Programming JavaScript

Santosh Kalwar, 14.02.2022

JavaScript Topics

- DOM related misc topics
  - append and prepend
  - dataset

- DOM forms and advanced events concept
  - Visualize and Practice

- Introduction to Object-Oriented Programming
  - Classes
  - Class concepts
  - Constructor
  - Practice

- Supermini Projects
  - Supermini project 06
  - Supermini project 07
  - Supermini project 08

DOM related topics

There are a lot of scenarios where you need to add some HTML to an existing one.
Using `innerHTML` is not effi cient because the browser has to reconstruct the entire content of the element. For example, the code below:

```
< div  id="container">
        <p>Hello  World</p>
</div>>
const     div     =   document.querySelector( "#container") ;

div.innerHTML  +=  `<p>Another  paragraph</p>`;
```

The `div.innerHTML += ...` (which is similar to `div.innerHTML = div.innerHTML + ...`) will recreate all the previous content alongside the new one (so, the previous paragraph will be removed then recreated alongside the new one).

This is not very effi cient and could create some issues if you had event listeners. Instead, we need a method that allows us to add some HTML to the end or the beginning of the element. We call this *append* or *prepend*.

The word *append* means to add something at the end. And, *prepend* means to add something at the beginning .

Let's learn about the method that allows us to append/prepend HTML.

element.insertAdjacentHTML(position, htmlString)

The element.insertAdjacentHTML will place the htmlString without having to reconstruct the remaining HTML inside the element. It could either prepend or append depending on the position that you provide.

Assuming the following HTML code:

```
<div id="job-positions">
    <div class="position">2009-2013</div>
    <div class="position">2013-2015</div>
</div>
```

Append

Let's see how you can **append** (add at the end):

```
const positions = document.querySelector("#job-positions");
positions.insertAdjacentHTML("beforeend", `<div class="position">2015-2020</div>`);
```

This will add the html string at the end of the #job-positions element. So, the new **DOM** will look like the following:

```
<div id="job-positions">
    <div class="position">2009-2013</div>
    <div class="position">2013-2015</div>
    <div class="position">2015-2020</div>
</div>
```

The beforeend as a position means that the htmlString should be inserted before the end of the element. The previous elements remain intact and do not need to be reconstructed by the browser making it faster and more effi cient.

Prepend

Prepend

Similarly, you can use the same method with the position set to afterbegin to **prepend** (place at the beginning):

```
const positions = document.querySelector("#job-positions") ;
positions.insertAdjacentHTML( "afterbegin", `<div
class="position">2007-2009</div>`);
```

Assuming we run this JS code against the fi rst HTML code given at the top of the lesson, the resulting DOM would be:

```
< div id="job-positions" >
        <div class="position">2007-2009</div>
        <div class="position">2009-2013</div>
        <div class="position">2013-2015</div>
</div>>
```

So, depending on what you use, you can choose between beforeend and afterbegin. To make it easier for yourself, always go with beforeend and then change it to afterbegin if necessary.

Other positions

Other positions

While there are other positions, they are very rarely used. If you're curious, you can take a look at them in the MDN link below:

element.insertAdjacentHTML() on MDN

Use backticks for htmlString

The 2nd argument of this method expects an htmlString that will be prepended or appended to the DOM.

It's very common that this htmlString will span multiple lines, which is why we recommend that you use the backtick character to create a **template string**:

```
element.insertAdjacentHTML( "beforeend", `<p>An example of a...
        ... very long paragraph</p>`);
```

This is because template strings support multiline strings whereas double quotes and single quotes don't.

Another benefi t of using template strings is **interpolation** .

Array to DOM

It's very common to have an array of items that you'd like to insert into the DOM. For that, you can use a combination of forEach on the array and insertAdjacentHTML:

```
<ul id="apps-list"></ul>
const apps = ["Calculator", "Phone", "Messages"];
const list = document.querySelector("#apps-list");

apps.forEach(app => {
    list.insertAdjacentHTML("beforeend", `<li>${app}</li>`);
});
```

The resulting DOM will be:

```
<ul id="apps-list">
    <li>Calculator</li>
    <li>Phone</li>
    <li>Messages</li>
</ul>
```

Notice how we iterate over the array and then call `insertAdjacentHTML` on `list` and pass the `htmlString`. The `htmlString` is an `<li>` followed by the `app` (interpolated), and followed by `</li>`.

innerHTML vs insertAdjacentHTML

Both of these methods have their usages. In summary, you can ask yourself these questions:

- Do I want to write HTML and overwrite all the previous values? If yes, then use `innerHTML`.
- Do I want to keep the previous HTML and add some HTML at the beginning or at the end? If yes, then use `insertAdjacentHTML`.

insertAdjacentHTML's security risk

The `insertAdjacentHTML` method presents the same security risk as `innerHTML`.
So, you should not use it if the variables you're interpolating might be coming from the user.

Similarly, we've got the `insertAdjacentText` method that will insert text without interpreting HTML.

element.insertAdjacentText() on MDN

Chapter recap

- innerHTML += ... is ineffi cient because it recreates the entire HTML. This could also remove existing event listeners.
- Instead, when you want to add a piece of HTML, you should use the insertAdjacentHTML method.
- element.insertAdjacentHTML(position, htmlString) will prepend/append the htmlString depending on the position.
- A position of beforeend will append (add at the end).
- A position of afterbegin will prepend (add at the beginning).
- Use backticks with the htmlString of insertAdjacentHTML as it makes your life easier (multiline string + interpolation support).
- Do I want to write HTML and overwrite all the previous values? If yes, then use innerHTML.
- Do I want to keep the previous HTML and add some HTML at the beginning or at the end? If yes, then use insertAdjacentHTML.
- The insertAdjacentHTML method presents the same security risk as innerHTML. If the variables are provided by the user, then prefer using insertAdjacentText instead.

Dataset

There are multiple ways to build web applications. One of those ways is when your app is completely powered by a backend such as Laravel (PHP), Symfony (PHP), Rails (Ruby), Django (Python), Express.js (NodeJS), etc...
In those scenarios, you may need to pass some data from the server into your JavaScript code.
This can be done by writing an attribute to an HTML element.

For example:

```
<!-- This works, but DON'T use it. -->
<form id="payment-form" currency=" EUR" >
    ...
</form>
```

You will be able to read this attribute with `element.getAttribute("currency")`.
While this works, it's not recommended because what if the browser ends up adding an attribute called `currency` in the future? Your code would break.

To avoid this issue, the HTML spec recommends that developers prefi x their own custom attributes with `data-`.
So, the HTML becomes:

```
<!-- This is recommended ☑ -->✅
<form id="payment-form" data-currency="EUR">
    ...
</form>
```

Dataset

This avoids potential confl icts with the browser and shows that this is a custom attribute. We call this a data attribute.

To read this data attribute, you can access the dataset object on the element:

```
const form = document.querySelector("#payment-form") ;
console.log(form.dataset); // {currency: "EUR"}
const currency = form.dataset.currency; // "EUR"
```

Notice how the data- attributes are automatically collected in the dataset object that you can access.

Let's take a more advanced example:

```
< form id="payment-form" data-currency="EUR" data-user-id=" 2" >
        ...
</form>>
```

If we log the dataset object with console.log(form.dataset), you will get:

```
{
        userId: "2",
        currency: "EUR"
}
```

Notice that the kebab-case (user-id) was automatically converted to camelCase (userId).
Also, notice that the value of data attributes is **always** a string.

Write dataset

You can also update/set a value for a data attribute by assigning it to a new value .

Assuming the following HTML :

```
<div id="navbar" data-user-id="42"></div>
const navbar = document.querySelector("#navbar");
navbar.dataset.userId = 43;
navbar.dataset.rememberMe = false;
```

The updated DOM element will look like this :

```
<div id="navbar" data-user-id="43" data-remember-me="false"></div>
```

Notice that the value is always a string. This may become challenging when reading boolean values.
The "false" string is true (because it's a string that contains text) .

So, in this scenario, you can compare the string to "true" . So, if the string is "true" it will return true . In all other cases, it will return false :

```
const rememberMe = navbar.dataset.rememberMe === "true"; // false
```

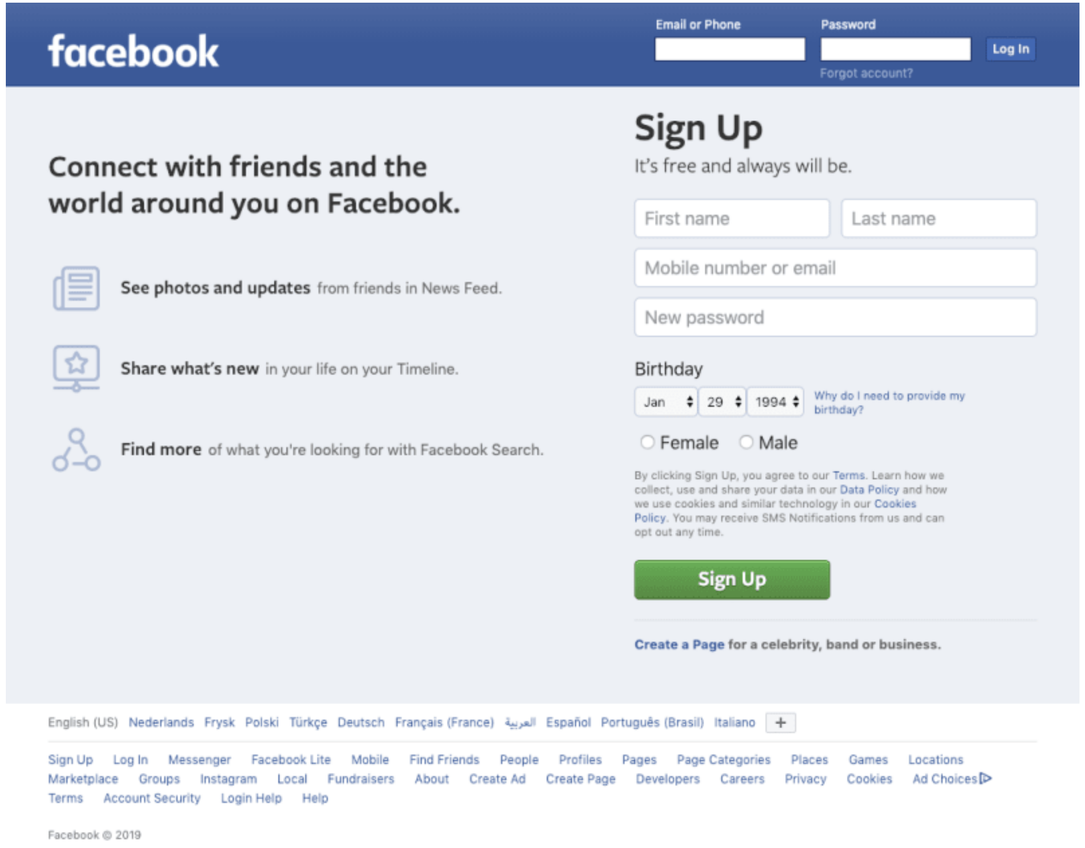
This === true will allow you to convert "true" and "false" into a boolean .

element.dataset on MDN

DOM Forms

A <form> element does not show up on the page, so why do we have to use it?

Let's take a look at this example:



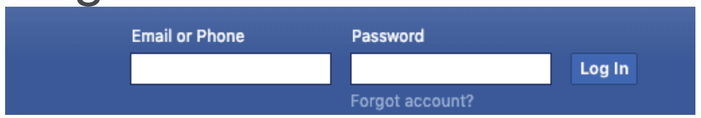
How many forms do you see?

One or two?

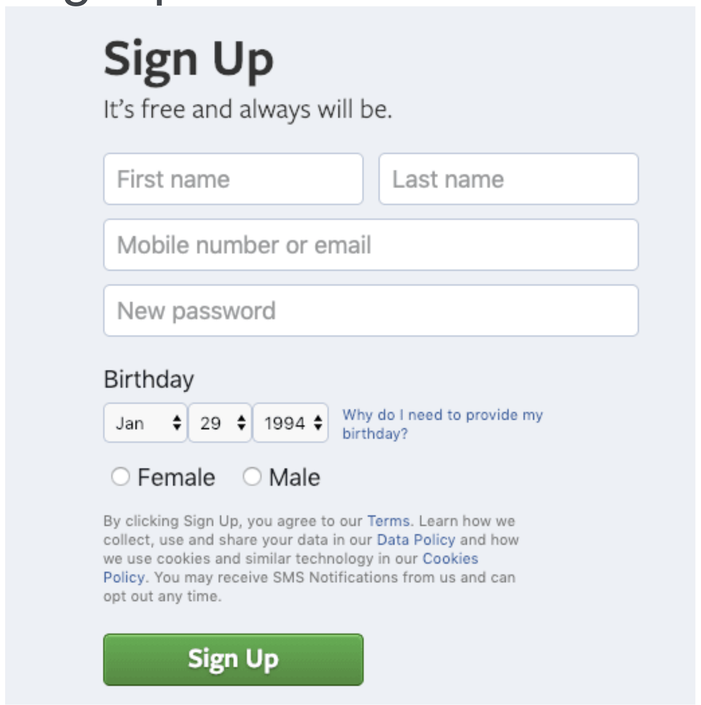
DOM Forms

The answer is two:

1. Login form:



2. Signup form:



The reason why we need forms is to be able to group several inputs together based on their action. The email & password together are responsible for the login. Whereas the first name, last name, password, birthday & gender are responsible for creating an account. We use forms, to be able to group inputs together while only sending the data of that form when it gets submitted.

How do you submit a form?

You can send the data in a form by either clicking on the button or by pressing the Enter key on the keyboard while having the focus inside one of the textboxes.

This is what we call the **submit** event.

This behavior offers a good user experience because tech-savvy users as well as users who rely on accessibility software are used to navigating your website using the keyboard. So they can write their email and password and then hit *Enter*.

While other users can use the mouse to click on the **Sign Up** button to submit their data.

Both of these actions will trigger the submit event on the form which you can listen to.

Requirements for the “submit” event

You'll most likely never have to think about it, but here are the requirements for the **submit** behavior (and event):

1. have a form element
2. have at least 1 input or textarea inside of the form
3. have a button with type="submit"

The button could either be an `<input type="submit" value="...">` or a `<button type="submit">...</button>`.

So here's an example of a form that triggers the submit event:

```
<form id="address-form">
    <input type="text" placeholder="Enter your address">
    <input type="submit" value="Save">
</form>
```

Listening for the submit event

Listening for the submit event is similar to listening to the `click` event, except that the `submit` event fires **only** on the `<form>` element.

So using the HTML code above, here's how you'd listen to the submit event:

```
const form = document.querySelector("#address-form");

form.addEventListener("submit", event => {
    // event callback (when the form is submitted)
});
```

So, it's exactly like what you've learned in the previous chapter, the only 3 things to note are:

1. the `addEventListener` has to be on the `<form>`
2. the event type `submit` instead of `click`
3. we will use the `event` details so make sure to pass it as the first parameter of the callback

Prevent default

When you submit a form, the browser will take all the values your user has written and **send** them to the backend of your website. However, this causes the whole page to be reloaded. That's because the browser will send the data to the same URL by default unless you specify the `form`'s `action` attribute.
Let's take an example:

If you have the `index.html` open and you fill in your email & password and click submit, the browser will send your email and password to the `index.html` page which will make the page refresh/reload.

This is the default behavior for forms. The reason behind it is rather historical. Before the **fetch** API (and its recent predecessor) made it to browsers, this was the only way of submitting data from the frontend to the backend.

However, nowadays we've got the `fetch` API, and thus we don't want to *reload* the page every time the user sends some information.
That's why we have to prevent the default behavior of the submit event by calling the `preventDefault` method on the `event` details:

```
form.addEventListener("submit", event => {
    event.preventDefault();
    // the form will not reload anymore
});
```

Prevent default

Don't forget the `()` because `preventDefault` is a method. Also, don't forget to capitalize the `D` in `preventDefault`.

As long as you add a `submit` event listener to a form, then you will need to prevent the default, or else your code will not run as the page will reload.

Note that in the browser console, you still have to prevent the default action in your code to pass the tests.

 **event.preventDefault()** on MDN

Why is this default behavior still like this?

You might be wondering why is the default behavior still the same? Can't browsers just fix it?

Unfortunately no, browsers cannot change the default behavior because this will break so many websites.

This is what's awesome about the web. It's backward compatible, meaning that it strives to keep all previous pages working even if they were written 20 years ago.

Also, sometimes you *do* want to rely on this behavior if you're not using `fetch`. That would be the case if your backend is written in Rails/Laravel/Symfony/etc. and you'd like to submit the data without using `fetch`.

User input

Now that you know how to listen to the submit event and prevent the default action on the form submit, you can start reading what the user has written inside the form.

```
<form id="weather-form">
    <input type="text" id="city" placeholder="Enter your
city">
    <input type="submit" value="Show weather">
</form>
```

You've got this form, and you'd like to read the city that the user has written when they submit the form. How do you do that?

User input

Let's start by adding an event listener on submit and preventing the default:

```
const form = document.querySelector("#weather-form");

form.addEventListener("submit", event => {
    event.preventDefault();

    // TODO:
    // read the user's city
    // pass this city to getWeatherInfo
    // getWeatherInfo(userCity);
});
```

`getWeatherInfo()` is a hypothetical function in this example that takes a `city` argument so that it can **fetch** the weather info for that city.

inputElement.value

To be able to read the user's input, we have to access the `value` property as explained in previous chapters:

```
const form = document.querySelector("#weather-form");
const city = document.querySelector("#city");

form.addEventListener("submit", event => {
    event.preventDefault();

    // read the user's city and pass it to getWeatherInfo()
    console.log(city.value); // see in the console to make sure it's working
    getWeatherInfo(city.value);
});
```

Technically, there's nothing new in this code, we're just combining what we learned. We're reading the `value` property on the **city** element.
However, you **have to make sure** that you access `value` inside the **submit event**. Otherwise, the `value` will return an empty string (`""`).

The reason for that is that you want to read what the user has written the moment they submit the form, not when they load the page.

inputElement.value

So this example will **NOT** work:

```
const form = document.querySelector("#weather-form");
// this will NOT work ❌
// city will be an empty string because it's empty when the page loads
const city = document.querySelector("#city").value; // ❌

form.addEventListener("submit", event => {
    event.preventDefault();

    //city is still empty because we read its value before "submit"
    console.log(city); // ""
});
```

Some developers assume that JavaScript will automatically re-read the `value` on the `#city` input. But, that's not the case. This is not a limitation of JavaScript.

So, always make sure to access the `value` property inside the *submit* event to get the value when the user submitted the form.

Chapter recap

- A `<form>` element groups several inputs together and separates multiple forms on a web page.
- The `submit` event is when the user clicks on the submit button *or* presses the **enter** key inside the form.
- The `submit` event fires on the `<form>` element (not on the button).
- By default, the browser will send the data to the current page. To avoid that, you need to prevent the default action with `event.preventDefault()`.
- Make sure to access the `value` property inside the submit event. Otherwise, it'll be an empty string (or the value that is pre-filled inside the textbox)

DOM advanced events

Now that you've learned the most common concepts surrounding DOM Events, we'd like to mention 2 more concepts:

element.removeEventListener(eventType, callback)

Similar to how you can add an event listener, you can remove an existing event listener on an element using the `element.removeEventListener(eventType, callback)` method.

For it to work, however, you have to provide both arguments. So, the `callback` needs to be the same as the one you provided in the `addEventListener` call. This is why, we'll need to define the function and give it a name, then use that function as the `callback`. Let's take a look at an example:

```
const button = document.querySelector("button");

button.addEventListener("click", () => {
    console.log("button clicked");
});
```

With the code above, we *cannot* remove the event listener because if we try and call `button.removeEventListener`, we need to provide the same callback as above.

DOM advanced events

So, we'll have to write the callback as a function and give it a name instead of an anonymous function:

```
const button = document.querySelector("button");

const handleClick = () => {
    console.log("button clicked");
}

button.addEventListener("click", handleClick);
```

In the code above, we extracted the callback into a named function called `handleClick`. Before we continue, note this line `button.addEventListener("click", handleClick);`. You should **not** add the `()` here because you do not want to call `handleClick` right now, but, instead, want to tell JavaScript which function you're referring to.

If you do use the `()` (`button.addEventListener("click", handleClick());// ❌`, the function `handleClick` will be called on page load and the event listener will not work.

Once event listener

If you need to add an event listener that only runs once, there's an easier way instead of adding an event listener and then removing it. You can add an event listener and specify `once: true` in its options.

```
const button = document.querySelector("button");

button.addEventListener("click", () => {
    console.log("button clicked");
}, {
    once: true
});
```

Notice the 3rd argument for `addEventListener` which is an object. `{once: true}` means that the event should execute only once and then it gets automatically removed.

After the user clicks on the button the first time, the event listener will be automatically removed by the browser.

Other events

We've learned about the `click` and `submit` events. They are by far the most common events.

The `click` event also works on mobile (as long as you have the `<meta name="viewport">` defined with a content such as `"width=device-width,initial-scale=1"`).

We'll have visualization challenges for most of the events below so that you can see how they work!

Let's take a look at some other events:

focus/blur

The focus and blur events are often used in form validation. They let you know when a user *focuses* (put the cursor inside of it) on a textbox and when they remove the focus (blur).

The word focus means that the element is selected to receive user input from the keyboard. If you write something on your keyboard, it will be written inside the element that is focused.
When you remove the focus, then this will dispatch a blur event.

You can listen to both of these events on text boxes. Let's see an example:

```
<input type="text" id="name" placeholder="Enter your name">
const name = document.querySelector("#name");

name.addEventListener("focus", () => {
    console.log("user focused inside the name");
});

name.addEventListener("blur", () => {
    console.log("user removed focus from the name");
});
```

Focus event on MDN **Blur event** on MDN

DOMContentLoaded

This event fires on the `document` element only. It signifies that the HTML has been loaded successfully by the browser.
This means that the browser has finished reading all of the content of your HTML file. It doesn't mean however that images and other assets have finished loading.

```
document.addEventListener("DOMContentLoaded", () => {
    console.log("DOM is ready");
});
```

This event used to be quite popular a few years ago, but nowadays, you can place your `<script>` at the end of the page (right before the closing tag of the body) and you won't have to worry about waiting until the DOM is ready.
However, you may encounter it in some online forums.

DOMContentLoaded event on MDN

Scroll

The scroll event triggers on any element that scrolls. It is often used on the window object as following:

```
window.addEventListener("scroll", () => {
    console.log("page scrolled");
});
```

However, adding a scroll event will most likely slow down your page. Its usage is discouraged as it makes scrolling slow, especially for scroll-based animations. You may be able to use the scroll event performantly if you debounce the event.

Debouncing the event means you listen to changes much less than usual to preserve resources. You can get a debounce function from the Internet or from popular libraries such as lodash.

Scroll-based animations will be possible in a performant way in the future once CSS Houdini becomes stable and supported by all browsers.

Scroll event on MDN

change

The `change` event is often used on the `<select>` element. It lets you know when the user has selected a new choice.

```
<select id="countries">
    <option value="">Select a country</option>
    <option value="NL">Netherlands</option>
    <option value="BR">Brazil</option>
</select>
const countries = document.querySelector("#countries");

countries.addEventListener("change", () => {
    console.log(countries.value);
});
```

The `change` event will only trigger when the user chooses a new value. So, since by default we see `Select a country`, the event will only trigger once the user chooses a new entry (for example `Netherlands` or `Brazil`).

Once they've chosen `Netherlands`, the event will only trigger again once they've chosen a new value such as `Select a country` or `Brazil`.

Change event on MDN

Keydown/keyup

The keydown and keyup events are used to know when the user has typed a character on the keyboard. These can be used to implement keyboard shortcuts.

The only difference between keydown and keyup is that keydown triggers while the user starts pressing the button and before the character is being typed. On the other hand, keyup fires after the character has been typed.

For most scenarios, you end up needing keyup. These events can be either added to the document (to know when a user has pressed a key anywhere on the page) or inside a textbox.

```
document.addEventListener("keydown", event => {
    console.log(event.key);
});

document.addEventListener("keyup", event => {
    console.log(event.key);
});
```

Notice that we can know which character was pressed by reading the event.key.

Keyup event on MDN **Keydown event** on MDN

Chapter recap

- You can remove an event listener using the `element.removeEventListener(eventType, callback)` method.
- If you need to add an event listener that only runs once, there's an easier way instead of adding an event listener and then removing it. You can add an event listener and specify `once: true` in its options.

- `focus` is triggered when the user enters focus (the cursor) in a textbox.
- `blur` is triggered when the user removes focus (the cursor) from a textbox.
- `DOMContentLoaded` is fired when the browser has finished loading & constructing the entire HTML on your page.
- `scroll` is triggered every time the user scrolls.
- `change` is used to know when a `<select>` has a new option chosen.
- `keydown` and `keyup` are used to know when the user has typed a character on the keyboard.

Practice / Classroom coding

- **visualize01**: Visualize blur/focus
- **visualize02**: Visualize scroll
- **visualize03**: Visualize change
- **visualize04**: Visualize keyup

Practice / Classroom coding

- **dom01.js** : Complete the `renderShoppingList` function such that it renders an `<li>` element for every item in the `items` array it receives. Also, the order of the items should be the same as the one in the array. So, the fi rst item should show up fi rst (at the top).

- **dom02.js** : Complete the `addItemToShoppingList` function such that it adds the (single) item it receives to the element with id `shopping-list` as a new `<li>` element. Every time this function is called, it should add a new item to the existing list.

- **dom03.js** : Every time you click on the `Add` button, the `addItemToShoppingList` function is called and it receives the text inside the `textbox` .

- **dom04.js** : Complete the `getUserIdFromCard` function such that it returns the value of `data-user-id` (number) from the `user-card` element.

- **dom05.js** : Complete the `getIsActiveFromCard` function such that it returns the value of `data-is-active` (boolean) from the `user-card` element.

Introduction to Object Oriented Programming

Like every new concept explained, we're going to start with the question: **Why do we need Classes?**
Let's take a look at the problem that classes solve.

Let's say we are creating functions that describe a **User** of our application. For example, the user has a `firstName`, a `lastName`, and an `age`. These are the **variables** representing the user. But we also have some **functions** that correspond to every user in our application. That is the functions `getFullName()`, `getInitials()`, and `canVote()`. Assuming these functions exist, here's how we use them:

```
let firstName = "Sam";
let lastName = "Blue";
let age = 30;

getFullName(firstName, lastName); // "Sam Blue"
getInitials(firstName, lastName); // "SB"
canVote(age); // true
```

Now compare that to the code below, which assumes that we have a **class** called User de fi ned somewhere:

```
let sam = new User("Sam", "Blue", 30);
sam.getFullName(); // "Sam Blue"
sam.getInitials(); // "SB"
sam.canVote(); // true
```

Introduction to Object Oriented Programming

Notice how the second example is much more expressive. If we were to read it in plain English, we'd say:

- We start by creating a **new user** with the first name of "Sam", the last name of "Blue" and the age of 30.
- Then we get the full name of that `user` by calling `user.getFullName()`.
- Then we get the initials of that `user` by calling `user.getInitials()`.
- Finally, we check if that `user` can vote by calling `user.canVote()`.

In the code above, there's a `class` called `User` that has been defined somewhere. And this class `User` wraps and contains all the functions and variables that describe a user in our application.

So what is a Class?

So why do we use classes? There are many benefi ts for using classes, but for now, a class allows us to group together all the variables and functions describing an entity in our application (for example a user, a person, an employee, a recipe, etc.).

Objects allowed us to group several variables into one object, so how is a class different than an object then ?

an object is only a representation of variables, whereas a class also defi nes the behavior because we can have functions related to that entity.

For example, an object user will contain key/value pairs describing a user. Whereas a class User will contain variables and functions describing a user.

If we look at the two code samples above, we can see how we moved from having functions such as getFirstName() and canVote() that were quite generic, into user.getFirstName() and user.canVote(). It became clear that these functions are only called for a **user** .

Methods & Properties

A quick note on naming: we said that a class groups variables and functions together. Once you start working with classes, the variables inside a class are called **properties** and the functions inside a class are called **methods** .

Using the same class **User** from the previous lesson, let's take a look at how we can create two different users:

```
let sam = new User("Sam", "Blue", 30);
sam.getFullName(); // "Sam Blue"
sam.getInitials(); // "SB"
sam.canVote(); // true
```

```
let charley = new User( "Charley", "Don", 17) ;
charley.getFullName(); // "Charley Don"
charley.getInitials(); // "CD"
charley.canVote(); // false
```

Notice how sam and charley are two different variables.
The fi rst one is the result of new User("Sam", "Blue", 30) and the second one is the result of new User("Charley", "Don", 17).
They are both **instantiated** (created) from the same class **User**, but they both have different properties (variables). Also, calling methods (functions) on these variables gives us different results depending on which variable we're calling it on.

For example. sam.canVote() returns true (because Sam's age is 30; above 18)
whereas charley.canVote() returns false (because Charley's age is 17; under 18).

The ‘new’ keyword

If you look at the fi rst line in the code above, let sam = new User("Sam", "Blue", 30), you will notice a new keyword which is: **new** .

Because User is a class, you can create a **new instance of that class** with the new keyword.

What's an instance ?

To be able to answer this question, let's take a look at the output of console.log(sam):

```
User {
    firstName: "Sam",
    lastName: "Blue",
    age: 30,
    getFullName: function() {...},
    getInitials: function() {...},
    canVote: function() {...}
}
```

Note: that the output above is slightly simplifi ed.

You can see that sam is an **object** . We can see the word **User** before that object in the console because this is not *any* kind of object, but it's a specifi c object. It's an object created by the User class.

So the result of new User(...) will always be an **object** .

We will further expand this topic in the next lesson!

What you have to know, for now, is that when you create a new instance of a class, you will get back an object.

The ‘new’ keyword

So what is a class then? A class is a factory that creates objects.
A class is a **blueprint** for creating objects. (A blueprint is the planning of something, for example, a building).

The `class User` is the blueprint for creating user objects. One of the most important concepts in classes is to understand the difference between a **class** and an **instance**.

A **class** is a factory that is able to make **instances**.

Assuming a class `Person` that accepts one parameter (the full name), you can create several instances:

```
let person1 = new Person("Sam Doe") ;
let person2 = new Person("Charley Bron");
```

The `person1` and `person2` variables are **instances** of the class `Person`.
So the class `Person` is the template that we use to create the `person1` and `person2` objects which are called instances.

Every instance is different

Another important concept is that every instance we create is different. For example, using the code above, if we compare person1 and person2:

```
person1 === person2; // false (they are not the same)
```

we get false because these are 2 different instances (but from the same class).

You might be wondering whether the 2 instances below will be the same or not:

```
let sam1 = new Person( "Sam Doe") ;
let sam2 = new Person( "Sam Doe”) ;
```

because sam1 and sam2 were both instantiated from the same class and with the same parameter.

In fact, sam1 === sam2 will also give you false. That's because every instance from a class is a completely new and different object. Even if the objects have the same values inside.

Define your own class

Let's say we'd like to create a class that represents a User, then here's how you defi ne that class:

```
class User {

}
```

Notice how we use the class keyword followed by the name of the class (in this example **User**) and then we have the opening and closing curly brace which together defi ne the start and the end of the class. Later in the next 2 chapters, we will put the methods in between those two curly braces.

Naming conventio n

It's important to name the class in **UpperCamelCase**. So the fi rst character of every word should be in upper case, the rest in lower case.
Here are some examples:

- A class representing a **recipe** should be called Recipe.
- A class representing a **quick recipe** should be called QuickRecipe.
- A class representing a **yearly result** should be called YearlyResult.
-

Class definition vs Class usage

Similar to function de fi nitions, we separated between the function de fi nition and the function usage. We have the same concept in classes.
The class is de fi ned once but can be used more than once. We will separate between the two with the comments: // class de fi nition and // class usage :

```
// class definition
class User {

}

//    class      usage
let user1 = new User();
let user2 = new User();
```

Class constructor

When you defi ne a class, you can create a function inside this class called **constructor** which will be automatically called whenever you create a new **instance** of this class.

Here's how you defi ne it:

```
// class definition
class User {
    constructor() {
        console.log( "creating instance") ;
    }
}
```

We added in this example a console.log() inside the constructor() method.

Now, whenever we create a new instance with the new keyword, we will see **creating instance user** in the console:

```
// class usage
let user1 = new User(); // "creating instance" will be logged to the console
let user2 = new User(); // "creating instance" will be logged to the console
```

So this function runs as soon as you create a **new** instance. Throughout the **class** chapters, we will use this constructor() method to **set up** the instances that we're creating. This will become clearer throughout the next chapter.

Syntax

Let's take a look at the syntax of the constructor method:

```
class User {
    constructor() {
        // code here
    }
}
```

Notice how the constructor() goes **inside** the curly braces of class User { ... } . This is because the constructor method is part of the class User.

Also, notice how there is no function keyword. So functions defi ned inside a class (which are called methods), do not use the function keyword. They directly go inside the class and take the parentheses after their name and the curly braces.

Chapter recap

- The variables inside a class are called **properties** and the functions inside a class are called **methods** .
- You can create a **new instance of a class** with the `new` keyword.
- When you create a new instance of a class, you will get back an object.
- A **class** is a factory that is able to make **instances**.
- Every instance created from the same class is different.
- A class is a factory (or a blueprint) that creates an object. This object will contain properties and methods that describe an entity of your application.
- The class name should be in **UpperCamelCase**.
- Here's how you defi ne a class User: `class User { }`
- The `constructor()` method inside a class is automatically called every time you create a new instance of this class (when you use the `new` keyword followed by the class name).
- The `constructor()` method goes inside the class defi nition because it's part of the class.
- The syntax of writing the `constructor()` is the name of the method, followed by parentheses and curly braces. No `function` keyword.

Chapter recap

- A class allows us to group together all the variables and functions describing an entity in our application.
- The variables inside a class are called **properties** and the functions inside a class are called **methods** .
- You can create a **new instance of a class** with the `new` keyword.
- When you create a new instance of a class, you will get back an object.
- A class is a factory that creates objects.
- A class is a **blueprint** for creating objects.
- A **class** is a factory that is able to make **instances**.
- Every instance created from the same class is different.
- A class is a factory (or a blueprint) that creates an object. This object will contain properties and methods that describe an entity of your application.
- The class name should be in **UpperCamelCase**.
- Here's how you defi ne a class User: `class User { }`
- A class allows us to group together all the variables and functions describing an entity in our application.

Practice / Classroom coding

- **class1.js**: We have defi ned an empty class `User` in `class1.js`. Create a new variable called `user` and assign it to a new instance of the class `User`. The class `User` is empty for now, so it doesn't expect any arguments. Feel free to `console.log()` the new instance so that you can visualize the returned object in the console.

- **class2.js**: Defi ne a class (empty for now) that represents a **recipe**. Then create a new variable called `recipe` and assign it to a new instance of that class.

- **class3.js**: Write the `Recipe` class such that it automatically console logs the string: **New recipe created** whenever we create a new instance of that class.

- **class4.js**: Write the `Recipe` class such that it logs the **name** and the **calories** every time a new instance of the class is being created.

Supermini Project 06

Complete the class **NameVariations** such that it contains 3 instance methods:

- getNumberOfChars which returns the number of characters for the name instance variable
- getLower which returns the name instance variable in lower case
- getUpper which returns the name instance variable in upper case

The class **NameVariations** is instantiated with the name.

Supermini Project 07

Complete the class Tasks with the following instance methods:

- importCsv which receives a CSV string, converts it into an array of tasks and stores it into the instance variable this.tasks.
- getCount which returns the number of tasks.
- getFirst which returns the first task.
- getLast which returns the last task.
- getUnformattedTasks which returns a string of all the tasks lower-cased and separated by a comma character and a space character. (CSV export)

Supermini Project 08

Complete the class Passport such that it includes the following instance methods:

- getFirstName which returns the first name in lower case.
- getLastName which returns the last name in upper case.
- getFullName which returns the first name and last name separated by a space character.
- getInitials which returns the first character of the first name followed by a dot character (.), followed by the first character of the last name and followed by a dot character (.).
- getIsValidName which returns **"Yes"** (string) when the first name is at least 1 character long and the last name is at least 1 character long and the last name does **NOT** end with a dot character (.). In all other cases, it should return **"No"**.

Note: the class is initialized with 2 arguments: the first name and the last name.