been used instead.

‼️ root

One of the often voiced complaints about CSS used to be the lack of variable support (and thus languages that compile into CSS were born, e.g. Sass). In our TU Delft weather app we are playing around with quite a few colors and, importantly, as you can see in the red-clouds example above, we may have to alter several CSS rules to achieve seemingly a single effect. If we now decide to change that color value again, we have to determine all rules that need to be altered (again). Clearly, variables (i.e. *entities containing specific values that can be reused*) would be very helpful.

In fact, since 2015/2016 (*yes, it took that long*), CSS variables are supported in all major browsers. Here is how they work with our toy example introduced earlier 👇🏽:

```
<!DOCTYPE html>
<html>
    <head>
        <style>
            :root {
                --color1: tomato;
                --color2: orange;
                --highlight1: black;
                --highlight2: darkviolet;
            }
            span {
                width: 100px;
                height: 100px;
                display: inline-block;
            }

            span:nth-child(2n){
                background-color: var(--color1);
            }
            span:nth-child(2n+1){
                background-color: var(--color2);
            }

            span:nth-child(4){
                border: 4px solid var(--highlight1);
            }

            span:nth-of-type(4){
                border: 4px solid var(--highlight2);
            }
        </style>
    </head>
    <body>
```

```
        <h1>Weather</h1>
          <span id="c1"></span>
          <span id="c2"></span>
          <span id="c3"></span>
          <span id="c4"></span>
          <span id="c5"></span>
          <span id="c6"></span>
          <span id="c7"></span>
          <span id="c8"></span>
        </ul>
    </body>
</html>
```

☝🏽 In this example, we create a global CSS variable, i.e. one that is available to all elements in the DOM tree. For this reason, we make use of the pseudo-element `:root` which represents the `<html>` element. Variables are defined with the custum prefix `--` and can be accessed using the `var()` functionality. Non-global CSS variables can be added in the same manner to any element, though they are then only available within their`{....}` block.

In our Delft weather app the `:root` element looks as follows:

```
:root {
  --boardColorOne: steelblue; /* first board color */
  --boardColorTwo: deeppink; /* second board color */
  --hoverTextColor: mediumvioletred; /* text color when mouse hovering */
  --borderBoardColor: pink;
  --nailColor: darkblue;
  --boardRotationStart: 5deg;
  --boardRotationEnd: -5deg;
  --switchOnColor: dodgerblue;
  --switchOffColor: grey;
  --switchColor: white;
}
```

and the swinging letter boards are defined with the following CSS rules:

```
.letter {
  width: 130px;
  height: 150px;
  /* border-radius rounds the corners of an element; here
   * the top 2 corners have no radius (no rounding) while
   * the bottom 2 corners are rounded
   */
  border-radius: 0 0 20px 20px;
  /*linear-gradient creates an image based on 2+ colors; has many options
*/
  background: linear-gradient(to top, var(--boardColorTwo), var(--
boardColorOne));
  margin: 10px;
```

```
    float: left;
    font-size:100px;
    font-family: 'Kanit', sans-serif;
    border: 5px solid var(--borderBoardColor);
    opacity: 0.7; /*determines transparency: 0 (invisible), 1 (opaque) */
    /* creates shadow around element: x-offset, y-offset, blur amount, color
     */
    box-shadow: 5px 5px 10px var(--boardColorTwo);

    /* point around which transformation is applied, here we want
     * the nail (roughly middle of the element and at the top) to
     * be the origin, otherwise the swinging does not happen
     * "around" the nail */
    transform-origin: 50% 0;

    /* initially: same animation for all letter boards
     * => keyframes (describes the content of the animation)
     * => animation timing (param change over time)
     * => duration (of a single animation cycle)
     * => iteration-count (how often should each cycle be performed)
     * => direction (restart from the origin or ... )
     */
    animation: swinging ease-in-out 1s infinite alternate;
}
```

While many of these CSS properties won't yet make sense, we point out three properties that together provide easy ways to achieve a good look:

- `border-radius` as an important property to easily achieve rounded edges and circles (MDN documentation);
- `linear-gradient` is a function that creates an image that contains a color gradient (MDN documentation) which is then assigned to the `background` property;
- `box-shadow` as a CSS property that adds a shadow effect to its elements (MDN documentation).

The moon in our Delft weather app for instance relies heavily on these three properties.

!! hover and active

In the beginning we mentioned mouse movements as one of the document external factors that we can make our elements react to. Two popular pseudo-classes in this category are `:hover` and `:active`:

- `:hover` is a selector that becomes active when a mouseover on the element occurs.
- `:active` is a selector that becomes active when the element is currently *being active* (usually that means clicked).

Here is a simple example 👇:

```
<!DOCTYPE html>
<html>
    <head>
```

```
        <style>
            button {
                background: white;
                color: darkgray;
                width: 100px;
                padding: 5px;
                font-weight: bold;
                text-align: center;
                border: 1px solid darkgray;
            }

            button:hover {
                color: white;
                background: darkgray;
            }

            button:active {
                border: 1px dashed;
                border-color: black;
            }
        </style>
    </head>
    <body>
        <main>
            <button>Add Todo</button>
        </main>
    </body>
</html>
```

While this toy example is not very impressive, `:hover` can easily be employed to create image galleries (preview vs. full image) as well as pure CSS dropdown menus (hide the menu items apart from the header and only reveal them when the mouse hovers over the header).

‼️ enabled and disabled

Particularly important for board game applications such as those in our assignments may be the pseudo-classes `:enabled` and `:disabled`. Imagine a game item that is only available sometimes; in our demo game for instance, once a letter has been clicked it is no longer possible to select it again. And of course this information should be visually conveyed to the user:

- `:enabled` is an element that can be clicked or selected.
- `:disabled` is an element that cannot be clicked or selected.

An example 👇:

```
<!DOCTYPE html>
<html>
    <head>
        <script>
            function disable(el){
```

```
                document.getElementById(el.id).disabled = true;
            }
        </script>
        <style>
            button {
                background: white;
                color: darkgray;
                width: 100px;
                padding: 5px;
                font-weight: bold;
                text-align: center;
                border: 1px solid darkgray;
            }

            button:enabled:hover {
                color: white;
                background: darkgray;
            }

            button:enabled:active {
                border: 1px dashed;
                border-color: black;
            }

            button:disabled {
                background: #ddd;
                color: #aaa;
                border: 1px solid #bbb;
            }
        </style>
    </head>
    <body>
        <main>
            <button id="b" onclick="disable(this)">Add Todo</button>
        </main>
    </body>
</html>
```

☝🏽 Here you can see that it is possible to combine pseudo-classes, in this case we use `button:enabled:active` and define a style that an enabled **and** active button should have. Once the button is disabled, its style will not change anymore, no matter the mouse movements.

‼️ not

`:not(X)` is a pseudo-class that matches all elements that are not represented by selector `X`. Let's look at at 👇🏽:

```
<!DOCTYPE html>
<html>
  <head>
```

```
      <style>
        main :not(.todo){
          color: orange;
        }
      </style>
    </head>
    <body>
      <main>
        <h2>Todos</h2>
        <p class="firsttodo">Today's todos</p>
        <p class="todo">Tomorrow's todos</p>
        <p class="todo">Saturday's todos</p>
        <p>Sunday's todos</p>
      </main>
    </body>
  </html>
```

☝ Important to know here is, that the selector `e1 e2` selects **all `<e2>` elements inside `<e1>` elements**. The `:not` selector is a short form for `*:not` with `*` being the **universal selector** (selecting all elements). Altogether our CSS rule says that any element within `<main>` that does not have class attribute `.todo` is assigned an orange font color.

‼ Selector combinations

`e1 e2` is not the only possible element combination as selector. Most commonly used are:

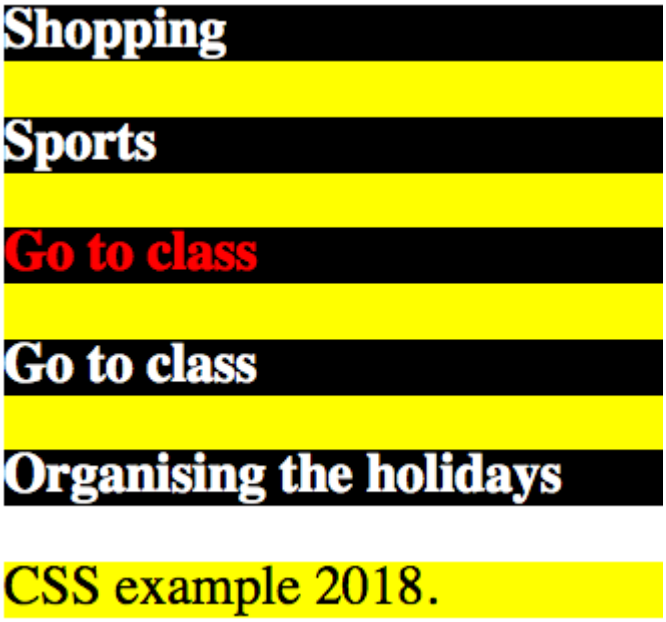
Selector	Description
e1	Selects all `<e1>` elements
e1 e2	Selects all `<e2>` elements within `<e1>`
e1,e2	Selects all `<e1>` elements and all `<e2>` elements
e1>e2	Selects all `<e2>` elements that have `<e1>` as parent
e1+e2	Selects all `<e2>` elements that follow `<e1>` immmediately

An example should make the differences between them clear 👇:

```
<!DOCTYPE html>
<html>
  <head>
    <style>

      /* select all p elements within div */
      div p {
        background-color: black;
        color: white;
      }
      /* select all main and all footer elements */
```

```
        main, footer {
          background-color: yellow;
        }
        /* select all p elements that have div as parent */
        div > p {
          font-weight: bold;
        }
        /* select all p elements that follow div immediately */
        div + p {
          color: red;
        }
      </style>
    </head>

    <body>
      <main>

        <div class="today">

          <div class="urgent">
            <p>Shopping</p>
            <p>Sports</p>
          </div>

          <p>Go to class</p>  <!-- follows directly after a <div></div> -->

        </div>

        <div class="tomorrow">

          <p>Go to class</p>

          <div class="urgent">
            <p>Organising the holidays</p>
          </div>

        </div>

      </main>

      <footer>
        <p>CSS example 2018.</p> <!-- a <p> element not inside a <div> -->
      </footer>
    </body>
  </html>
```

The rendering of the above code looks as follows:



🎮 If you are looking for a fun way to practice these selectors, head over to the CSS diner!

Pseudo-elements

A **pseudo-element** creates an abstractions about the document tree *beyond* those specified by the document language; it provides access to an element's **sub-parts** (e.g. the first letter of a paragraph).

In order to distinguish pseudo-classes and pseudo-elements, the `::` notation was introduced in the specification, though browsers also accept the one-colon notation.

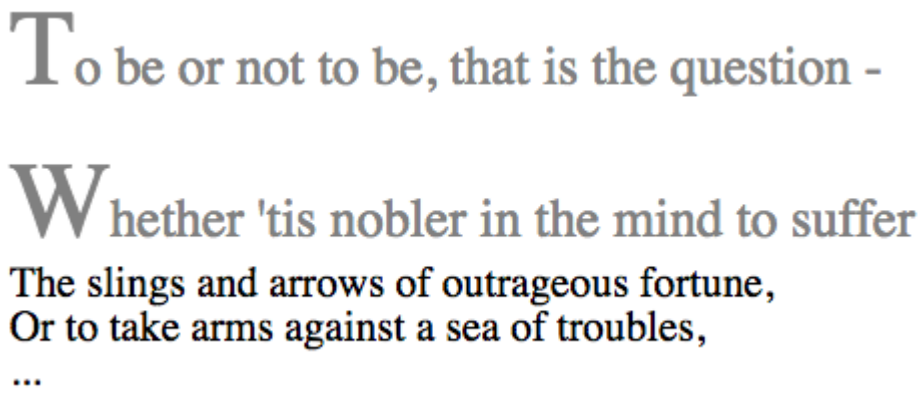
‼️ first-letter and first-line

So, what are abstractions that go beyond what is specified in the document language? Two popular examples are the `::first-letter` and the `::first-line` pseudo-elements; they do exactly what the names suggests, enabling you to style the first letter and line respectively. Without those pseudo-elements, you would have to wrap the first letter in a `<span>` element and find a clever way of computing what exactly the first line of a piece of text is (to again, wrap it in a `<span>` element). Let's look at an example 👇:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
    p::first-line {
      color: gray;
      font-size: 125%;
    }

    p::first-letter {
      font-size: 200%;
    }
    </style>
  </head>
```

```
    <body>
      <p>
        To be or not to be, that is the question -
      </p>

      <p>
        Whether 'tis nobler in the mind to suffer <br>
        The slings and arrows of outrageous fortune, <br>
        Or to take arms against a sea of troubles, <br>
        ...
      </p>
    </body>
  </html>
```

The browser renders this code as follows:



When you open this example in your own browser, change the size of the browser window - the first line, no matter how long or small, will always be rendered in grey.

The code example ☝ also showcases the percent unit for the `font-size` property. The base font-size of the document equates to `100%` and thus this unit allows you to scale the font-size in relation to the initial size. This is especially helpful when you design web applications for different device sizes - no additional *tuning* for different devices is required.

‼ before and after

Adding (cosmetic) content right before and after an element is achieved through:

- `::before`
- `::after`

in combination with the `content` property. These pseudo-elements are not only useful to style text in a particular manner but to also create visual artifacts (such as each cloud in our Delft weather app) that on first sight may require an image instead of just a few CSS rules. Important to know is that `::before` is the first child of the element it is attached to and `::after` is the last child of the element it is attached to.

Here is one extreme example of this concept, where all document *content* is delegated to the style sheet 👇 (the unicode characters 201C and 201D are quotation marks):

```
<!DOCTYPE html>
<html>
  <head>
    <style>
    cite::before {
      content: "\201CTo be, or ";
    }

    cite::after {
      content: "not to be ... \201D";
    }
    </style>
  </head>
  <body>
    <cite></cite>
  </body>
</html>
```

This is a poor choice admittedly as accessibility is close to zero. The document appears as content-less to a **screen reader**, a form of assistive technology that most commonly makes use of a text-to-speech engine to translate a HTML document into speech. We will come back to this issue in the next section.

Before we do that, let us quickly go over how the cloud effect was achieved in the Delft weather app with just a few lines of CSS.

Below 👇 on the right you see on the right the cloud shape we use and on the left how it looks like if all elements of it are assigned a different color: our cloud consists of a `<div>` element (a rectangle with rounded edges) and a `::before` pseudo-element (a rectangle with rounded edges that is rotated 33 degrees and moved slightly with respect to its parent) and a `::after` pseudo-element (a rectangle with rounded edges that is rotated 33 degrees moved slightly with respect to its parent):

The relevant CSS code snippets in our Delft weather app looks as follows:

```
/* the cloud itself */
.cloud {
    width: 200px;
    height: 60px;
    background: #fff;
```

```
    border-radius: 200px;
    /* these last two properties are relevant for the animation of each
cloud only */
    position: absolute;
    animation: moveclouds 15s linear infinite;
}

/* both sub-parts of cloud share a lot of rules, so we use the el1,el2
selector combination */
.cloud::before, .cloud::after {
    content: '';
    background: white;
    width: 100px;
    height: 80px;
    border-radius: 100px;
    transform: rotate(33deg);
    /* important here is to know that the before/after pseudo-elements
    * are children of the element they belong to and thus absolute
    * positioning makes sense here
    */
    position: absolute;
    top: -15px;
    left: 10px;
}

/* we overwrite a few of the previously defined properties */
.cloud::after {
    width: 110px;
    height: 90px;
    top: -45px;
    /* left: auto means that the position of the element is based on the
right property value */
    left: auto;
    right: 5px;
}
```

!! Data attributes

As just seen, CSS does not only describe the style, it *can* carry data too. There are issues though:

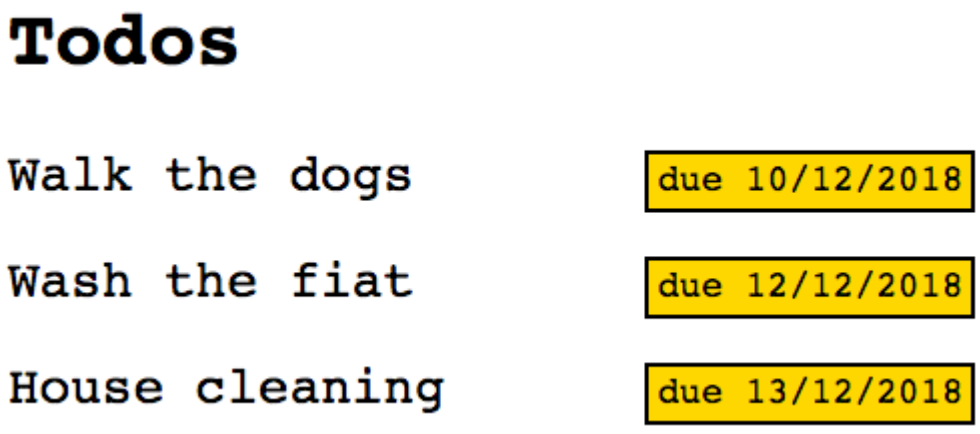
- Data is distributed across HTML and CSS files.
- CSS is conventionally not used to store data.
- Content is not part of the DOM (leading to the just described accessibility problem).

Here is another example of storing data in CSS 👇:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
```

```
      * {
        font-family: monospace;
      }

      p::after {
        background-color: gold;
        border: 1px solid;
        font-size: 70%;
        padding: 2px;
        margin-left: 50px;
      }

      p#t1::after {
        content: "due 10/12/2018";
      }

      p#t2::after {
        content: "due 12/12/2018";
      }

      p#t3::after {
        content: "due 13/12/2018";
      }
      </style>
    </head>
    <body>
      <main>
        <h2>Todos</h2>
        <p id="t1">Walk the dogs </p>
        <p id="t2">Wash the fiat </p>
        <p id="t3">House cleaning</p>
      </main>
    </body>
  </html>
```

This is rendered as follows:



Instead of storing data directly in CSS, a better way is to *make use of data stored in HTML elements*. This is achieved through so-called **data attributes**: attributes on any HTML element that are prefixed with

`data-`.

CSS can access those data attributes with the `attr()` function: it retrieves the value of the selected element and data attribute.

Rewriting the example above with `data-` attributes removes the data from CSS and reduces the amount of CSS rule duplication 👇 (the rendering remains the same):

```
<!DOCTYPE html>
<html>
  <head>
    <style>
    * {
      font-family: monospace;
    }
    p::after {
      background-color: gold;
      border: 1px solid;
      font-size: 70%;
      padding: 2px;
      margin-left: 50px;
    }

    p::after {
      content: "due " attr(data-due);
    }
    </style>
  </head>
  <body>
    <main>
      <h2>Todos</h2>
      <p id="t1" data-due="10/12/2018" data-level="urgent">Walk the dogs
</p>
      <p id="t2" data-due="12/12/2018">Wash the fiat </p>
      <p id="t3" data-due="13/12/2018">House cleaning</p>
    </main>
  </body>
</html>
```

A canonical example for `data-` attributes are tooltips 👇:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
    li {
      cursor: help;
    }

    li:hover::after{
```

```
        background-color: rgba(10,10,10,0.7);
        color: gold;
        border: 1px dashed;
        padding: 5px;
        font-size: 60%;
        content: attr(data-name);
        position: relative;
        bottom: 15px;
        left: 5px;
      }
      </style>
    </head>
    <body>
      <main>
        <ul>
          <li data-name="Cascading Style Sheets">CSS</li>
          <li data-name="HyperText Markup Language">HTML</li>
          <li data-name="HyperText Transfer Protocol">HTTP</li>
          <li data-name="HyperText Transfer Protocol Secure">https</li>
        </ul>
      </main>
    </body>
  </html>
```

This example ☝ also showcases the use of the `cursor` property. Hovering over the list items results in a help icon. Note, that `cursor: none` results in no cursor being rendered, though this should be used with care as it tends to confuse users. The `position`, `bottom` and `left` properties will be discussed next (in short: they determine the placement of the tooltip).

Element positioning

One of the more complex aspects of CSS are the myriad of ways to achieve element positioning (MDN's CSS layout guide provides you with a good idea of what options exists). We here consider three CSS properties used for element positioning:

- `float` defines how an element floats in the containing element (which in turn determines how other elements flow around it);
- `position` defines how an element is positioned in a document;
- `display` defines the display type of an element.

Let's also define two additional concepts, that stem from HTML4:

- **Block-level elements** are surrounded by line-breaks. They can contain block-level and inline elements. **The width is determined by their containing element.** Examples of block-level elements are `<main>` or `<p>`.
- **Inline elements** can be placed within block-level or inline elements. They can contain other inline elements. **The width is determined by their content.** Examples are `<span>` or `<a>`.

Note: While HTML5 has categories with more complex interactions (such as `inline-block`), for the three properties listed above, the block-level and inline notions are sufficient. In the one case where we

go beyond those notations we will point it out explicitly.

‼️ Float

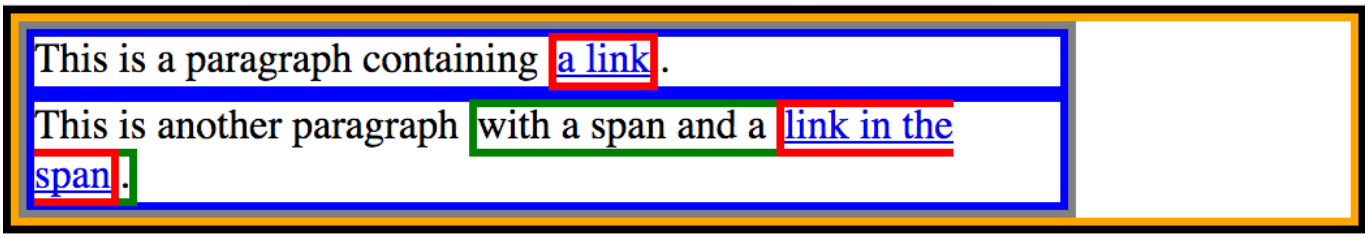
By default, elements *flow*. Their order is determined by the order of their appearance in the HTML document.

In order to understand what exactly this means, experiment with the following code 👇:

```
<!DOCTYPE html>
  <head>
    <style>

      * {
        border-style: solid;

        /* Try out what happens when the following two lines
         * are removed ("CSS reset")
        */
        margin: 0;
        padding: 0;
      }

      body {
        border-color: orange;
      }

      main {
        border-color: grey;
      }

      span {
        border-color: green;
      }

      a {
        border-color: red;
      }

      p {
        border-color: blue;
      }

      /* Block-level vs. inline elements:
       * use `width: auto` as alternative
       */
      main {
        width: 400px;
      }

      /* element floating (use left/right/none) */
      /* a {
```

```
        *  float: left;
        * }
        */

    </style>
  </head>
  <body>
    <main>
      <p>
        This is a paragraph containing <a href="#">a link</a>.
      </p>
      <p>
        This is another paragraph
        <span>
          with a span and a <a href="#">link in the span</a>.
        </span>
      </p>
    </main>
  </body>
</html>
```

Admittedly, the resulting rendering is not looking great:



However, it serves a purpose: it helps us to understand how the different elements appear within their parent element.

☝ Try out the following code variations (remember, that you can make these changes directly in the *Style Editor* of your browser):

- Manually resize the browser window to explore how the block-level and inline elements render at different window sizes.
- In the `*{...}` we have employed a CSS reset, by setting both `margin` and `padding` to `0`. Remove those two lines of code.
- Replace `main { width: 400px; }` with `main { width: auto; }` and once more manually resize the browser window.

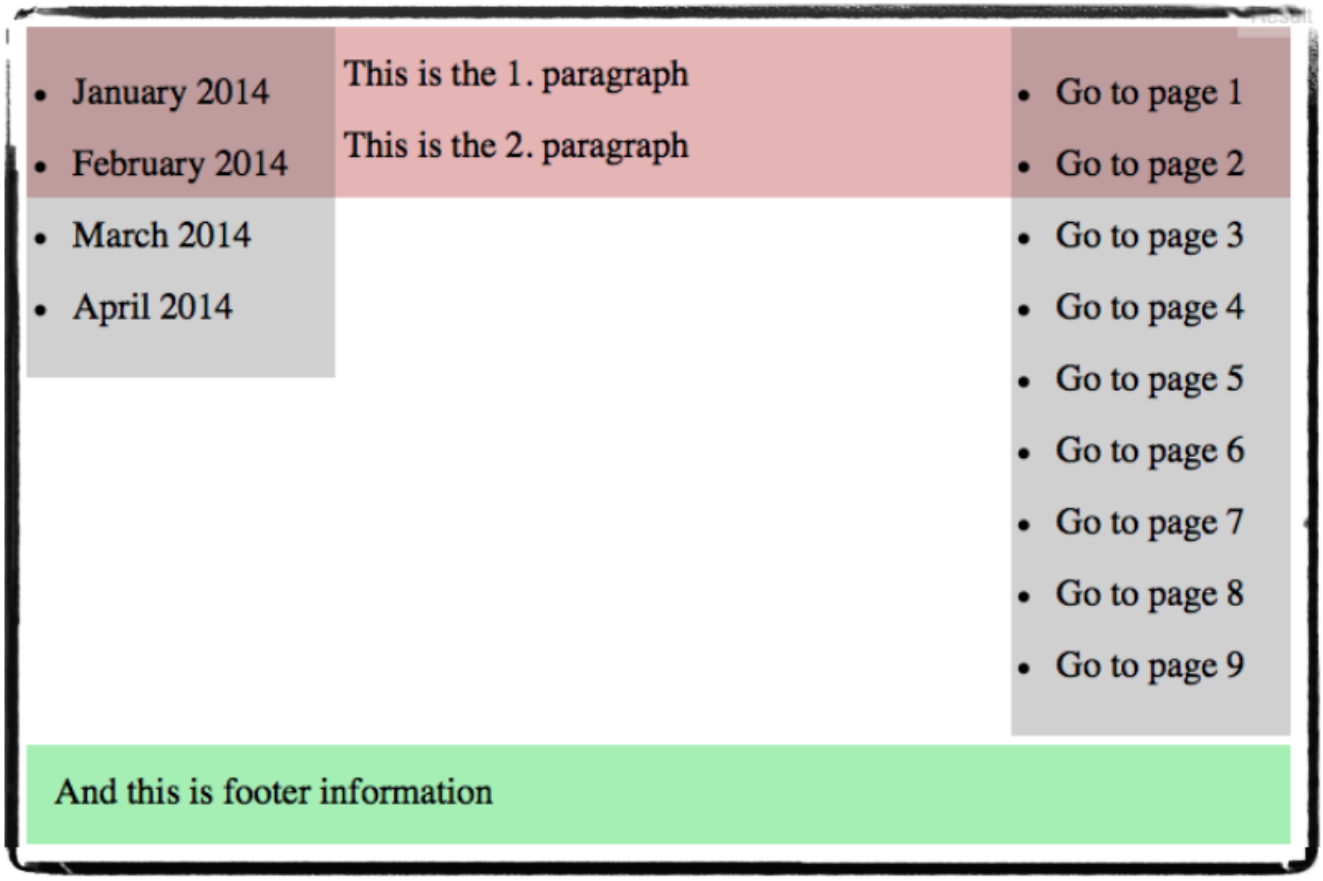
Let's now explore the use of `float`. With the `float` property, we can change the default flow. `float:left` (or `float:right`) takes an element **out of the flow**; it is then moved to the **leftmost** (or rightmost) possible position **in the containing element** - which is either the element edge or another float. In addition, if an element should not float, we can use `float: none`.

Try out the effects of `float` on the code example ☝ by using one after the other:

- `a {float: right}`

- `a {float: left}`
- `a {float: none}`

If needed, we can also reset the flow the the value `clear`, either to reset the `left`, `right` or `both` sides of the flow. The canonical example for flow resetting are web page layouts that have **sidebars**, such as this one 👇:



Here, we have two sidebars that each *float* to the left and right of the main content respectively. The footer should appear below both sidebars. Try out the HTML below 👇 to see the effect of both `float` and `clear`: remove the commenting of one CSS rule at a time.

```
<!DOCTYPE html>
  <head>
    <style>

    .nav {
      background-color: grey;
    }

    main {
      background-color: red;
    }

    footer {
      background-color: green;
    }
```

```
    /* Floating to the sides */
    /*
     * #nav1 {
     *  float: right;
     * }
     */

    /* Floating to the sides */
    /*
     * #nav2 {
     *  float: left;
     * }
     */

    /*
     * Resetting the flow: try what happens if clear:left or clear:right
     * is used alone. Instead of both left and right, we can also use
     * clear:both;
     */
    /*
     * footer {
     *  clear: left;
     *  clear: right;
     * }
     */
  </style>
</head>
<body>
  <div id="nav1" class="nav">
    <ul>
      <li>Go to page 1</li>
      <li>Go to page 2</li>
      <li>Go to page 3</li>
      <li>Go to page 4</li>
      <li>Go to page 5</li>
    </ul>
  </div>

  <div id="nav2" class="nav">
    <ul>
      <li>January 2019</li>
      <li>February 2019</li>
      <li>March 2019</li>
      <li>April 2019</li>
    </ul>
  </div>

  <main>
    <p>
      Paragraph 1.
    </p>

    <p>
```

```
        Paragraph 2.
      </p>
    </main>

    <footer>
      Footer information.
    </footer>
  </body>
</html>
```

!! Position

The `position` property enables fine-grained movement of elements. This is in contrast to `float`, which meets our demands for coarse-grained positioning. Elements can be moved around in any direction (up/down/left/right) by absolute or relative units.

The `position` property has a number of possible values:

Value	Description
`position:static`	the default
`position:relative`	the element is adjusted on the fly, other elements are **not** affected
`position:absolute`	the element is taken out of the normal flow (**no space is reserved for it**)
`position:fixed`	similar to `absolute`, but fixed to the **viewport** (=the area currently being viewed)
`position:sticky`	in-between `relative` and `fixed` (we do not consider it further in this class)

Important to know when using the `position` property is the direction of the CSS coordinate system: the top-left corner is `(0,0)`. The y-axis extends **downwards**. The x-axis extends to the **right**.

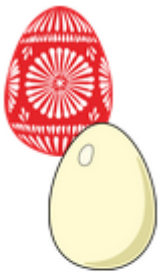
Let's walk through each of the position values in turn, starting with `relative`, where the movement of the element is **relative** to an element's original position. The horizontal offset from the original position is set through properties `left` and `right`, the vertical offset is controlled through `top` and `bottom`.

For the `position` property, we use a stack of eggs and reposition one or more eggs at a time. The original stack looks like this:

The following code uses relative positioning to reposition two of the four eggs:

```
<!DOCTYPE html>
  <head>
    <style>

    img {
      width: 50px;
    }
    /*
     * Try out different px values (px is the magic unit of CSS).
     * Remember that the coordinate system starts at the top-left
     * corner and extends downward and to the right!
     * What happens if the px value is negative?
     * At what point do the "eggs" leave the viewport?
     */
    #egg2 {
      position: relative;
      bottom: 20px;
      left: 20px;
    }

    #egg4 {
      position: relative;
      bottom: 50px;
      right: 10px;
    }

    </style>
  </head>
  <body>
    <main>
        <img
src="https://openclipart.org/image/300px/svg_to_png/19477/shokunin_easter_
```

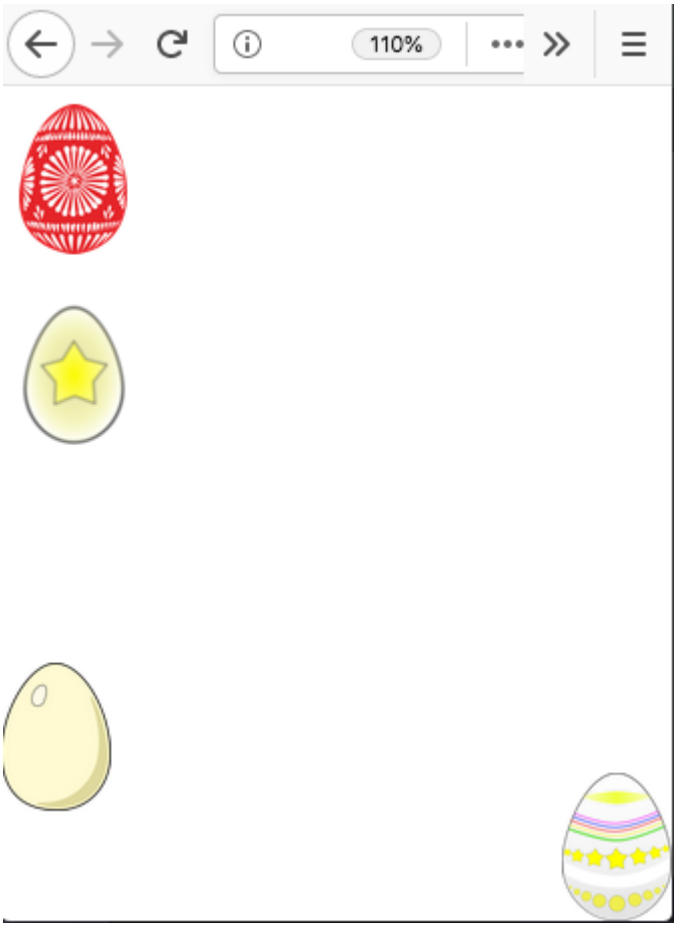
```
egg_single.png" id="egg1" /><br />
        <img
src="https://openclipart.org/image/300px/svg_to_png/6695/dStulle_white_egg
.png" id="egg2" /> <br />
        <img
src="https://openclipart.org/image/300px/svg_to_png/15696/mystica_Easter_e
gg_%28star%29.png" id="egg3" /> <br />
        <img
src="https://openclipart.org/image/300px/svg_to_png/15694/mystica_Easter_e
gg_%28Painted%29.png" id="egg4" /><br />
    </main>
  </body>
</html>
```



If we change the positioning of some eggs to `position:absolute`, we now see that **they are taken out of the normal flow** and **no space** is reserved for them. The positioning of those elements is now relative to the *nearest ancestor* or the window itself (in our case it is the window itself):

```
<!DOCTYPE html>
  <head>
    <style>

    img {
      width: 50px;
    }

    /*
     * Try out different px values.
     * Remember that the coordinate system starts at the top-left
     * corner and extends downward and to the right!
     */
    #egg2 {
      position: absolute;
      bottom: 50px;
      left: 0px;
```

```
    }

    #egg4 {
      position: absolute;
      bottom: 0px;
      right: 0px;
    }

    </style>
  </head>
  <body>
    <main>
        <img
src="https://openclipart.org/image/300px/svg_to_png/19477/shokunin_easter_
egg_single.png" id="egg1" /><br />
        <img
src="https://openclipart.org/image/300px/svg_to_png/6695/dStulle_white_egg
.png" id="egg2" /> <br />
        <img
src="https://openclipart.org/image/300px/svg_to_png/15696/mystica_Easter_e
gg_%28star%29.png" id="egg3" /> <br />
        <img
src="https://openclipart.org/image/300px/svg_to_png/15694/mystica_Easter_e
gg_%28Painted%29.png" id="egg4" /><br />
    </main>
  </body>
</html>
```



The `position:fixed` value is similar to `position:absolute`, but now the containing element is the **viewport**, i.e. the area of the document that is visible in the browser. This means that elements with `position:fixed` remain visible. Here is an example (you need to minimize the window until you need to scroll down to see all of the four eggs to achieve a visible effect):

```
<!DOCTYPE html>
  <head>
    <style>

    img {
      width: 50px;
    }

    #info {
      position: fixed;
      background: yellow;
      left: 20px;
      top: 20px;
    }

    </style>
  </head>
  <body>
    <main>
        <img
src="https://openclipart.org/image/300px/svg_to_png/19477/shokunin_easter_
egg_single.png" id="egg1" /><br />
        <img
src="https://openclipart.org/image/300px/svg_to_png/6695/dStulle_white_egg
.png" id="egg2" /> <br />
        <img
src="https://openclipart.org/image/300px/svg_to_png/15696/mystica_Easter_e
gg_%28star%29.png" id="egg3" /> <br />
        <img
src="https://openclipart.org/image/300px/svg_to_png/15694/mystica_Easter_e
gg_%28Painted%29.png" id="egg4" /><br />
    </main>

    <div id="info">
        These are no easter eggs.
    </div>
  </body>
</html>
```

Lastly, we cover the `display` property, which enables us to change the element type at will (block-level to inline and vice versa) at will and *hide* elements from view:

Value	Description
`display:inline`	The element is treat as an inline element.

Value	Description
`display:block`	The element is treated as a block element (line breaks before and after the element).
`display:none`	The element (and its descendents) are hidden from view; no space is reserved in the layout.
`display:inline-block`	A block element (height/width can be set) that does result in line breaks before/after the element.

Once more, an example is sufficient to highlight the use of each of these values - uncomment the CSS rules one at a time:

```
<!DOCTYPE html>
  <head>
    <style>
      p, span {
        border-style: solid;
        padding: 10px;
        margin: 10px;
      }
      p {
        border-color: blue;
      }

      /*
       * Try out the effect of adding each of these
       * display settings.
       */

      /*
       * span {
       *  border-color: red;
       * }
       */

      /*
       * span {
       *  display: block;
       * }
       */

      /*
       * p {
       *   display: inline;
       * }
       */

      /*
       * span {
       *  display: none;
```

```
         * }
         */

        /*
         * p {
         *  display: inline-block;
         *  width: 300px;
         * }
         */
      </style>
    </head>
    <body>
      <p>
        This is paragraph one.
      </p>

      <span>
        Span element one.
      </span>

      <span>
        Span element two.
      </span>

      <p>
        This is paragraph two.
      </p>
    </body>
  </html>
```

CSS media queries

So far, we have covered the basics of CSS but largely ignored the fact that in today's **multi-device** world, we are designing web applications for vastly different screen sizes. Different devices should be served different styles:

- when **printing** a web application's screen  , the information printed should be the essentials (no ads, no sidebars, etc.);
- when **viewing** a web application on a small screen  , non-essential information (e.g. a footer) should be removed;
- when **viewing** a web application on a large screen  all available information should be presented;
- when using **text-to-speech** devices  , non-essential information should be removed.

CSS media queries enable the use of **device/media-type dependent** style sheets. While the HTML document is written once, the CSS is written once per device type. There are four device types currently in use:

Value	Description

Value	Description
`media all`	Suitable for all device types.
`media print`	Suitable for documents in print preview.
`media screen`	Suitable for screens.
`media speech`	Suitable for speech synthesizers.

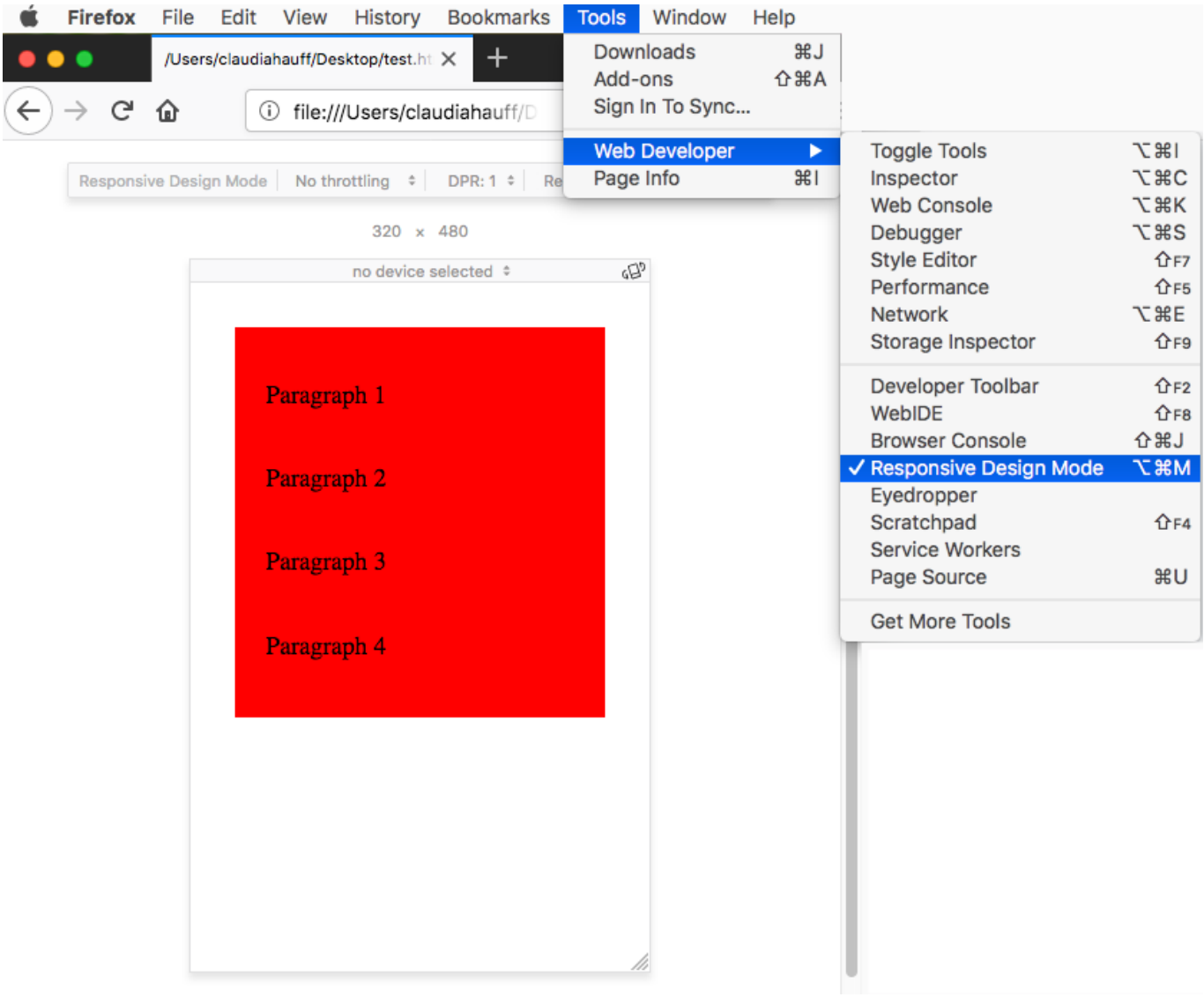
Media queries are specified as so-called at-rules, that start with an `@` and instruct CSS how to behave. We will not only encounter at-rules for media queries, but also when discussing CSS animations later on.

Here is a concrete example of how media queries enable a **responsive design** 👇:

```
<!DOCTYPE html>
<html>
  <head>

    <!-- We can link a style sheet conditional on the media attribute -->
    <!-- Example of a logical or (,) in the media query -->
    <link rel="stylesheet" media="screen and (min-width: 800px), (min-
width: 3000px)" href="large-device.css">

    <style>
      * {
        padding: 10px;
      }

      main {
        background-color: red;
      }

      #sidebar {
        background-color: green;
        float: right;
      }

      /* when printing, use black and white */
      @media print {
        body {
          color: black !important;
          width: 100%;
        }
      }

      /* hide the sidebar for small devices */
      /* Example of a logical or (",") as well as and ("and") in the media
query */
      @media print, screen and (max-width: 400px) {
        #sidebar {
          display: none;
        }
```

```
        }
      </style>
    </head>
    <body>
      <div id="sidebar">
        <ul>
          <li>Anchor 1</li>
          <li>Anchor 2</li>
          <li>Anchor 3</li>
        </ul>
      </div>

      <main>
        <p>
          Paragraph 1
        </p>
        <p>
          Paragraph 2
        </p>
        <p>
          Paragraph 3
        </p>
        <p>
          Paragraph 4
        </p>
      </main>
    </body>
  </html>
```

Use your browser's responsive design mode to test the behaviour of the media queries. For `@media print` you can use the **Print simulation** mode in the Developer Tools of Firefox, or alternatively, the browser's *Print as PDF* feature.



Animations and transitions

In general, CSS styles (states) are defined by the developer. The **rendering engine** takes care of the transition between styles. A rendering engine - also known as *browser engine* or *layout engine* - is responsible for translating HTML+CSS (among others) to the screen. The major browsers ship with their own rendering engines, the names of which you will encounter from time to time, especially when using CSS animations and transitions:

Engine	Browsers
`Gecko`	Firefox
`Trident`	Internet Explorer
`EdgeHTML`	Microsoft Edge
`WebKit`	Safari, older versions of Google Chrome
`Blink`	Google Chrome, Opera

Rendering engines do a lot of heavy lifting, and can also be attacked. This GitHub Gist is an example of a Safari DoS (Denial-of-service) attack; the device running Safari crashes after trying to render 3485 nested `<div>` elements!

Animations consist of:

- an animation style (e.g. `linear`, 3 seconds duration, 10 times);
- a number of **keyframes** that act as transition waypoints.

Transitions are animations with a simpler syntax. They consist of exactly **two states**: start and end state.

CSS vs. JavaScript animations

There are several advantages to using CSS-based instead of JavaScript-based animations:

- CSS is relatively easy to use and debugging them is easier than debugging JavaScript code.
- The rendering engines are optimized for CSS-based animations; there is no need to optimize your JavaScript code.
- CSS animations can do much more than animating buttons. To see what is possible (beyond the Delft weather app demo), head over to CodePen and look at a few CSS animations.

Animations

We look at two animation examples in more detail: a TU DELFT neon sign and the Delft weather app. The former is focused on text-only content, the latter is heavy on non-textual content (clouds, raindrops, etc.).

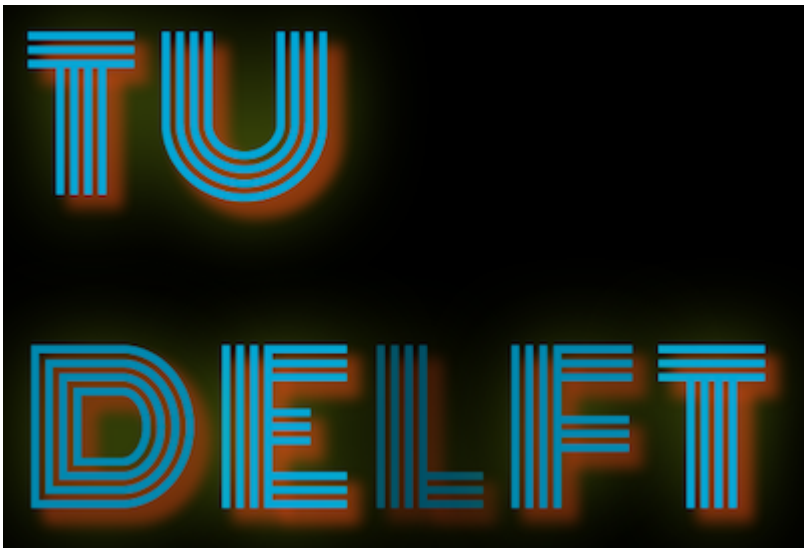
‼️ Animations: TU DELFT neon sign

We adapted the neon sign based on this this CodePen example 👇:

```
<!DOCTYPE html>
<html>
  <head>
    <!-- Loading a specific font:
https://fonts.google.com/specimen/Monoton -->
    <link href="https://fonts.googleapis.com/css?family=Monoton"
rel="stylesheet">

    <style>
      body{
        background: black;
      }

      .neon{
        font-family: 'Monoton', cursive; /* referencing the loaded font */
        font-size: 150px;
        color: #00A6D6;
        position: absolute;
        top: 50%;
        left: 50%;
        transform: translate(-50%, -50%);
        font-weight: 400;
        letter-spacing: 10px;
```

```
        text-shadow: 10px 10px 10px #E64616, 5px 5px 60px #A5CA1A;
      }

      /*
       * Instead of linear, also try out `ease-in`, `ease-out`,
`steps(10)`.
       */
      .flicker-slow{
        animation: flicker 3s linear infinite;
      }

      .flicker-fast{
        animation: flicker 1s linear infinite;
      }
      /*
       * The above short-hand can be replaced by the
       * following animation properties:
       */

      /*
      .flicker-slow{
        animation-name: flicker;
        animation-duration: 3s;
        animation-timing-function: linear;
        animation-iteration-count: infinite;
      }

      .flicker-fast{
        animation-name: flicker;
        animation-duration: 1s;
        animation-timing-function: linear;
        animation-iteration-count: infinite;
      }
      */

      @keyframes flicker {
        0%, 30%, 33%, 55%, 100% {
            opacity: .99;
        }

        31%, 50%, 56% {
          opacity: 0.3;
        }
      }
    </style>
  </head>
  <body>
    <h1 class="neon">
      TU <span class="flicker-slow">D</span>E<span class="flicker-
fast">L</span><span class="flicker-slow">F</span>T
    </h1>
  </body>
</html>
```

It renders as follows:



The example ☝ contains a number of interesting points:

- It is easy to load additional fonts. A popular free font service is Google Fonts; more information on how to use it is available from MDN.
- The CSS property `text-shadow` adds - as the name suggests - shadow to text. Importantly, a list of shadows can be added in a comma-separated list which are applied front-to-back (the first shadow is on top). Shadows can be defined in a number of ways, we here stick to a single one: `offset-x offset-y blur-radius color`.
- The `animation` property is a short-hand property that combines a number of `animation-*` properties in one line. Specifically in our example `animation: flicker 3s linear infinite` refers to the keyframes (`flicker` - the `animation-name`), the duration of one animation cycle (3 seconds - the `animation-duration`), the `animation-timing-function` (`linear` means that the animation moves from start to end in a constant rate) and the `animation-iteration-count` (here: `infinite`, i.e. the animation never stops). We defined here two types of flickers: a slow flicker (3 seconds to complete a cycle) and a fast flicker (1 second to complete the cycle). Different letters of our `TU Delft` string are assigned to different *flicker classes*.
- The `@keyframes` control the intermediate steps in a CSS animation. In order to achieve flickering we change the opacity of the text string. In one animation cycle, we repeatedly move from an opacity of `.99` to `.3` and back. Specifically, we define 8 waypoints of our animation (with either opacity of `.99` or `.3`): `0%, 30%, 31%, 33%, 50%, 55%, 56%, 100%`. The rendering engine is then responsible to turn this code into an actual animation that resembles flickering.

To summarize, for us the most important animation properties are the following:

Property	Despcription
`animation-iteration-count`	Number of times an animation is executed (default: 1); the value is either a positive number or `infinite`.
`animation-direction`	By default the animation restarts at the starting keyframe; if set to `alternate` the animation direction changes every iteration.

Property	Despcription
`animation-delay`	Number of seconds (s) or milliseconds (ms) until the animation starts (default 0s). A **negative** value (e.g. `x=-5s`) means that the animation starts immediately but already `x` seconds into the animation cycle
`animation-name`	Identifies the `@keyframes`.
`animation-duration`	The duration of a single animation cycle in seconds (s) or milliseconds (ms), e.g. `2.5s`, `500ms`.
`animation-timing-function`	Specifies how a CSS animation progresses between waypoints (common choices are `linear`, `ease-in`, `ease-out`, `steps(N)`).

Let's look at a second example 👇, which shows a slightly different way to define keyframe waypoints: instead of `0%` the start state can also be defined with `from`, while the end state (`100%`) can be defined with `to` (and these two can also be mixed with other waypoints such as `50%`). This animation slides TU Delft's letters in place, with different letters moving at different speeds:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      body {
        background-color: black;
        width: 900px;
        margin: 100px auto;
      }

      .box {
        width: auto;
        height: 100px;
        background-color: #00a6d6;
        margin: 0 auto;
        float: left;
        font-size: 100px;
        padding: 20px;
        color: white;
        text-shadow: 3px 3px 2px black;
      }

      @keyframes move {
        from {
          transform: translate(0px, 1000px)
        }

        to   {
          transform: translate(0px, 0px)
        }
```

```
        }

        #box1, #box7 {
          animation: move 1s;
        }

        #box2, #box8 {
          animation: move 4s;
        }

        #box3, #box6 {
          animation: move 2s;
        }

        #box4, #box5 {
          animation: move 3s;
        }
      </style>
    </head>
    <body>
        <div class="box" id="box1">T</div>
        <div class="box" id="box2">U</div>
        <div class="box" id="box3">  </div>
        <div class="box" id="box4">D</div>
        <div class="box" id="box5">E</div>
        <div class="box" id="box6">L</div>
        <div class="box" id="box7">F</div>
        <div class="box" id="box8">T</div>
    </body>
</html>
```

☝ The animation's start state is defined through the `transform` property, which allows us to rotate, scale, skew and translate an element: at the start of the animation, the elements are moved (`translate`) from their original position to a position that is 1000 pixels down. The end state of the animation moves the elements to position 0/0 again. While this may seem to move all letter elements to the same position, the `float:left` property ensures that the letters appear next to each other as intended.

‼ Animations: Delft weather app

We now go over the **lightning** effect in our Delft weather app: it consists of a single `<div>` covering the viewport. The animation takes 5 seconds and for 75% of the time does nothing. Then it briefly paints the `<div>` background white with increasing opacity (that's lightning), the `<div>` becomes transparent again and then this happens a second time. So here, one animation cycle results in two *lightning bolts*.

Once again, we only have to define the states of the animation, the rendering engine does the rest.

```
/* +++++++++++++++++++ LIGHTNING ++++++++++++++++++++
 * Adapted/inspired from https://codepen.io/Chrislion_me/pen/rVqwbO
 */
.lightning {
```

```
  /* the entire background-image should be filled with lightning;
   * thus we position with respect to the parent (<body>) at 0/0
   */
  position: absolute;
  width: 100%;
  height: 100%;

  animation: flash ease-out 5s infinite;
}

@keyframes flash {
  /* basic idea: the white div covering the entire screen appears
   * very briefly with varying levels of opacity;
   * The from/74% duplication of opacity: 0 is important here, as
   * otherwise the transition from the start to 75% is a gradually
   * whiter screen.
   */
  from { opacity: 0; }
  74% { opacity: 0;}
  75% { background-color: white;  opacity: 0.6; }
  76% { background-color: white;  opacity: 0.2; }
  80% { background-color: white;  opacity: 0.95; }
  82% { opacity: 0; }
  92% { opacity: 0; }
  93% { background-color: white;  opacity: 0.5; }
  94% { background-color: white;  opacity: 0.2; }
  96% { background-color: white;  opacity: 0.9; }
  to { opacity: 0; }
}
```

Beyond lightning, we also have an animated moon: it appears to be radiating continously if you look closely. This is a simple animation of the moon's `box-shadow` property. The moon itself is a `<div>` element with a `border-radius` set such that it results in a perfect circle :

```


#moon {
  /* the moon is positioned with respect to the <body> element at
   * the top left corner
   */
  position: absolute;
  left: 160px;
  top: 80px;
  border-radius: 50%; /* making it round */
  height: 100px;
  width: 100px;
  border: 1px solid black; /* a black line around the circle*/
  /* we create a background image that has a radial gradient
   * going from dark-grey to white
   */
  background: radial-gradient(circle at 20px 10px, #333, #fff);
  /* shadow: has no offset (equal shadow on all sides) and a large blue
```

```
  value */
    box-shadow: 0 0 25px #fff;
    animation: flicker 3s infinite;
  }

  @keyframes flicker {
    0% {
      box-shadow: 0 0 25px #fff;
    }
    25% {
      box-shadow: 0 0 60px #fff;
    }
    50% {
        box-shadow: 0 0 50px #fff;
    }
    90% {
        box-shadow: 0 0 27px #fff;
    }
  }
```

Lastly, while lightning and the moon were done in *pure CSS*, for the raindrops and the clouds we made use of a bit of JavaScript: note though that this is not strictly necessary. Since we need hundreds of raindrops/clouds that are all slightly different (slightly different locations, size, color, opacity, speed) we would need to create hundreds of different CSS rules. JavaScript makes this much easier - we create these rules programmatically. Let's quickly look at how the clouds were created 👇:

Some of the CSS parts were put directly into the CSS files - those that are the same for every cloud `<div>` we create 👇:

```
  .cloud {
      width: 200px; height: 60px;
      background: #fff;
      border-radius: 200px;
      position: absolute;
    /* z-index:10; with this, the cloud is in front of the switches */
    animation: moveclouds 15s linear infinite;

  }

  .cloud::before, .cloud::after {
      content: '';
      background: white;
      width: 100px; height: 80px;

    /* important here is to know that the before/after pseudo-elements
     * are children of the element they belong to and thus absolute
     * positioning makes sense here
     */
      position: absolute;
    top: -15px;
```

```
  left: 10px;

    border-radius: 100px;
    transform: rotate(33deg);
}

.cloud::after {
    width: 110px; height: 90px;
    top: -45px;
  /* left: auto means that the position of the element is based on the
right property value */
  left: auto;
  right: 5px;
}

/* simple animation: clouds go from left to right */
@keyframes moveclouds {
    0% {margin-left: -20%;}
    100% {margin-left: 120%;}
}
```

The JavaScript snippet then creates `<div>` elements and assigns to each the `.cloud` class as well as the properties unique to each cloud. For this to work, it is vital that the created element is added to the DOM tree (the rendering engine will not render it otherwise). For this reason, our HTML file contains a `<div id="clouds">` element and all generated individual cloud elements are added as children of it with the line `clouds.appendChild(c)` 👇:

```
var totalNumClouds = 30;

/*
 * checkbox unchecked: find the <div> with id "clouds" and delete all its
children.
 * checkbox checked: find the <div> with id "clouds" and create four <div>
elements
 * with class .cloud and id's c1 to c4.
 */
function toggleClouds(e){
  let clouds = document.getElementById("clouds");

  if(document.getElementById("cloudsCheckbox").checked==false){
    while(clouds.firstChild){
      clouds.removeChild(clouds.firstChild);
    }
  }
  else {
    for(let i=1; i<=totalNumClouds; i++){
      let c = document.createElement("div");
      c.classList.add("cloud");//class .cloud in the CSS file

      //random location
      c.style.top=(50+getRandomInt(120))+"px";
```

```
        //random opacity
        c.style.opacity = Math.random();

        //random animation speed
        c.style.animationDuration = (15+getRandomInt(20))+"s";

        //random starting point within the animation
        c.style.animationDelay = (-1 * getRandomInt(40))+"s";

        //slightly different size per cloud
        c.style.transform = "scale("+(0.1+Math.random())+")";
        clouds.appendChild(c);//add element to the DOM tree
      }
    }
  }
```

Note: the rain drops are generated in a similar fashion, so we do not go into the code details here.

‼️ Transitions

Transitions are animations with only two states (a start state and an end state).

We actually have been using transitions all this time already, e.g. when defining `:hover`. In this case, the transition is from the original element style to the hover style.

We can make use of the `transition` property to control the animation between the start and end state. The example below 👇 shows off what the impact of `transition` is: we define two boxes which, when being hovered over, change their rotation by 70 degrees.

- `#box1` has no `transition` property, which makes the style change instanteneous;
- `#box2` has the `transition` property which determines how fast the transition for different properties takes place and we see a smooth animation when hovering.

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      #box1 {
        width: 100px;
        height: 100px;
        background-color: lightgreen;
      }

      #box1:hover {
        transform: rotate(70deg);
        background-color: darkgreen;
      }

      #box2 {
```

```
        width: 200px;
        height: 200px;
        background-color: yellow;
        transition: width 2s, height 2s, background-color 2s, transform
2s;
      }

      #box2:hover {
        transform: rotate(70deg);
        background-color: red;
      }
    </style>
  </head>
  <body>
    <div id="box1">
    </div>

    <div id="box2">
    </div>
  </body>
</html>
```

☝ The line `transition: width 2s, height 2s, background-color 2s, transform 2s` is a short-hand for the `transition-property` (the CSS property to which a transition should be applied) and the `transition-duration` (the seconds or milliseconds until the transition is complete). As seen here, we can define multiple transition properties in one line.

Browser-specific prefixes

A last word on **vendor prefixes**. Older existing CSS code snippets will contain vendor prefixes. As an example, consider this Stack Overflow question. It contains CSS properties such as `-webkit-transition`, `-webkit-animation-duration` and so on. In the past, browser vendors decided to use browser-specific prefixes to include experimental CSS features in their rendering engines. This led to a lot of duplicate code, e.g.

```
 main:-webkit-full-screen {
 } /* Chrome */
 main:-moz-full-screen {
 } /* Firefox */
main:-ms-fullscreen {
} /* Internet Explorer */
main:fullscreen {
} /* W3C proposal */
```

This approach has now been deprecated and although the vendor-specific prefixed properties still work, the use of such properties should be avoided.

Self-check

Here are a few questions you should be able to answer after having followed the lecture and having worked through the required readings:

- Through which mechanism can data be stored directly in CSS files?
- True or False? A single DOM element cannot contain multiple classes.
- True or False? A `class` attribute can be added to any DOM element.
- What is the main purpose of a "CSS reset"?
- What are the differences between CSS animations and CSS transitions?
- What is a rendering engine?

//TODO: add more