Session 1

Mohamed Emary

March 3, 2024

1 Introduction

In web development we use three main technologies: **HTML**, **CSS**, and **JavaScript**.

HTML defines the structure of your website. The basic building blocks of HTML are elements, which are created using tags.

CSS defines the style of your website.

JavaScript defines the behavior of your website.

HTML & CSS are not programming languages, they are markup languages, but JavaScript is a programming language.

Visual Studio Code is a text editor developed by Microsoft for Windows, Linux and macOS. It includes support for debugging, embedded Git control, syntax highlighting, intelligent code completion, snippets, and code refactoring.

To see your HTML, CSS, and JavaScript code in action, you need to open them in a **web browser**.

Tim Berners-Lee invented the World Wide Web in 1989.

2 Some HTML Tags and Their Syntax

HTML has paired tags and self-closing tags.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    <h1>This is a Nested Tag</h1>
    <img src="image.jpg" alt="Image" />
  </body>
</html>
```

`<html>` is the parent of all other elements (all other elements are children of it), `<head>` and `<body>` are siblings.

To save use `ctrl + s`, but it's better to activate `autosave` in the settings.

2.1 SEO (Search Engine Optimization)

SEO is Search Engine Optimization, it's used to rank your website higher in search engines.

We should stick to some rules when making our websites so that search engines rank them higher.

One of these rules is that one HTML page should never have more than one `h1` tag.

2.2 Image Tag

`<img>` is a self-closing tag, it doesn't have a closing tag.

`src` is the source of the image, and `alt` is the alternative text for the image.

`alt` is used when the image can't be displayed, and it's used by screen readers to describe the image to the user. It's also important for SEO.

Google `<img>` tag SEO guidelines.

2.3 Anchor Tag

`<a>` is the anchor tag, it's used to create hyperlinks.

`href` is the hyperlink reference, it's the URL of the page you want to link to.

`target="_blank"` is used to open the link in a new tab. For security reasons it's used with `rel="noopener noreferrer"` to prevent the new tab from accessing the `window.opener` object.

2.4 Lists

We have two types of lists in HTML: ordered lists and unordered lists.

2.4.1 Ordered List

Ordered lists has an attribute called `type` which can be `1`, `A`, `a`, `I`, or `i`.

```
<ol>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ol>
```

2.4.2 Unordered List

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```

2.5 Tables

Tables in HTML are created using the `<table>` tag and have three main elements: `<thead>`, `<tbody>`, and `<tfoot>`.

```
<table>
  <thead>
    <tr>
      <th>Header 1</th>
      <th>Header 2</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Row 1, Cell 1</td>
      <td>Row 1, Cell 2</td>
    </tr>
    <tr>
      <td>Row 2, Cell 1</td>
      <td>Row 2, Cell 2</td>
    </tr>
  </tbody>
  <tfoot>
    <tr>
      <td>Footer 1</td>
      <td>Footer 2</td>
    </tr>
  </tfoot>
</table>
```

Table element has a `border` attribute which is used to specify the width of the border around the table, for example `border="1"`.

You can control the width or height of the table using the `width` and `height` attributes.

You can make the table take the full width of the page using the `width` attribute and setting it to `100%`.

You can make a cell span multiple columns using the `colspan` attribute, and you can make a cell span multiple rows using the `rowspan` attribute.

To give the table a background color, you can use the `style` attribute and set the `background-color` property. You can also set the `color` property to change the text color.

3 Summary

- **HTML** defines the structure of your website.
- **CSS** defines the style of your website.
- **JavaScript** defines the behavior of your website.
- **Visual Studio Code** is a text editor developed by Microsoft for Windows, Linux and macOS.
- **Tim Berners-Lee** invented the World Wide Web in 1989.
- **SEO** is used to rank your website higher in search engines.

3.1 Tag

Table 1: Tags Summary

Tag	Description
`<html>`	The root element of an HTML page.
`<head>`	Contains metadata about the HTML page.
`<title>`	Specifies the title of the page.
`<body>`	Contains the visible page content.
`<h1>`	Defines a level 1 heading. We have `<h1>` to `<h6>`.
`<img>`	Embeds an image into the page.
`<a>`	Creates a hyperlink.
`<ol>`	Defines an ordered list.
`<ul>`	Defines an unordered list.
`<li>`	Defines a list item.
`<table>`	Defines a table.
`<thead>`	Groups the header content in a table.
`<tbody>`	Groups the body content in a table.
`<tfoot>`	Groups the footer content in a table.
`<tr>`	Defines a row in a table.
`<th>`	Defines a header cell in a table.
`<td>`	Defines a cell in a table.

3.2 Attribute

Table 2: Attributes Summary

Attribute	Description
`src`	Specifies the source of an image.
`alt`	Specifies an alternate text for an image when the image cannot be displayed.
`href`	Specifies the URL of the page the link goes to.
`target`	Specifies where to open the linked document.

Attribute	Description
`rel`	Specifies the relationship between the current document and the linked document.
`type`	Specifies the type of numbering for list items.
`border`	Specifies the width of the table border.
`width`	Specifies the width of the table.
`height`	Specifies the height of the table.
`colspan`	Specifies the number of columns a cell should span.
`rowspan`	Specifies the number of rows a cell should span.

Session 2

Mohamed Emary

March 6, 2024

1 HTML Forms

HTML forms are used to collect user input. To make a form in HTML you need to use `<form>` tag. The user input is most often sent to a server for processing. The form tag is used to create an HTML form for user input.

Form can contain input elements like text fields and labels for these fields.

Example:

```
<form action="/action_page.php">
  <label for="fname">First name:</label><br/>
  <input type="text" id="fname" name="fname"><br/>
  <label for="lname">Last name:</label><br/>
  <input type="text" id="lname" name="lname"><br/><br/>
  <input type="submit" value="Submit">
</form>
```

As you see in the example the value in `for` in the `label` element should be exactly the same as the value in `id` of the `input` field.

Each input field should hava a `name` attribute to be able to be sent to the server.

1.1 Type Attribute

Sometimes you want the user to enter a password, an email, a phone number, a date, a color, a number, etc. HTML5 has a lot of new input types for these cases. Just use the `type` attribute to specify the type of input.

There is a lot of input types, some of them are:

1. `text` (*default*)
2. `password`
3. `email`
4. `number`
5. `date`
6. `color`

7. `url`
8. `tel`
9. `search`
10. `file`
11. `radio`
12. `checkbox`

Example:

```
<form>
  <label for="email">Email:</label><br/>
  <input type="email" id="email" name="email"><br/>
  <label for="password">Password:</label><br/>
  <input type="password" id="password" name="password"><br/><br/>
  <input type="submit" value="Submit">
</form>
```

Each input element should have a `name`, `type` attributes.

You can also add the `value` attribute to set the default value of the input field.

The submit button is used to send the form to the server and it has a `submit` type.

The `<br/>` used in the example above is used to add a line break between the input fields.

To allow uploading files you can use the `file` type, and to allow multiple files you can use the `multiple` attribute.

You can also allow only a specific type of files using the `accept` attribute.

Example:

```
<form>
  <label for="file">Select files:</label><br/>
  <input type="file" id="file" name="file" multiple accept=".png"><br/>
  <input type="submit">
</form>
```

To allow any type of images you can use `accept="image/*"`. See this link for more information.

1.2 Button

The `button` element has a `type` attribute that can be `submit`, `reset`, or `button`.

- `submit`: The button submits the form data to the server. (This is the *default*)
- `reset`: The button resets all the form data to its initial values.
- `button`: The button does nothing.

You can also use a regular `input` tag instead of `button` and set the `type` attribute to `submit`, `reset`, or `button`, but it's better to use the `button` tag.

1.3 Radio Buttons & Checkboxes

Another type of input we can use is `radio`. One important thing about `radio` is that all the radio buttons in a group should have the same `name` attribute. You should also add `value` attribute to each radio button.

When you add a label for each radio button, you can click on the label to select the radio button.

Example:

```
<form>
  <fieldset>
    <legend>Select your favorite fruit:</legend>

    <input type="radio" id="apple" name="fruit" value="apple">
    <label for="apple">Apple</label><br>

    <input type="radio" id="banana" name="fruit" value="banana">
    <label for="banana">Banana</label><br>

    <input type="radio" id="cherry" name="fruit" value="cherry">
    <label for="cherry">Cherry</label>
  </fieldset>
</form>
```

The same applies for checkboxes except that you can select multiple checkboxes at the same time.

1.4 Textarea

The `textarea` tag is used to create a multi-line text input. The `rows` and `cols` attributes are used to specify the size of the textarea but the user can still resize it using the mouse by dragging the bottom right corner of the textarea.

Example:

```
<form>
  <label for="message">Message:</label><br/>
  <textarea id="message" name="message" rows="4" cols="30"></textarea><br/>
  <input type="submit">
</form>
```

1.5 Select Option

The `select` tag is used to create a drop-down list and the `option` tag is used to define the options in the list. If you don't give a `value` attribute to the `option` tag, the value will be the text inside the `option` tag.

You can also group the options using the `optgroup` tag.

When using `select` with `option` you can only choose what is available in the list (you can't enter a value that is not in the list).

Example:

```
<form>
  <label for="cars">Choose a car:</label><br/>
  <select id="cars" name="cars">
    <optgroup label="Swedish Cars">
      <option value="volvo">Volvo</option>
      <option value="saab">Saab</option>
    </optgroup>
```

```
    <optgroup label="German Cars">
      <option value="mercedes">Mercedes</option>
      <option value="audi">Audi</option>
    </optgroup>
  </select><br/>
  <input type="submit">
</form>
```

1.6 Datalist

The `datalist` tag is used to create a list of options for the `input` tag. The `list` should take the `id` of the `datalist` tag.

It also allows the user to enter a value that is not in the list, But you can't group the options like in the `select` tag.

Example:

```
<form>
  <label for="browser">Choose a browser:</label><br/>
  <input list="browsers" id="browser" name="browser">
  <datalist id="browsers">
    <option>Chrome</option>
    <option>Firefox</option>
    <option>Internet Explorer</option>
    <option>Opera</option>
  </datalist><br/>
  <input type="submit">
</form>
```

1.7 Input Validation

In text fields you can use `maxlength` and `minlength` attributes to specify the maximum and minimum number of characters allowed in the input field.

In numbers fields you can use `max` and `min` attributes to specify the maximum and minimum values allowed in the input field.

You can also use the `required` attribute to make the input field required, so the user can't submit the form without filling this field.

In `select` if the user didn't choose any option, the first option will be sent to the server, and to change that you can use the `selected` attribute in the `option` tag you want to be the default.

To hide an input field from the user you can use the `hidden` attribute. And to disable an input field you can use the `disabled` attribute. The disabled input fields doesn't send any data to the server however the hidden input fields do.

To prevent the user from changing the value of an input field you can use the `readonly` attribute.

Table 1: hidden, disabled, readonly attributes

Attribute	Description
`hidden`	Hides the input field from the user but sends the data to the server.
`disabled`	Disables the input field and doesn't send the data to the server.
`readonly`	Prevents the user from changing the value of the input field, and sends the data to the server.

2 Extra Information

To check wheather a tag or attribute you are using is supported by the browser you can use the Can I Use website.

To check your HTML code markup you can use W3C Markup Validation Service.

3 Summary

3.1 Tags

Table 2: Tags Summary

Tag	Description
`form`	Creates an HTML form for user input.
`input`	Defines an input field where the user can enter data.
`label`	Defines a label for an `input` element.
`button`	Defines a clickable button.
`textarea`	Defines a multi-line text input area.
`select`	Defines a drop-down list.
`option`	Defines an option in a drop-down list or a datalist.
`datalist`	Defines a list of options for an `input` element.
`optgroup`	Defines a group of related options in a drop-down list.
`fieldset`	Groups related elements in a form.
`legend`	Defines a caption for a `fieldset` element.

3.2 Attributes

Table 3: Attributes Summary

Attribute	Description
`name`	Specifies the name of an `input` element that is sent to the server.
`type`	Specifies the type of an `input` element.
`value`	Specifies a pre-defined value of an `input` element.
`placeholder`	Specifies a short hint that describes the expected value of an `input` element.
`for`	Specifies which `input` element a label is bound to.
`id`	Specifies a unique id for an `input` element.
`list`	Refers to a `datalist` element that contains some options.
`multiple`	Specifies that a user can enter more than one value in an `input` element.
`accept`	Specifies the types of files that the server accepts.
`rows`	Specifies the visible number of lines in a `textarea` element.
`cols`	Specifies the visible width of a `textarea` element.
`minlength`	Specifies the minimum number of characters allowed in an `input` element.
`maxlength`	Specifies the maximum number of characters allowed in an `input` element.
`min`	Specifies the minimum value allowed in an `input` element.
`max`	Specifies the maximum value allowed in an `input` element.
`selected`	Specifies that an option should be pre-selected when the page loads.
`required`	Specifies that an input field must be filled out before submitting the form.
`readonly`	Specifies that an input field is read-only.

Attribute	Description
`hidden`	Specifies that an input field is hidden from the user.
`disabled`	Specifies that an input field is disabled so the user can't use it.

3.3 Input Types

Table 4: Input Types Summary

Type	Description
`text`	Used to create plain text input fields.
`password`	Used to create password input fields.
`email`	Used to create email input fields.
`number`	Used to create numeric input fields.
`date`	Used to create date input fields.
`color`	Used to create color input fields.
`url`	Used to create URL (links) input fields.
`tel`	Used to create telephone number input fields.
`search`	Used to create search input fields.
`file`	Used to create file upload input fields.
`radio`	Used to create radio buttons.
`checkbox`	Used to create checkboxes.
`submit`	Used to create form submit buttons.
`reset`	Used to create form data reset buttons.
`button`	Used to create buttons that do nothing.
`range`	Used to create a range of numeric values.

Session 3

Mohamed Emary

March 10, 2024

1 Review & Questions

In the first part of the session, eng. Shimaa reviewed the previous sessions and asked us some questions to make sure we understand the previous sessions well.

2 CSS

CSS stands for *Cascading Style Sheets*. It is a style language used for describing the look and formatting of a document written in HTML. The first version of CSS was introduced in 1996.

2.1 General look of a CSS *Rule*

```
selector {
  property: value; /* declaration */
}
```

2.2 Where should CSS code be? & How to link it?

CSS code can be placed in three different locations:

- **Inline** in the HTML element inside the `style` attribute.
- **Internal** in the head section of the HTML document in a separate `<style>` tag.
- **External** in a separate file linked to the HTML document using the `<link>` tag.

2.2.1 Inline CSS

You can apply CSS to an HTML element using the `style` attribute.

```
<p style="color: red;">This is a paragraph.</p>
```

2.2.2 Internal CSS

You can write CSS inside the head section of the HTML document using the `<style>` tag inside the `<head>` element of the HTML document.

```
<head>
  <style>
    p {
      color: red;
    }
  </style>
</head>
```

2.2.3 External CSS file

You can link an external CSS file to an HTML document using the `<link>` tag inside the `<head>` element of the HTML document.

```
<head>
  <link rel="stylesheet" href="css/style.css">
</head>
```

In the example above `css/style.css` is the path of the CSS file in your project. You should create a folder called `css` and put the CSS file inside it with a name of your choice for example `style.css`.

2.3 Why to separate CSS from HTML?

There are multiple reasons to separate the CSS from the HTML:

- **Maintainability**: It is easier to maintain and update the code when the HTML and CSS are separated.
- **Reusability**: You can use the same CSS file for multiple HTML files.
- **Performance**: The browser can cache the CSS file and use it for multiple pages.

2.4 If we have different styles for the same element, what would happen?

If you have different styles for the same element, the style that is defined last will be applied.

```
<head>
  <style>
    p {
      color: red;
    }
  </style>

  <!-- In this linked file, the color of p is blue -->
  <link rel="stylesheet" href="css/style.css">
</head>
```

In the example above the color of the paragraph will be blue not red, because the linked file is defined after the internal style.

2.5 Selectors

Selectors are used to select the HTML elements that you want to style only.

2.5.1 Tag

To select an HTML element, you can use the tag name of that element, for example to select the paragraph tag you should use `p` in the CSS file.

Example:

HTML:

```
<p>This is a paragraph.</p>
```

CSS:

```
p {
  color: red;
}
```

2.5.2 Class

To select an HTML element, you can use the class name of that element, and to use the class in the CSS file you should use a dot `.` before the class name for example if you have a class called `intro` you should use `.intro` in the CSS file.

Example:

HTML:

```
<p class="intro">This is a paragraph.</p>

<p>This is not red.</p>
```

CSS:

```
.intro {
  color: red;
}
```

In the example above, the color of the first paragraph will be red but the second paragraph will not be affected.

Some guidelines to follow when using class:

1. You can't use spaces in class names, but you can use a hyphen `-` or an underscore `_` to separate the words.
2. You should also use a descriptive name for the class to make it easier to understand the code.

To give an element multiple classes you can separate them with a space inside the same `class=""` attribute, for example `class="intro text-center"`, but you can't use the class attribute more than once in the same element.

Classes are also used to reduce code repetition so if you want to apply the same style to multiple elements you can give them the same class.

2.5.3 ID

To select an HTML element, you can also use the id of that element, and to use the id in the CSS file you should use a hash `#` before the id name for example if you have an id called `intro`

you should use `#intro` in the CSS file.

The difference between the class and the id is that the class can be used for multiple elements but the **id should be unique** in the HTML document.

Since ID should be unique, you shouldn't use the same id more than once in the same HTML document.

2.5.4 Grouping

You can group multiple elements to apply the same style to them using a `div` element.

Example:

HTML:

```
<div class="intro">
  <h1>This is a heading.</h1>
  <p>This is a paragraph.</p>
</div>
```

CSS:

```
.intro {
  color: red;
}
```

What if you want to apply a style to an element only if it is *inside* a specific element? (Nested Selectors)

You can use the space to select an element only if it is inside another element. For example if you want to apply a style to the paragraph only if it is inside a `div` with a class `intro` you can use `.intro p` in the CSS file.

What if you want to apply a style to an element with a specific class only? Or apply the style to an element with multiple classes?

To apply a style to a paragraph only if it has a class `intro` you can use `p.intro`, and to apply a style to an element only if it has both the classes `intro` and `center` you can use `.intro.center` without a space between the classes names.

What if you apply a style to multiple classes?

You can use a comma `,` to apply the same style to multiple classes, for example `.intro, .center` will apply the same style to the elements with the class `intro` or the class `center`.

2.6 Specificity

Specificity is the means by which browsers decide which CSS property values are the most relevant to an element and, therefore, will be applied.

The following list of selector types is by increasing specificity:

1. Universal selectors (e.g., `*`)
2. Type selectors (e.g., `h1`)
3. Class selectors (e.g., `.example`)
4. ID selectors (e.g., `#example`)

The rules above also applies when combining multiple selectors in the same rule, for example if you have a rule with a tag and a class, it will be more specific than a rule with a class only.

For a more detailed explanation on how to calculate specificity, you can check the following link.

2.7 Some Styling Properties

The default value for `height` is `auto` and for `width` is `100%`.

If you want your styling to be dynamic and responsive, you should use a relative unit like `%`, for example `width: 100%` will make the width of the element 100% of the width of its parent element.

2.8 Block & Inline Elements

2.8.1 Block-level elements

Start on a new line and take up the full width available

Example block elements are `<div>`, `<h1>` to `<h6>`, `<p>`, `<form>`, `<header>`, `<footer>`, `<section>`, and `<ul>`.

2.8.2 Inline elements

Do not start on a new line and only take up as much width as necessary

Example inline elements are `<a>`, `<span>`, `<img>`, `<label>`, `<input>`, `<strong>`, `<em>`, and `<b>`.

You can change whether an element is block or inline using the `display` property. For example, you can change a `<div>` to an inline element using `display: inline;` or an `<a>` tag to a block element using `display: block;`.

With inline elements `width` and `height` properties have no effect.

2.8.3 Inline-block

Elements are similar to inline elements, but they can still have width and height

2.9 Replaced Elements

How can `<img>` element be inline and still have width and height?

This is because the `<img>` element is **replaced inline element**.

Replaced elements can be given explicit width and height values using the `width` and `height` properties. This allows you to control the size of the element, regardless of its content.

A replaced element in HTML is an element that is replaced with another element, such as an image, a video, or an audio file. Replaced elements are not rendered in the same way as other HTML elements, and they do not have the same properties or behaviors. Replaced elements are used to embed content that cannot be created with HTML.

The most common replaced elements are:

- `<img>` : Inserts an image into the document.
- `<video>` : Inserts a video into the document.

- `<audio>` : Inserts an audio file into the document.
- `<iframe>` : Inserts a frame into the document.
- `<input>` : Inserts an input field into the document.

You can set the width and height of an `<img>` element using the `width` and `height` attributes in the HTML, or you can use CSS. Here's an example:

```
<img src="image.jpg" alt="An image" style="width: 200px; height: 200px;">
```

In this example, the image will be displayed as a `200px` by `200px` square, regardless of the actual dimensions of `image.jpg`.

3 Summary

CSS (Cascading Style Sheets), a style language used for describing the look and formatting of HTML documents. The first version was introduced in 1996.

Key points discussed in this session:

1. **CSS Rule Structure**: A CSS rule consists of a selector and a declaration block. The declaration block contains properties and their values.
2. **CSS Placement**: CSS can be placed inline, internally within the HTML document, or externally in a separate file.
3. **Separation of CSS from HTML**: This is recommended for maintainability, reusability, and performance.
4. **CSS Selectors**: These are used to select HTML elements to style. They can be based on tag names, class names, or IDs.
5. **Specificity**: This is how browsers decide which CSS property values are the most relevant to an element and will be applied. Specificity increases from universal selectors to type selectors, class selectors, and ID selectors.
6. **Styling Properties**: The document discusses some styling properties like `height` and `width`.
7. **Block & Inline Elements**: Block-level elements start on a new line and take up the full width available, inline elements do not start on a new line and only take up as much width as necessary, and inline-block elements are similar to inline elements, but they can still have width and height.
8. **Replaced Elements**: These are elements whose appearance and dimensions are defined by an external resource, such as an image, video, or audio file.

Session 4

Mohamed Emary

March 10, 2024

1 Review & Questions

In the first part of the session, eng. Shimaa reviewed the previous sessions and asked some questions.

2 Cont. CSS

2.1 Float & Clear

The `float` and `clear` section is from W3Schools So you better read it from there.

2.1.1 Float

As we know from the excercise of the previous session, one of the problems with inline-block is the extra space between the elements, and to solve this issue, a possible solution is to use `float`.

`float` is a CSS property that allows an element to be taken out of the normal flow and placed along the left or right side of its container.

The `float` property is used for positioning and formatting content e.g. let an image float left to the text in a container.

The `float` property can have one of the following values:

- `left` or `right` - The element floats to the left or right of its container
- `none` - The element does not float (will be displayed just where it occurs in the text). This is default
- `inherit` - The element inherits the float value of its parent

`float` has two problems:

1. Floating elements are removed from the normal flow of the document, One of the obvious downsides of this is that the parent element no longer contains the floated element. For example if the container has a background color, it will not expand to contain the floating element.

2. The last floating element have to be cleared, otherwise it will affect the layout of the next element.

2.1.2 Clear

The `clear` property specifies on which sides of the cleared element no elements are allowed to float, it specifies what should happen with the element that is next to a floating element.

The `clear` property can have one of the following values:

- `none` (*default*) - The element is not pushed below left or right floated elements.
- `left` or `right` - The element is pushed below left or right floated elements
- `both` - The element is pushed below both left and right floated elements
- `inherit` - The element inherits the clear value from its parent

When clearing floats, you should match the clear to the float: If an element is floated to the left, then you should clear to the left.

2.2 Margin & Padding

`margin` is the space **outside** the border of an element. It is used to create space between the **element and the surrounding elements**.

`padding` is the space **inside** the border of an element. It is used to create space between the **element's border and the content**.

To give margin to an element use `margin` property in the form `margin: top right bottom left;` in clockwise order.

You can also use `margin-top`, `margin-right`, `margin-bottom`, and `margin-left` properties to specify the margin for each side separately.

In the shorthand `margin` property, if you specify one value, it will be applied to all sides, if you specify two values, the first value will be applied to the `top` and `bottom`, and the second value will be applied to the `right` and `left`.

If you ignore `left` for example it will be the same as `right`, and if you ignore `bottom` it will be the same as `top`.

All the above also apply to the `padding` property.

Sometimes we use `margin: auto;` to **center an element horizontally** (horizontally only vertical direction will not be affected).

Using both `margin-right: auto;` and `margin-left: auto;` will also center the element.

Using `margin-right: auto;` only will push the element to the left, and using `margin-left: auto` only will push it to the right.

To center items inside a div use `text-align: center;`.

The HTML page `body` has a default `margin` of `8px`, but it can vary between browsers.

2.2.1 Margin Collapse

The top and bottom margins of blocks are sometimes combined (collapsed) into a single margin whose size is the largest of the individual margins (or just one of them, if they are equal), a

behavior known as margin collapsing.

It happens due to an empty div inside another div, and the margin of the empty div will collapse with the margin of the parent div, so you can use padding instead of margin to solve this problem, or use `overflow: auto;` in the parent div.

When we use a **percentage** in the margin or padding it will be a percentage of the **width of the parent** element. For example if you have a parent with width `500px` and you have a child with `margin-top: 50%;` the margin will be `250px`, however it's not recommended to use percentage with margin.

Box model in dev tools allows you to see the margin, padding, and borders of each element.

2.3 Dealing with Fonts in CSS

`font-size` property is used to specify the size of the font.

The default font size for paragraphs is `16px`.

`font-size` can be specified in `px`, `em`, `rem`, `vw`, `vh`, `vmin`, `vmax`, or `%`.

When we use a percentage `%` with `font-size` it's a percentage of the default font size which is `16px` for example `50%` will be `8px`.

`font-weight` ranges from 100 to 900 and default is 400. It's unitless. Some fonts don't have all the weights.

Some weights have names like `normal` which is `400`, `bold` which is `700`, `bolder` which is `900`, and `lighter` which is `100`.

`color` is used to change the color of the text. It can be a color name, a hex value, an RGB value, or an HSL value.

`font-style` specifies the style of the font. It can be `normal` (default), or `italic`.

`font-family` is used to change the shape of the font. One of the problems with this property is that if the font is not available on the user's device it will use the default font (fallback), so you can use font stack to solve this problem. You can also use fonts from google fonts or add your own font to the project files.

How to embed a font from google fonts?

1. Go to google fonts and choose the font you want.
2. Click on the + icon to add the font to the selection.
3. Click on the embed tab and copy the link tag and paste it in the head of your html file.
4. Copy the font-family and paste it in the css file.

If you choose more than one font, google fonts will give you a link tag with the two fonts in it, so you don't have to add a separate link tag for each font.

You can know that font used in a web page using the dev tools, or using a browser extension like WhatFont.

2.4 Background

It can be a color or an image. A color using `background-color` and an image using `background-image: url('path to the image');`

If the image is not important for SEO add it using CSS `background-image`, as SEO don't see CSS code but if it affects the SEO use it with the HTML `<img>` tag.

User can't save the image if it's added using CSS, but if it was added using HTML, he can save it.

`background-repeat` can be:

- `no-repeat` - The background-image will not be repeated.
- `repeat-x` - The background-image will be repeated horizontally only.
- `repeat-y` - The background-image will be repeated vertically only.
- `repeat` - The background-image will be repeated both horizontally and vertically.
- `space` - The background-image will be repeated as much as possible without clipping. The first and last images will be pinned to either side of the element, and whitespace will be distributed evenly between the images.

`background-position` can be `top`, `bottom`, `left`, `right`, `center`, or you can use the x and y coordinates.

- `background-position: center center;` this will center the image in both x and y axis.
- `background-position: top right;` this will put the image in the top right corner.
- `background-position: 50% 50%;` this will center the image in both x and y axis.
- `background-position: 50% 100%;` this will center the image in the x axis and put it in the bottom of the y axis.
- `background-position: 50px 100px;` this will put the image `50px` from the left and `100px` from the top.
- `background-position: 100%` the default value for the y axis is `50%`.

`background-size` can be specified in pixels, percentage, or using the keywords `cover`, or `contain`.

Examples:

- `background-size: 100px 100px;` this will make the image 100px by 100px, and the image original aspect ratio will not be preserved.
- `background-size: 100% 100%;` this will make the image fit the container, and the image original aspect ratio will not be preserved.

What is the difference between cover and contain?

Both `contain` and `cover` will preserve the image original aspect ratio, however:

1. `background-size: cover;` this will make the image cover the whole container, and it will not be repeated.
2. `background-size: contain;` this will make the image fit inside the container, and it can be repeated.

`background-attachment` can be `scroll` (*default*) or `fixed`.

`vh` means viewport height, and `vw` means viewport width. These units are used to make the element take a percentage of the visible area of the screen. Each 1vw or 1vh is equal to 1/100 of the viewport width or height.

3 Summary

Float & Clear

- `float` is used to position an element along the left or right side of its container.
- `clear` is used to specify on which sides of an element no elements are allowed to float.

Margin & Padding

- `margin` is the space outside the border of an element
- `padding` is the space inside the border of an element.
- `margin: auto;` can be used to center an element horizontally.

Margin Collapse

Margin collapse happens when the top and bottom margins of blocks are combined into a single margin.

Dealing with Fonts in CSS

Some important font properties in CSS include:

- `font-size`
- `font-weight`
- `color`
- `font-style`
- `font-family`

How to embed a font from Google Fonts?

- Go to google fonts and choose the font you want.
- Click on the + icon to add the font to the selection.
- Click on the embed tab and copy the link tag and paste it in the head of your html file.
- Copy the font-family and paste it in the css file.

Background

- `background-color` is used to set the background color of an element.
- `background-image` is used to set the background image of an element, and it has some important properties like:
  - `background-repeat`
  - `background-position`
  - `background-size`
  - `background-attachment`.

`vh` means viewport height, and `vw` means viewport width.

Session 5

Mohamed Emary

March 20, 2024

1 Cont. CSS

1.1 Border

`border` property is used to set the border of an element. It is a shorthand property for setting the `width`, `style`, and `color` of the border on different sides of an element, in the form `border: width style color;`.

To remove the default border, set to an element like button use `border: none;`.

`border-width`: It is used to set the width of the border. It takes values in `px`, `em`, `rem`, `%`, etc.

`border-width` can also be set individually for each side of the element using `border-top-width`, `border-right-width`, `border-bottom-width`, and `border-left-width`, or the shorthand property `border-width: top right bottom left;`.

`border-style`: It is used to set the style of the border. It can take values like:

- `solid`
- `dotted`
- `dashed`
- `double`
- `outset`
- `inset`

`border-color`: It is used to set the color of the border. It can take values like:

- `color-name`
- `#hex`

- `rgb()`
- `rgba()`
- `hsl()`
- `hsla()`
- `transparent`

`border-radius`: It is used to set the radius of the border. It can take values in `px`, `em`, `rem`, `%`, etc.

To make element look like a circle, set `border-radius` to `50%` if the element is a square (width and height are equal).

You can also set the radius individually for each corner of the element using:

- `border-top-left-radius`
- `border-top-right-radius`
- `border-bottom-left-radius`
- `border-bottom-right-radius`

1.2 CSS Sprites

One of the important things you should take in mind as a web developer is the **HTTP requests**. The more the requests, the more the time it takes to load the page which will lower the performance of the website.

So to improve the performance of the website, we can use CSS sprites. CSS sprites are a way to reduce the number of HTTP requests made for image resources, by combining images in one file.

For example, consider the following image, It has 20 icons of different colors, each of 76x76 pixels and the whole image is 384x310 pixels.

Figure 1: CSS Sprites

It has a lot of icons. Instead of loading each icon separately, we can combine all the icons in one image and use CSS to display the required icon.

```
1 .icon {
2     width: 76px;
```

```
    height: 76px;
    background-image: url('icons.png');
}

.icon-1 {
    background-position: 0 0;
}

.icon-2 {
    background-position: -76px 0;
}
```

1.3 Background Clip

`background-clip`: It is used to specify the painting area of the background. It can take values like:

- `border-box` (default) - Starts placing the background **from the border** of the element.

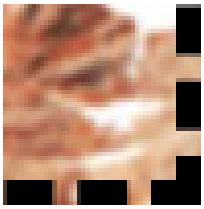

Figure 2: border-box

- `padding-box` - Starts placing the background **from the padding** of the element.

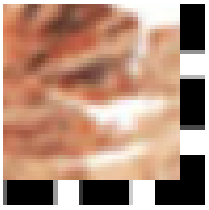

Figure 3: padding-box

- `content-box` - Starts placing the background **from the content** of the element.

Figure 4: content-box

- `text` - Makes the background to be painted within the text, but the text `color` has to be set to `transparent`. (Not supported in all browsers)

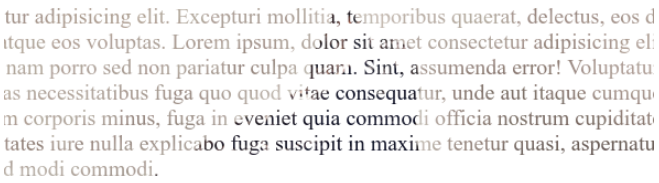

Figure 5: text

Note that `background-clip` has some compatibility issues with older browsers.

1.4 Viewport Units

Viewport units are a new set of units designed to be used in CSS for responsive design. They are relative to the viewport width and height.

Viewport is the browser window size. 1vw = 1% of viewport width, 1vh = 1% of viewport height.

1.5 Position

A CSS property that allows you to control the position of an element. It can take values like:

- `static` (default) - The element is positioned according to the normal flow of the document.
- `relative` - The element is positioned according to the normal flow of the document, and then offset relative to itself based on the values of `top`, `right`, `bottom`, and `left`.
- `absolute` - The element is removed from the normal flow of the document, and no space is created for the element in the page layout. It is positioned relative to its closest non statically positioned ancestor if any; otherwise, it is placed relative to the initial containing block.
- `fixed` - The element is removed from the normal flow of the document, and no space is created for the element in the page layout. It is positioned relative to the initial containing block established by the viewport, except when one of its ancestors has a `transform`, `perspective`, or `filter` property set to something other than `none`, in which case that ancestor behaves as the containing block.
  - With `fixed` position give the element a `width` of `100%` to make it cover the full width of the viewport.
- `sticky` - The element is treated as `relative` positioned until it crosses a specified threshold, at which point it is treated as `fixed` positioned.

When using `position: relative;` the element will be positioned relative to its normal position.

```
.relative-one {
    position: relative;
    top: 20px; /* Moves the element 20px down of its normal position */
    left: 20px; /* Moves the element 20px right of its normal position */
}

.relative-two {
    position: relative;
    top: 20px;
```

```
    bottom: 20px; /* No effect since the page flows from top to bottom */
                  /* Normally no one will use top and bottom together */
}

.relative-three {
    position: relative;
    left: 20px;
    right: 20px;  /* No effect since the page flows from left to right
    ↪  unless you change it */
                  /* Normally no one will also use right and left together
                  ↪  */
}
```

In `static` position, the `top`, `right`, `bottom`, and `left` properties have no effect.

When using `position: absolute;` the element will be positioned relative to the viewport, unless its ancestor (parent or a parent of a parent) has a `position` property set to `relative`, `absolute`, `fixed`, or `sticky`, in which case it will be positioned relative to its non statically positioned ancestor.

```
.absolute-one {
    position: absolute;
    top: 20px;  /* Moves the element 20px down of the top of the viewport
    ↪  */
    left: 20px; /* Moves the element 20px right of the left of the
    ↪  viewport */
}

.absolute-two {
    position: absolute;
    bottom: 0px;
    right: 0px;
} /* Moves the element to the bottom right of the viewport */
```

When using `position: relative` the original place of the element will be reserved even if the element is moved away from it.

The element can also span over other elements.

When using `position: absolute` the original place of the element will not be reserved.

To make an absolute child move relative to its parent give its parent a position different from `static`, otherwise it will move relative to its closest positioned ancestor and if no positioned ancestor is found it will move relative to the viewport.

When using `position: fixed` the original place of the element will not be reserved, and the element will be positioned relative to the viewport.

When using `position: sticky` the original place of the element will be reserved until it reaches a specified threshold (with scroll for example), at which point it is positioned relative to the viewport.

When using `position: sticky;` the element will be positioned relative to its normal position until it crosses a specified threshold, at which point it is positioned relative to the viewport.

Relative positioning is mostly used in animations because it allows you to control the position of

an element relative to its normal position, so you can move elements around without disrupting the rest of the layout.

`z-index` property is used to specify the stack order of an element. An element with greater `z-index` will be displayed above an element with a lower `z-index`.

The element with the greater `z-index` will be displayed above the element with the lower `z-index`.

default value for `z-index` is `auto`, which is the default order at which elements were written in the HTML code.

```html
<div class="one"></div>
<div class="two"></div>
<div class="three"></div>
```

```css
.one {
    position: relative;
    z-index: 1;
}

.two {
    position: relative;
    z-index: 3;
}

.three {
    position: relative;
    z-index: 2;
}
```

In the page `.two` will be displayed above `.three` and `.one` will be displayed below `.three`.

1.6 Stacking context

Stacking context is used to determine which elements appear in front of others. Elements with a higher stack order (higher `z-index` value) appear in front of elements with a lower stack order.

If we have a collection of sibling elements, each with a `z-index` value, the element with the highest `z-index` value will be displayed above the others. And if one of those elements has a lower `z-index` value than its siblings, it will be displayed below them.

Important Note

If we have 3 siblings `.one`, `.two`, and `.three` which have `z-index` values of 1, 2, and 3 respectively, and element `.one` has a child `.one-child` with `z-index` value of 999, `.one-child` will still be displayed below `.two` and `.three` because `.two` and `.three` have higher `z-index` values than the parent `.one`. But if you remove the `z-index` value from the parent `.one` and try to give `.one-child` a `z-index` value of 999 again, the `.one-child` will be displayed above `.two` and `.three`.

> Also, if we still have the parent `.one` without a `z-index` value (the child element here appears above its parent siblings) but we give it an `opacity` value like 0.5 for example, the `.one-child` item will return again below `.two` and `.three` because the `opacity` property affects the stacking context.

Coloring System:

- color name (eg. `red`)
- Hex `#RRGGBB`
- Hex with alpha `#RRGGBBAA`
- `rgb(r, g, b)`
- `rgba(r, g, b, a)` - a is the opacity

You can't specify opacity with hex colors.

To hide an element You can use:

- `display: none` - Item space will be not be reserved
- `visibility: hidden` - Item space will be reserved
- `opacity: 0` - Item space will be reserved

2 Absolute Position

Using absolute position we can make an element expand to the full width and height of the page and make it cover the whole page.

```
.cover {
    background-color: teal;
    position: absolute;
    top: 0;
    right: 0;
    bottom: 0;
    left: 0;
}
```

Mostly we use positioning when we want to make a layered design, or to make an element fixed in a position on the page without being affected by the scrolling and without affecting other elements.

Absolute position can be used to make a layer appear on top of another layer, like for example a "Sale" label on top of a product image.

It's better to use percentage % with `width` and leave `height` as `auto`, you can also use `vh` unit to make the element cover the full height of the viewport.

Session 6

Mohamed Emary

March 24, 2024

1 Some CSS Properties

1.1 Box Sizing

The `box-sizing` property allows us to include the padding and border in an element's total width and height.

```
div {
  box-sizing: border-box;
}
```

So if your element width is 100px, and you give it a 5px border instead of the 100px getting wider to 110px (not 105px because of left and right width), the 100px stays 100px and the 10px border is added inside the 100px so the actual content width is 80px.

The default value for `box-sizing` is `content-box`.

Note that `box-sizing` does not include margin, only padding and border.

So how to solve this margin issue?

To solve this we can put the element inside a container with a `border-box` box-sizing, then we give the container a padding. So the outer space for the element is the padding of the container.

1.2 Hover

To make an element interact with the mouse hover, we can use the `:hover` pseudo-class.

Pseudo-classes are keywords added to a selector that specifies a **special state of the selected elements**. Pseudo-classes are used with **a colon : behind them**.

Remember that it's only one colon : for pseudo-classes, not two because **two colons :: are used for pseudo-elements**.

In this example when we hover on the button, the background color changes to red.

```
button:hover {
  background-color: red;
```

```
}
```

The general form when dealing with pseudo-classes is:

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

You can even make the hover on one element affect another element.

```
button:hover + p {
  color: red;
}
```

1.3 Transition

To make a transition effect on an element, we can use the `transition` property.

Transition makes element changes state smoothly over a specified **duration**.

To make a transition effect on an element we need to specify 3 things:

- `transition-property` - The property we want to transition (ex: width).
- `transition-duration` - The duration of the transition (ex: 5s).
- `transition-timing-function` - The timing function. (**optional**)
- `transition-delay` - The delay before the transition starts (ex: 2s). (**optional**)

For example if we have an element that changes width on `:hover` we can make it transition smoothly in 5 seconds like this:

```
div{
  width: 100px;
  transition-property: width;
  transition-duration: 5s;
  transition-timing-function: ease-in;
}

div:hover {
  width: 300px;
}
```

The `transition-timing-function` is used to specify the speed curve of the transition effect. The default value is `ease`.

`transition-timing-function` can have the following values:

Value	Description
`cubic-bezier`	a timing function that allows you to specify your own values
`ease` *Default*	Specifies a transition effect with a slow start, then fast, then end slowly (equivalent to `cubic-bezier(0.25,0.1,0.25,1)`)
`linear`	Specifies a transition effect with the same speed from start to end (equivalent to `cubic-bezier(0,0,1,1)`)

Value	Description
`ease-in`	Specifies a transition effect with a slow start (equivalent to `cubic-bezier(0.42,0,1,1)`)
`ease-out`	Specifies a transition effect with a slow end (equivalent to `cubic-bezier(0,0,0.58,1)`)
`ease-in-out`	Specifies a transition effect with a slow start and end (equivalent to `cubic-bezier(0.42,0,0.58,1)`)

The function `cubic-bezier` takes 4 prameters:

- `x1`: The x-coordinate of the first control point
- `y1`: The y-coordinate of the first control point
- `x2`: The x-coordinate of the second control point
- `y2`: The y-coordinate of the second control point

It's hard to define the transition curve using `cubic-bezier` so you can use this website to help you.

Also this website will help you better understand each value for `transition-timing-function`.

Most of the time we don't specify each property separately, instead we use the shorthand `transition` property.

It takes the following values:

```
transition: property duration timing-function delay;
```

It's important to keep the order `duration` and `delay` values since both take time values.

you can also specify transition effect for more than one property in the same line.

```
transition: width ease-in 2s, height 4s, background-color 1s;
```

If all properties have the same duration you can specify it once.

```
transition: all 2s;
```

This will make all properties transition in 2 seconds.

for example you can use it in a code like this:

```
div {
  width: 100px;
  height: 100px;
  background-color: blue;
  transition: all 2s;
}
div:hover {
  width: 300px;
  height: 300px;
  background-color: red;
}
```

This will make the `width`, `height`, and `background-color` transition take 2 seconds.

You can even ignore `all` and just use the duration value and this will make all properties transition in the same duration specified inside `transition`.

Notice that in the last code, the transition effect was specified inside `div` and not inside `div:hover`. This is because we want our transition effect to be applied to the element itself and not only in hover state.

1.4 Transform

Transform is a CSS that allows us to move elements. It can take the following values:

- `rotate()` - rotates an element. It can take a value in degrees like `rotate(45deg)`.
- `rotateX()` - rotates an element around its X-axis.
- `rotateY()` - rotates an element around its Y-axis.
- `scale()` - scales an element. It can take two values like `scale(2,2)` which is the scale factor for the width and height.
- `scaleX()` - scales an element horizontally (width).
- `scaleY()` - scales an element vertically (height).
- `skew()` - skews an element. It can take two values like `skew(30deg, 20deg)` which is the skew factor for the horizontal and vertical axis.
- `skewX()` - skews an element horizontally.
- `skewY()` - skews an element vertically.
- `translate()` - moves an element. It can take two values like `translate(50px, 100px)` which is the distance to move the element horizontally and vertically.
- `translateX()` - moves an element horizontally.
- `translateY()` - moves an element vertically.

The transform by default happens around the center of the element, but you can change the origin of the transform using `transform-origin`, for example you can make it `top right` or `bottom left`.

You can also apply `transition` to the transform property to make the transform effect smooth.

```
1 div {
2   transition: transform 4s;
3 }
4
5 div:hover {
6   transform: rotate(360deg);
7 }
```

With `skew` you can make pretty designs inside your website. Search for skew web design.

You can use negative values in each one of these functions

You can apply more than a `transform` function to an element but it should be on the same line and separated by a space like `transform: rotate(45deg) translate(50px, 100px);`, otherwise the last transform will override the previous ones.

For example using `transform: rotate(45deg);` then using `transform: translate( 50px, 100px);` the translate will override the rotate so you will not see the rotation effect.

1.5 Overflow

The `overflow` property specifies what happens if content overflows an element's box. For example a text inside a `div` that is too much to fit inside the `div` so it overflows.

Using this property you can control the overflow of the content in four ways:

- `visible` - The overflow is not clipped. It renders outside the element's box. (*default*)
- `hidden` - The overflow is clipped, and the rest of the content will be invisible.
- `scroll` - The overflow is clipped, and a scrollbar is added to see the rest of the content.
- `auto` - Similar to `scroll`, but it adds a scrollbar only when necessary.

You can also control the overflow for each direction separately using `overflow-x` and `overflow-y`.

With `overflow` property we can solve some issues we faced before:

First: **Margin Collapse**.

Overflow & Margin Collapse

We already know margin collapse from **Session 4**, but a quick reminder:
Margin Collapse happens when two margins touch, they collapse into a single margin.
This problem happens only with top and bottom margins.

To solve margin collapse problem:

- Use `padding` on the parent container, instead of `margin` on the child container.
- Use `border` on the parent container.
- Use `overflow: auto;` on the parent container.

Second: **float related issues**.

Overflow & Float

We know from **Session 4** that float layout has two issues:

1. Floating elements are removed from the normal flow of the document, so parent element no longer contains the floated element. Example downside is if the container has a background color, it will not expand to contain the floating element.
2. The last floating element have to be cleared, otherwise it will affect the layout of the next element.

We can solve the first issue only using float
just give the parent container `overflow: auto;` property.

2 IFrame

IFrame allows us to embed another HTML page inside our current HTML page.

The syntax for IFrame is:

```
<iframe src="URL"></iframe>
```

You can also specify the width and height of the IFrame.

```
<iframe src="URL" width="500" height="500"></iframe>
```

You mostly will find embed option in the share menu of many websites like YouTube and Google Maps.

3 Important Exercise

Watch transform exercise videos on google drive.

4 Summary

In this session, we covered several **CSS properties** like:

- **Box Sizing**: This property allows us to include the padding and border in an element's total width and height by giving it `border-box` value. The default value is `content-box` and it does not include margin.
- **Hover**: This pseudo-class allows an element to interact with the mouse hover. Pseudo-Classes are used with a colon : behind them.
- **Transition**: This property allows for smooth state changes over a specified duration. It requires the specification of the transition property, duration, timing function (optional), and delay (optional).
- **Transform**: This property allows us to move elements. It can take several values such as rotate, scale, skew, and translate. The transform origin can be changed using `transform-origin`.
- **Overflow**: This property specifies what happens if content overflows an element's box. It can be set to visible, hidden, scroll, or auto.
    - We also discussed how to solve **margin collapse** and **float** related issues using `overflow: auto;` property on the parent container.

We have also covered the **IFrame** concept which allows us to embed another HTML page inside our current HTML page.

Session 7

Mohamed Emary

March 28, 2024

1 Shadow property

It's a CSS property that adds a shadow to an element.

An element can have more than one shadow. The shadow property syntax is:

```
shadow: x-shadow y-shadow blur spread color inset;
```

`inset` makes the shadow appear inside the element.

To give an element like `h1` a shadow, you can use the `text-shadow` property, It takes four values:

```
text-shadow: x-shadow y-shadow blur color;
```

The positive x direction is to the right and the positive y direction is down. You can use negative values to move the shadow in the opposite direction.

2 Gradient

It's a value that can be used with the `background` property to create a gradient background. The syntax is:

```
background: linear-gradient(direction, color1, color2, ...);
```

The direction can be `to top`, `to bottom`, `to left`, `to right`, `to top left`, `to top right`, `to bottom left`, `to bottom right`, or an angle in degrees like `45deg`.

Browser dev tools can help you to create gradients like specifying the angle.

Each color inside the `linear-gradient` can also take a percentage value to specify the position of the color.

```
background: linear-gradient(to right, red 20%, blue 50%, green 80%);
```

You can also use opacity with the colors so if you have a background image you can see it through the gradient.

```
background: url(./path/to/image), linear-gradient(to right, rgba(255, 0, 0,
↪  0.5), rgba(0, 0, 255, 0.5));
```

`radial-gradient` is another type of gradient that creates a circular gradient. The syntax is:

```
background: radial-gradient(shape size at position, color1, color2, ...);
```

3 Before and After pseudo-elements

They are used to add content before or after an element. They are used with the `::before` and `::after` selectors.

An element can't have more than one `::before` or `::after` pseudo-element.

The content of the pseudo-element can be text, an image, or nothing.

```
element::before {
  content: "before";
}
```

Example:

If you have a `h1` element and you want to add a horizontal line before and after it but it should be away from the text by 10px, you can use the following CSS:

```
h1{
  text-align: center;
  position: relative;
}
/*  review the code
h1::before, h1::after{
  content: "";
  position: absolute;
  top: 50%;
  width: 10%;
  height: 1px;
  background-color: black;
}  */
```

4 Selection

It's a CSS property that allows you to style the selected text. The syntax is:

```
selector::selection {
  color: white;
  background-color: red;
}
```

`::selection` is a pseudo-element so it has the specificity of a pseudo-element which is the same as an element.

Pseudo-elements take `::` while pseudo-classes take `:`.

5 Animation

If you want to animate an element, you can use `@keyframes` to define the animation and the `animation` property to apply the animation to the element.

The syntax of `@keyframes` is:

```
@keyframes animation-name {
  from {
    property: value;
  }
  to {
    property: value;
  }
}

/* You can also use percentages */

@keyframes animation-name {
  0% {
    property: value;
  }
  50% {
    property: value;
  }
  100% {
    property: value;
  }
}
```

The `animation` property syntax is:

```
selector {
  animation: animation-name duration delay iteration-count;
}
```

Always keep the order of `duration` and `delay`.

Example:

```
@keyframes change-colors {
  from {
    background-color: red;
  }
  to {
    background-color: blue;
  }
}

.animated {
  animation: change-colors 5s 1s infinite;
}
```

6 media queries

Session 8

Mohamed Emary

March 31, 2024

1 Flex Display

`display: flex` is a CSS property that makes the element a flex container. The flex container is the parent element that contains the flex items.

1.1 Flex Direction

`flex-direction` property specifies the direction of the flexible items. It can be set to:

- `row` (*default*)
- `row-reverse`
- `column`
- `column-reverse`

The `row` will be from left to right if the page `direction` is `ltr` (left to right) and from right to left if the page `direction` is `rtl` (right to left).

`row-reverse` will be from right to left if the page `direction` is `ltr` and from left to right if the page `direction` is `rtl`.

The same applies to `column` and `column-reverse`.

1.2 Flex Wrap

`flex-wrap` property specifies whether the flexible items should wrap or not. It can be set to:

- `nowrap` (*default*)
- `wrap`
- `wrap-reverse`

1.3 Flex Flow

`flex-flow` property is a shorthand property for the `flex-direction` and `flex-wrap` properties. It takes two values: `flex-direction` and `flex-wrap`, so it is written as `flex-flow:`

`flex-direction` `flex-wrap`.

1.4 Order

`order` property specifies the order of the flexible items. It takes a number as a value. The default value is `0`. The items are ordered based on the value of the `order` property. The items with the lower values will be placed before the items with the higher values.

1.5 Placing flex items

`justify-content` property specifies how the flexible items are placed in the flex container. It can be set to :

- `flex-start`
- `flex-end`
- `center`
- `space-between`
- `space-around`
- `space-evenly`

`align-items` property specifies how the flexible items are placed in the flex container. It can be set to:

- `stretch` (*default*)
- `flex-start`
- `flex-end`
- `center`
- `baseline` - aligns the items based on their baselines which is the line that the letters sit on

`justify-content` property works on the main axis, while `align-items` property works on the cross axis.

The main axis is the axis defined by the `flex-direction` property, while the cross axis is the axis perpendicular to the main axis.

`align-content` property specifies how the flexible items are placed in the flex container **when there is extra space in the cross axis**. It can be set to:

- `stretch` (*default*)
- `flex-start`
- `flex-end`
- `center`
- `space-between`
- `space-around`

`align-content` property only works if the flex items are wrapped.

1.6 Row & Column Gaps

`row-gap` property specifies the space between the rows of a grid layout. `column-gap` property specifies the space between the columns of a grid layout.

The `gap` property is a shorthand property for the `row-gap` and `column-gap` properties. It takes two values: `row-gap` and `column-gap`, so it is written as `gap: row-gap column-gap`.

All these properties only work if the parent element has the `display: flex` property.

1.7 Grow & Shrink

`flex-grow` property specifies how much the item will grow relative to the rest of the flexible items. It takes a number as a value.

`flex-shrink` property specifies how much the item will shrink relative to the rest of the flexible items. It takes a number as a value.

The default value for `flex-grow` is `0`, and the default value for `flex-shrink` is `1`.

For example if you have 4 items inside a flex container and you have extra space of `300px` and you set the `flex-grow` property of one item to `1` and the `flex-grow` property of the other item to `2`, the first item will take `100px` and the other items will take `200px` each.

Another example, if you have 4 items inside a flex container and you lack space of `300px` and you set the `flex-shrink` property of one item to `1` and the `flex-shrink` property of the other item to `2`, and you set the rest of the items to `0`, the first item will shrink by `100px` and the other items will shrink by `200px` each.

1.8 Flex Basis

`flex-basis` property specifies the initial size of the item before the remaining space is distributed.

The value of `flex-basis` can be a `width` or `height` depending on the `flex-direction` property, so if the `flex-direction` is `row`, the value of `flex-basis` will be a `width`, and if the `flex-direction` is `column`, the value of `flex-basis` will be a `height`.

1.9 Flex Shorthand

`flex` is a CSS property that allows us to create a flexible layout. It is a shorthand property for the `flex-grow`, `flex-shrink`, and `flex-basis` properties.

Session 9

Mohamed Emary

April 4, 2024

1 Grid Layout

Grid is a CSS layout module that allows you to create two-dimensional grid-based layouts.

Most of the time we use grid used with 2 dimensional layouts, and flex with 1 dimensional layouts.

This image will help you understand the difference:

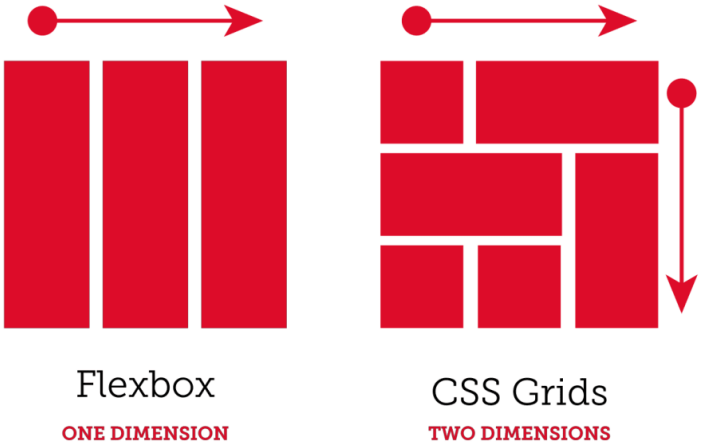

Figure 1: Grid Vs Flexbox

1.1 Display grid

To use grid, first we need to give the parent element a `display: grid` property.

Then we define the columns and rows of the grid.

```
.container {
  display: grid;
  grid-template-columns: 200px 200px 200px;
}
```

1.2 Grid Template Columns

`200px 200px 200px` means that we have 3 columns, each 200px wide, and if we have extra items they will wrap to the next row.

We can also use percentage values like `33% 33% 33%`, or use `auto` to make the columns automatically adjust to the content.

```
.container {
  display: grid;
  grid-template-columns: auto auto auto;
}
```

We can also use fractional units like `1fr 1fr 1fr` to make the columns take up equal space.

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
}
```

This will give use 3 equally spaced columns.

We can also give the middle column a double width by using `girl-template-columns: 1fr 2fr 1fr`.

And we can mix units like `grid-template-columns: 1fr 200px auto`. Or `grid-template-columns: 1fr 20% 1fr` so each `1fr` will take up 40% of the width available.

1.3 Grid Template Rows

Similarly we can define the rows of the grid using `grid-template-rows`.

```
.container {
  display: grid;
  grid-template-rows: 100px 100px 100px;
}
```

Every thing we can do with columns, we can do with rows.

Also the default hight of a row is `auto`, so it will adjust to the content.

1.4 Grid Template Shorthand Property

We can use the `grid-template` shorthand property to define both columns and rows.

It's used in the form: `grid-template: rows / columns`.

```
.container {
  display: grid;
  grid-template: 100px 100px / 1fr 1fr 1fr;
}
```

So this will give us 2 rows each taking 100px high, and 3 columns each taking 1fr of the width.

1.5 What is the difference between auto and fr?

`auto` will take up the space needed by the content, while `1fr` will take up the remaining space after the `auto` columns have been calculated.

1.6 Repeat function

We can use the `repeat()` function to repeat the same column multiple times.

The `repeat` function takes two arguments, the number of times to repeat, and the size of each row/column.

```
.container {
  display: grid;
  grid-template-columns: 200px repeat(3, 1fr);
}
```

This will give us 4 columns with the first one taking `200px` and the next 3 taking `1fr`.

1.7 Grid Template Areas

We can also use `grid-template-areas` to define the layout of the grid.

```
.container {
  display: grid;
  grid-template-areas:
    "header header header"
    "sidebar content content"
    "footer footer footer";
}
```

Then we assign the areas to the elements using the `grid-area` property.

```
.header {
  grid-area: header;
}

.sidebar {
  grid-area: sidebar;
}

.content {
  grid-area: content;
}

.footer {
  grid-area: footer;
}
```

1.8 Item Placement

1.8.1 All Items

We can use `justify-content` to align the items horizontally, and `align-items` to align them vertically.

Unlike CSS flexbox where `justify-content` aligns the items along the main axis and `align-items` along the cross axis, and the main axis and cross axis can be either horizontal or vertical depending on the direction of the flex container, in CSS grid `justify-content` always aligns the items along the x-axis and `align-content` along the y-axis.

Both `justify-content` & `align-content` can take values like:

- `start` (*default*)
- `center`
- `end`
- `space-between`
- `space-around`
- `space-evenly`

We also have a shorthand property `place-content` which combines both `justify-content` and `align-content`.

It works in the form: `place-content: align-content justify-content`.

So this:

```
align-content: space-between;
justify-content: center;
```

Is equivalent to:

```
place-content: space-between center;
```

We also have `justify-items` and `align-items` which are used to align the items **inside the grid cells**.

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  justify-items: center;
  align-items: center;
}
```

`justify-items` and `align-items` can take values like:

- `stretch` (*default*)
- `center`
- `start`
- `end`

`stretch` will stretch the items to fill the cell.

We also have a shorthand property `place-items` which combines both `justify-items` and `align-items`.

It works in the form: `place-items: align-items justify-items`.

So this:

```
align-items: center;
justify-items: start;
```

Is equivalent to:

```
place-items: center start;
```

This image will help you understand the difference between `justify-content`, `align-content` and `justify-items`, `align-items`:

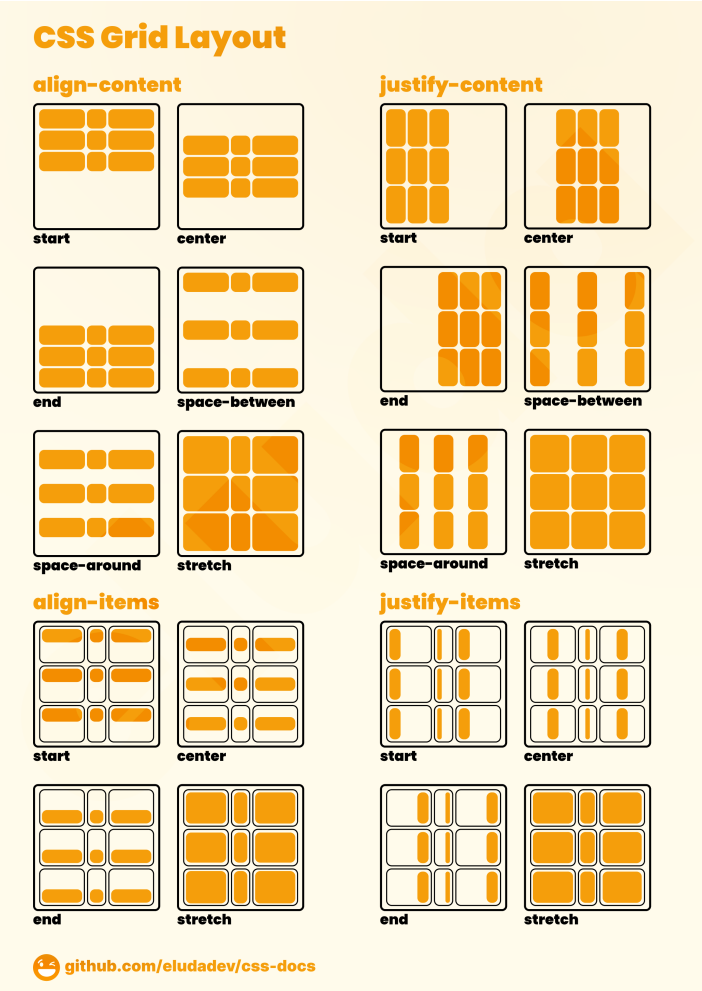

Figure 2: Content Vs Items

1.8.2 Single Item

We can use `align-self` and `justify-self` to align a **single item** inside the grid cell.

```
.item1 {
  align-self: center;
  justify-self: start;
}
```

`align-self` and `justify-self` can take values like:

- `stretch` (*default*)
- `center`
- `start`
- `end`

`stretch` will stretch the item to fill the cell.

We also have a shorthand property `place-self` which combines both `justify-self` and `align-self`.

It works in the form: `place-self: align-self justify-self`.

So this:

```
align-self: center;
justify-self: start;
```

Is equivalent to:

```
place-self: center start;
```

1.9 Grid Gap

If we want to only make a gap between the rows we can use `row-gap` and for the columns we can use `column-gap`.

The shorthand property `gap` can be used to define both the row and column gap. It takes two values, the row gap and the column gap.

It takes either one value for both row and column gap, or two values for row then column gap.

So using `gap: 20px 10px;` will give us a 20px gap between the rows and a 10px gap between the columns, and `gap: 20px;` will give us a 20px gap between both rows and columns.

1.10 Implicit Vs Explicit Grid

The explicit grid is the grid that we define using `grid-template-columns` and `grid-template-rows`.

The implicit grid is the grid that is created when we have more items than the number of columns and rows we defined.

By default, the implicit grid will create new rows to fit the extra items.

We can control how the implicit grid behaves using the `grid-auto-rows` and `grid-auto-columns` properties.

So for example using `grid-auto-rows: 100px;` will make any extra rows created by the implicit grid 100px high.

If you give `grid-auto-rows` more than one value, it will create rows with the height of the first value, then the second value, and so on.

For example using `grid-auto-rows: 100px 200px;` will make the first extra row 100px high, and the second extra row 200px high and so on.

1.11 Minmax function

We can use the `minmax()` function to define the minimum and maximum size of a row or column.

The `minmax()` function takes two arguments, the minimum size and the maximum size.

Using `grid-auto-rows: minmax(100px, auto);` any extra rows created by the implicit grid will have a minimum height of 100px and the maximum height will adjust to the content.

1.12 Cell Spanning

We can make an item span multiple rows or columns using the `grid-row-start`, `grid-row-end`, and `grid-column-start`, `grid-column-end` properties.

These properties take the line number where the item should start and the line number where it should end.

```
.item1 {
  grid-column-start: 1;
  grid-column-end: 3;
}
```

This will make the item span from the first column to the third column.

We also have shorthand properties `grid-row` and `grid-column` that can be used to define both the start and end lines.

```
.item1 {
  grid-column: 1 / 3;
}
```

So this is equivalent to the previous example.

Some Notes

If we have only 3 columns and used `grid-column: 1 / 5;` the item will span from the first column to the fifth column, and the extra columns will be created by the implicit grid.
So you can use the property mentioned before `grid-auto-columns` to define the width of these columns.

To make an items span the whole row we can use `grid-column: 1 / -1;`.
We can also use `span` keyword to make the item span multiple columns or rows, `grid-column: 2/ span 2;` will make the item start from the second column and span 2 columns.

The same thing can be done with rows.

1.13 Naming Rows and Columns

We can name the rows and columns when defining the grid using the `grid-template-rows` and `grid-template-columns` properties.

```css
.container {
  display: grid;
  grid-template-columns: [startCol] 100px [col2] 100px [col3] 100px
  ↪  [endCol];
  grid-template-rows: [startRow] 100px [row2] 100px [row3] 100px [endRow];
}
```

Then we can use these names to place the items.

```css
.item1 {
  grid-column: startCol / col3;
  grid-row: startRow / row3;
}
```

1.14 Parent Vs Child Properties

These properties are used with the grid container:

1. `grid-template-columns`
2. `grid-template-rows`
3. `grid-template`
4. `grid-template-areas`
5. `justify-content`
6. `align-content`
7. `place-content`
8. `justify-items`
9. `align-items`
10. `place-items`
11. `grid-auto-rows`
12. `grid-auto-columns`
13. `row-gap`
14. `column-gap`
15. `gap`
16. `grid-auto-rows`
17. `grid-auto-columns`

While these properties are used with the grid items:

1. `align-self`
2. `justify-self`
3. `place-self`
4. `grid-row-start`
5. `grid-row-end`
6. `grid-row`
7. `grid-column-start`
8. `grid-column-end`
9. `grid-column`

1.15 Websites to Help You Create Grid Layouts

There are many websites that help you create grid layouts visually like:

- CSS Grid Generator
- Grid Layout Generator

2 CSS Variables

CSS variables are used to store reusable values.

They are defined using the `--` prefix.

```
:root {
  --main-color: red;
}
```

Then we can use them in the CSS file using the `var()` function.

```
h1 {
  color: var(--main-color);
}
```

We can also define fallback values for the variables.

```
h1 {
  color: var(--main-color, blue);
}
```

If the `--main-color` variable is not defined, the color will be blue.

You can also use another variable as a fallback value.

```
h1 {
  color: var(--main-color, var(--secondary-color, blue));
}
```

Since our variables are scoped we defined inside `:root` so they can be accessed from anywhere in the CSS file.

The **scope** of a variable is the area where it can be accessed. For example if we define a variable inside a `div` it will only be accessible inside that `div`.

You can store any value in a variable like colors, font sizes, font family, height, width, etc.

The `:root` is the same as the `html` element, so we can use `:root` or `html` to define the variables but since `:root` is a pseudo-class and `html` is an element, it's better to use `:root` since it has higher specificity.

Don't define your variables inside `body` as you may have more than one `body` element in your HTML file (*we will get to this later*).

When writing a CSS variable with more than one word we use a hyphen `-` to separate the words. You can also use capital letters but it's not recommended.

3 Summary

3.1 Grid Layout

- Grid layout is a CSS module for creating 2D layouts.
- Use `display: grid` on the parent element.
- Define columns and rows using `grid-template-columns` and `grid-template-rows`.
  - Values can be pixels (`px`), percentages (`%`), `auto` (fits content), or `fr` (fractions of remaining space).

3.2 auto vs. fr

- `auto` takes up the space needed by the content.
- `1fr` takes up the remaining space after `auto` columns are sized.

3.3 Grid Template Areas

- Define areas using `grid-template-areas` property.
- Assign grid items to areas using `grid-area` property.

3.4 Item Placement

- `justify-content` & `align-content` align items within the container.
  - Works along x and y axis respectively.
  - Values include `start`, `center`, `end`, `space-between`, etc.
- `justify-items` & `align-items` align items within grid cells.
  - Values include `stretch` (default), `center`, `start`, `end`.
- `align-self` & `justify-self` align a single item within its cell.
- Shorthand properties available for combining these: `place-content`, `place-items`, `place-self`.

3.5 Grid Gap

- `row-gap` and `column-gap` define gaps between rows and columns respectively.
- `gap` is a shorthand property for both.

3.6 Implicit vs. Explicit Grid

- Explicit grid is defined using `grid-template-columns` and `grid-template-rows`.
- Implicit grid creates rows for extra items.
- `grid-auto-rows` and `grid-auto-columns` control implicit grid behavior.

3.7 Cell Spanning

- Use `grid-column-start`, `grid-column-end`, `grid-row-start`, and `grid-row- end` to span cells.

- Shorthand properties (`grid-row`, `grid-column`) available.

3.8 Naming Rows & Columns

- Define named rows and columns within `grid-template-rows` and `grid-template-columns`.
- Use these names for item placement with `grid-row` and `grid-column`.

3.9 Parent vs. Child Properties

- Parent properties define the overall grid layout. Listed Here.
- Child properties define individual item placement.

3.10 Sites for Creating Grid Layouts

- CSS Grid Generator
- Grid Layout Generator

3.11 CSS Variables

- Store reusable values with `--` prefix.
- Use `var()` function to reference variables in CSS.
- Define fallback values for variables.
- Variables are scoped (accessible within their defined area).
- Use `:root` (same as `html`) for global access.
- Use hyphens (-) in multi-word variable names.

Session 10

Mohamed Emary

April 7, 2024

1 Psuedo Classes

Pseudo classes are used to define a special state of an element. They have the same specificity as regular classes.

- `:first-child` - Selects the first child of an element
- `:last-child` - Selects the last child of an element
- `:nth-child(n)` - Selects the nth child of an element
- `:not(selector)` - Selects all elements that do not match the given selector

The order of the `nth-child` is relative to its siblings inside the parent element.

```
<div class="parent">
  <div class="child">1</div>
  <div class="child">2</div>
  <div class="child">3</div>
  <div class="child">4
    <div class="child">1</div>
    <div class="child">2</div>
  </div>
</div>
```

Using `.child:first-child` here will select the first child of the parent element.

Using `.child:nth-child(2)` here will select the second child of the parent element, and the second child of the 4th child of the parent element. `.child` here specifies the class of the element we want to select.

Using `.child:nth-child(2n)` will select all even children of the parent element, and using `.child:nth-child(2n+1)` will select all odd children of the parent element.

The `:nth-child` pseudo class has some compatibility issues with older browsers.

`:not(.item)` will select all elements that do not have the class `item`.

2 CSS Combinators

`.item > p` - Selects all `<p>` elements that are direct children of a `<div>` element

```
<div class="item">
  <p>First paragraph</p>      // Will be selected
  <div>
    <span>
      <p>Second paragraph</p> // Will not be selected
    </span>
  </div>
</div>
```

`.item + p` - Selects the first `<p>` element that is placed immediately after an `.item` element

```
<div>
  <div class="item"></div>
  <p>Second paragraph</p> // Will be selected
  <p>First paragraph</p>  // Will not be selected
</div>
```

`.item ~ p` - Selects all `<p>` elements that are siblings of an `.item` element

```
<div>
  <div class="item"></div>
  <p>Second paragraph</p> // Will be selected
  <p>Second paragraph</p> // Will be selected
</div>
```

using `*` will select all elements.

So in summary:

- `>` Works wil direct children
- `+` Works with the first sibling
- `~` Works with all siblings

  Note that the space is a combinator as well, and it selects all descendants. for example `.item p` will select all `<p>` elements that are descendants of an `.item` element.

3 Some Extra Selectors

3.1 Attribute Selectors

We can select elements based on their attributes.

- `[attribute]` - Selects all elements with a specific attribute
- `[attribute=value]` - Selects all elements with the specified attribute and value
- `div[class]` - Selects all `<div>` elements with a class attribute
- `div[class^="it"]` - Selects all `<div>` elements whose class attribute value begins with `it`. So for example, it will select `<div class="item">` or `<div class="item-1">` and so on.

- `div[class$="em"]` - Selects all `<div>` elements whose class attribute value ends with `em`. So for example, it will select `<div class="item">` or `<div class="problem">` and so on.
- `div[class*="it"]` - Selects all `<div>` elements whose class attribute value contains the substring `it`. So for example, it will select `<div class="item">` or `<div class="visited">` and so on.

4 Extra CSS Properties

4.1 Scroll Behavior

`scroll-behavior: smooth;` - Adds a smooth scrolling effect to the page

It can be used with an anchor tag to scroll to a specific part of the page smoothly.

4.2 Object Fit

Since images are replaced elements they can be styled using the `object-fit` property.

The default value is `fill`, which will stretch the image to fit the container.

Figure 1: fill

We can change it to `contain` to make the image fit the container without stretching it. But it will look something like this:

Figure 2: contain

Using `object-fit: cover;` will make the image cover the container without stretching it. It will look something like this:

Figure 3: cover

Using `object-fit: none;` will add the image with its original size without stretching it. It will look something like this:

Figure 4: none

Using `object-fit: scale-down;` will compare between `contain` and `none` and will use the smaller one. Our image is larger than the container so it will look the same as `contain`.

Figure 5: scale-down

4.3 Nesting Selectors

We can nest selectors in CSS. For example:

```
.parent {
  color: red;
  .child {
    color: blue;
    p{
      font-family: Arial;
    }
    &:hover {
      color: green;
    }
  }
}
```

This will make the text color of the `.parent` element red, and the text color of the `.child` element inside the parent element blue.

We can have nested selectors inside nested selectors. For example, the `p` element inside the `.child` element will have the font family of Arial.

The `&` symbol is used to refer to the parent selector. So `&:hover` will select the parent element when it is hovered which is the `.child` element in this case and change its text color to green.

4.4 Important Property

We can use the `!important` property to override any other style. It has the highest specificity even higher than inline styles.

```
p {
  color: red !important;
}
```

This will make the text color of `p` element red even if it has another color specified in the inline style.

If two same selectors have the `!important` property, the one that comes later will override the previous one. (The last one will be applied)

```
p {
  color: red !important;
}
p {
  color: blue !important;
}
```

This will make the text color of `p` element blue.

4.5 Inherit and Initial

Some properties are inherited by default. For example, font family, font size, and text color are inherited. So if we specify the font family for the `body` element, all the text inside the `body` element will have the same font family.

However, other properties like background color are not inherited, so we have to specify them for each element.

We can use the `inherit` keyword to make a property inherit the value of its parent element.

```
.parent{
  padding-top: 20px;
}

.child{
  padding-top: inherit;
}
```

The child element will inherit the `padding-top` value of the parent element, so it will have a padding top of 20px.

The `initial` keyword resets the property to its default value if it's inherited. For example, if we have a `color` property in the `body` element, and we want to reset it to its default value in the `p` element, we can use the `initial` keyword.

```
body{
  color: red;
}

p{
  color: initial;
}
```

The text color of the `p` element will be the default color which is black.

5 Solving Compatibility Issues

5.1 CSS

With CSS3, alot of new features were added, and some of them are not supported by older browsers. So we have to make sure that our website is compatible with all browsers.

To solve compatibility issues, we need to used prefixes for some properties.

- `-webkit-` for Chrome, Safari, and modern Opera
- `-moz-` for Firefox
- `-ms-` for Internet Explorer
- `-o-` for Opera

For example, `linear-gradient` is a CSS3 property that is not supported by all browsers. So we have to add prefixes for it.

```
background: -webkit-linear-gradient(red, blue);
background: -moz-linear-gradient(red, blue);
background: -ms-linear-gradient(red, blue);
background: -o-linear-gradient(red, blue);
background: linear-gradient(red, blue);
```

And it's important to add the **unprefixed version at the end**, so that the browser will use it if it supports it.

Compatibility issues are browser-specific, not web-engine-specific. For example, Chrome and Opera use the same engine, but they have different compatibility issues.

We can use a tool like Autoprefixer to add prefixes automatically to our CSS code.

6 Semantic HTML

HTML5 Introduced semantic tags like:

- `<header>` - For page header or section header
- `<footer>` - For page footer or section footer
- `<section>` - For Sections of a page
- `<main>` - For the main content of a page (Can't be repeated)
- `<nav>` - For navigation bar
- `<mark>` - For highlighted text
- `<figcaption>` - For figure caption
- `<article>` - For articles or blog posts
- `<aside>` - For content aside from the content it is placed in

Semantic tags like `<header>` have meaning, while non-semantic tags like `<div>` don't have meaning.

These elements are not supported by older browsers.

They are important for SEO and accessibility, so we have to make sure that they are supported by all browsers.

7 Font Awesome Icons

Sometime we need to add icons to our website, icons like phone, map, email, logo, etc. We can use images for that, but images are not scalable and they are larger in size which is not good for the website performance, So to solve this issue we can use an icons library like font awesome.

Font Awesome is a library of vector graphics icons that we can use in our website.

Vector graphics are images that can be scaled to any size without losing quality. They are also smaller in size than raster images so it will not affect the website performance.

It allows you to use icons like these:

To use font awesome we can use any of the following two methods:

Method 1: Using CDN

Visit font awesome page on cdnjs and copy the link of the `all.min.css` file with the version you want of font awesome library.

Then link it to your page as a stylesheet like below:

```
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax
↪  /libs/font-awesome/6.5.2/css/all.min.css">
```

This will add **all font awesome icons**, but if you want to add only some icons, for example, brands icons only you can copy its link from cdn too like this:

```
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax
↪  /libs/font-awesome/6.5.2/css/brands.min.css">
```

Method 2: Downloading the files

1. Download the font awesome compressed file from here then extract it.
2. Copy the `all.min.css` file from the `css` folder and paste it in your project CSS folder.
3. Copy the `webfonts` folder from the extracted folder and paste it in your project folder.
4. Add the following link to your HTML file:

```
<link rel="stylesheet" href="./css/all.min.css">
```

After completing any of the above methods, we can use font awesome icons in our HTML file.

To use a font-awesome icon like , we have to use the `<i>` tag with the `fa` class and the icon name class.

```
<i class="fa-brands fa-apple"></i>
```

The `fa-brands` class is used to specify the category of the icon, and the `fa-apple` class is used to specify the name of the icon.

The font-awesome icons are treated as text, so we can style them using text properties like `color`, `font-size`.

We can even add `hover` effects to them.

```
.fa-apple {
  color: blue;
  font-size: 30px;
}
.fa-facebook:hover {
  color: darkblue;
}
```

You can resize them using classes too like `fa-lg`, `fa-2x`, `fa-3x`, `fa-4x`, `fa-5x`, `fa-6x`, `fa-7x`, `fa-8x`, `fa-9x`, `fa-10x`. Some of these classes make icons appear inline, while others make them appear as block.

To give our icons the same width, we can use the `fa-fw` (**F**ont-**A**wesome **F**ixed **W**idth) class.

You can also animate icons using some classes like `fa-beat`, `fa-spin`, `fa-pulse`.

There a lot more you can do like stacking icons, making icons appear inline or block, and more. You can see all that in the font-awesome documentation from here

8 Summary

1. **Pseudo-Classes:**
   - They are used to target specific element states, not just element types.
     - Examples: `:first-child` (selects first child element), `:last-child` (selects last child element), `:nth-child(n)` (selects nth child element), `:not(selector)` (selects elements not matching a selector).
2. **CSS Combinators:**
   - They are used to combine selectors and target elements based on their structure in the HTML document.
     - `>` (greater than): selects direct children of an element.
     - `+` (plus): selects the first sibling element immediately after a specific element.
     - `~` (tilde): selects all sibling elements of a specific element.
     - Space: selects all descendant elements.
3. **Attribute Selectors:**
   - They are used to target elements based on their attribute presence or specific attribute values.
     - `[attribute]`: selects elements with a specific attribute.
     - `[attribute=value]`: selects elements with a specific attribute and value.
     - Selectors exist for elements starting with `^=`, ending with `$=`, or containing `*=` a specific value.
4. **Extra CSS Properties:**
   - `scroll-behavior: smooth;`: creates a smooth scrolling effect on the page.
   - `object-fit`: controls how images are displayed within their container, it can be:
     - `contain`
     - `cover`
     - `none`
     - `scale-down`
5. **Nesting Selectors:**
   - Allows for targeting styles based on the element hierarchy in HTML.
   - The `&` symbol within nested selectors refers to the parent selector.
6. **Important Property:**
   - `!important`: overrides any other style rule, even inline styles, due to its high specificity.
7. **Inherit and Initial:**
   - Properties can be inherited from parent elements or reset to their default values.
     - `inherit`: makes a property inherit the value from its parent element.
     - `initial`: resets a property to its default value.
8. **Solving Compatibility Issues:**

- To solve compatibility issues of some CSS3 properties we can use prefixes:
  - Prefixes are introduced for properties not supported by older browsers (`-webkit-`, `-moz-`, `-ms-`, `-o-`).
  - Tools like Autoprefixer can automate adding prefixes.

9. Semantic HTML:

- HTML tags with meaning used for better SEO and accessibility.
  - Examples: `<header>`, `<footer>`, `<section>`, `<main>`, `<nav>`, etc.

10. Font Awesome Icons:

- Explains how to integrate Font Awesome, a library of vector graphics icons, into your website.
  - Methods for including Font Awesome using a CDN or downloading files.
  - How to use Font Awesome icon classes within HTML tags.
  - Styling and customizing Font Awesome icons with CSS.

Session 11

Mohamed Emary

April 14, 2024

1 rem & em Units

1.1 rem

`rem` is a relative unit of length. It is relative to the root element's font-size.

By default, the root element is the `html` element and it has a font-size of `16px` so a `1rem` is equal to `16px`.

If you change the root font size to `20px`, `1rem` will be equal to `20px`.

```
html { /* You can also use :root */
  font-size: 20px;
}

h1 {
  font-size: 2rem; /* 40px */
}
```

1.2 em

`em` is a relative unit of length. It is relative to the font-size of the parent element, and the parent element can still be the root element.

If you have the following HTML:

```
<div>
  <p>Some text</p>
</div>
```

And the following CSS:

```
div {
  font-size: 20px;
}

p {
  font-size: 1em; /* 20px */
}
```

Here the root element still has a font-size of `16px`, but `em` is relative to the parent element,

which is the `div` element with a font-size of `20px`.

1.3 Which unit is better with each property?

- With `width` It's better to use percentages and `vh` with the first section.
- With `height` It's better to leave it as `auto`.
- With `margin`, `padding`, `border`, and `font-size` It's better to use `rem` or `em`.

2 Calc Function

`calc()` is a CSS function that can be used to perform simple arithmetic operations.

```
div {
  width: calc(100% / 4);
}
```

You can even use it to mix different units.

Like this:

```
div {
  width: calc(100% - 20px);
}
```

Or:

```
div {
  width: calc(5vw - 10px);
}
```

Another usefull example is that if you have 4 div inside a container and each of them has a margin of `10px` and width of `25%`, you can use `calc()` to calculate the width of the divs so they fit inside the container without going into the next line.

```
div {
  width: calc(100% / 4 - 20px);
  margin: 10px;
}
```

3 Bootstrap

3.1 What is Bootstrap?

Bootstrap is a free and open-source CSS library directed at responsive, mobile-first front-end web development. It contains CSS and JavaScript-based design templates for typography, forms, buttons, navigation, and other interface components.

In brief, Bootstrap is a CSS library that contains a lot of pre-built components that you can use to build your website.

Bootstrap was developed by Mark Otto and Jacob Thornton at Twitter.

It's better to watch Bootstrap inroduction video from session record on Google drive.

3.2 What is the difference between library and framework?

A library is a collection of pre-written code that you can use in your project.

A framework is larger than a library and it provides a structure for your project. It has a collection of libraries and tools that you can use to build your project.

Example of libraries: Bootstrap, tailwind, etc.

Example of frameworks: Angular, React, etc.

3.3 How to use Bootstrap?

3.3.1 Downloading Bootstrap

You can use Bootstrap by downloading the CSS and JS files from the official website and linking them in your HTML file.

Then from the downloaded files you get two files:

- `bootstrap.min.css`
- `bootstrap.bundle.min.js`

You will find these files in the `dist` folder. Then place them in your project folder.

Then in your HTML file, link the CSS file in the head section and the JS file at the end of the body section.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Trying Bootstrap</title>
    <link rel="stylesheet" href="./css/bootstrap.min.css">
  </head>
  <body>
    .
    .
    .
    your page content
    .
    .
    .
    <script src="./js/bootstrap.bundle.min.js"></script>
  </body>
</html>
```

3.3.2 Using CDN

CDN stands for Content Delivery Network. You can use Bootstrap by linking the CSS and JS files from a CDN.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Trying Bootstrap</title>
    <link
```

```
      rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist
↪   /css/bootstrap.min.css"
    >
  </head>
  <body>
    .
    .
    .
    your page content
    .
    .
    .
    <script
      src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist
↪   /js/bootstrap.bundle.min.js"
    >
    </script>
  </body>
</html>
```

3.4 Some Questions

What are the extra `.map` files that come with Bootstrap?

All the `.map` files are because Bootstrap is written in Sass and the `.map` files are used to map the compiled CSS to the original Sass files.

What is the difference between `bootstrap.min.css` and `bootstrap.css`?

`bootstrap.min.css` is the minified version of `bootstrap.css`. Minified files are smaller in size and faster to load.

So how to minify a CSS file?

You can use a CSS minifier like CSS Minifier.

What are the `rtl` files in Bootstrap?

`rtl` stands for right-to-left. These files are used to support right-to-left languages like Arabic.

What are the `utilities`, `reboot` and `grid` files in Bootstrap?

- **Utilities**: These are a collection of CSS classes that provide quick and easy-to-use styling options for various elements. Utilities in Bootstrap offer solutions for common tasks like spacing, alignment, typography, and more. They allow developers to apply styles without having to write custom CSS, which helps in maintaining consistency and speeding up development.
- **Reboot**: The "reboot" file in Bootstrap is a CSS file that provides a standardized set of CSS resets and basic styles. It aims to normalize styles across different browsers and devices, ensuring consistent rendering of HTML elements. Reboot resets the default styling of HTML elements like headings, paragraphs, lists, links, etc., making them consistent and predictable.

- **Grid**: The grid system in Bootstrap is a powerful layout utility that allows developers to create responsive and flexible layouts easily. It's based on a 12-column grid system, which provides a flexible structure for organizing content on a web page. Developers can use predefined classes to create responsive layouts that adjust automatically based on the screen size and device, ensuring a consistent user experience across different devices.

Why we use `bootstrap.min.css` and `bootstrap.bundle.min.js`?

`bootstrap.min.css` is the CSS file that contains all the styles of Bootstrap.

`bootstrap.bundle.min.js` is the JS file that contains all the JavaScript plugins of Bootstrap.

But you can still use other files that contain only the features you need.

What is the difference between `bootstrap.bundle.min.js` and `bootstrap.min.js`?

`bootstrap.bundle.min.js` includes Popper.js which is a library used to position tooltips and popovers.

3.5 What to do if you want to change the default Bootstrap styles?

You can override the default Bootstrap styles by writing your own CSS code and loading it after the Bootstrap CSS file.

```
<head>
  <title>Trying Bootstrap</title>
  <link
    rel="stylesheet"
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist
 ↪  /css/bootstrap.min.css"
  >
  <link rel="stylesheet" href="./css/styles.css">
</head>
```

Also don't override the Bootstrap class styles directly, instead, create a new class with the styling you want and apply it to the element.

You can apply the styling by overriding the variable values for the Bootstrap component you want.

In CSS:

```
.my-custom-class {
  /* Your custom styles here */
}
```

In HTML:

```
<div
  class="bootstrap-class-1 bootstrap-class-2 my-custom-class"
>
</div>
```

3.6 Sizing in Bootstrap

Bootstrap provides classes to set the width and height of elements.

- `w-25`, `w-50`, `w-75`, `w-100`, `w-auto` to set the width to 25%, 50%, 75%, and 100%, and auto respectively.
- `h-25`, `h-50`, `h-75`, `h-100`, `h-auto` to set the height to 25%, 50%, 75%, and 100% and auto respectively.

3.7 Colors in Bootstrap

Bootstrap provides classes to set the **background color**.

- `bg-primary`, `bg-secondary`, `bg-success`, `bg-danger`, `bg-warning`, `bg-info`, `bg-light`, `bg-dark`, `bg-white` to set the background color.

And to change the **text color** you can use the following classes.

- `text-primary`, `text-secondary`, `text-success`, `text-danger`, `text-warning`, `text-info`, `text-light`, `text-dark`, `text-white` to set the text color.

You can also change the background color opacity using the `--bs-bg-opacity` variable.

```
.bg-primary {
  --bs-bg-opacity: 0.5;
}
```

We also have some classes to control the **opacity** of the **background color**.

- `bg-opacity-25`, `bg-opacity-50`, `bg-opacity-75`, `bg-opacity-100` to set the opacity to 25%, 50%, 75%, and 100% respectively.

And to control the **opacity of the element** itself, you can use the following classes.

- `opacity-0`, `opacity-25`, `opacity-50`, `opacity-75`, `opacity-100` to set the opacity to 0%, 25%, 50%, 75%, and 100% respectively.

3.8 Alignment

Bootstrap provides classes to align elements.

- `text-start`, `text-center`, `text-end` to align text.

3.9 Text in Bootstrap

3.9.1 Font Size

- `fs-1`, `fs-2`, `fs-3`, `fs-4`, `fs-5`, `fs-6` to set the font size.

3.10 Font Style

- `fst-italic` to set the font style to italic.

3.10.1 Font Weight

- `fw-light`, `fw-normal`, `fw-bold` to set the font weight.

3.10.2 Text Transform

- `text-lowercase`, `text-uppercase`, `text-capitalize` to transform text.

3.10.3 Line Height

- `lh-base`, `lh-sm`, `lh-lg` to set the line height to base, small, and large respectively.

There are many other effects and classes that you can use in Bootstrap. You can find them in Bootstrap documentation.

Session 12

Mohamed Emary

April 18, 2024

1 Bootstrap Cont

Bootstrap contains:

- **Components** - Predefined components that can be used in the web page like buttons, forms, etc.
- **Utitlity classes** - Classes that can be used to apply some styles to the elements.
- **Grid system** - A system that helps to create responsive web pages.

1.1 Screen Sizes

Bootstrap has the following screen sizes:

- Screen Sizes:
  - Extra small $<$ 576px
  - Small `sm` $\geq$ 576px
  - Medium `md` $\geq$ 768px
  - Large `lg` $\geq$ 992px
  - Extra large `xl` $\geq$ 1200px
  - Extra extra large `xxl` $\geq$ 1400px

1.2 Containers

- `container` - Container class which has a `max-width` depending on the screen size, it also has a right and left padding.
- `container-fluid` - Container class which has a `width` of 100% in all screen sizes, it still has a right and left padding.
- `container-{SCREEN-SIZE}` - Container class which has a `max-width` depending on the screen size, it also has a right and left padding. ex: `container-sm`, `container-md`, `container-lg`, `container-xl`, `container-xxl`.

- `container-lg` - has a 100% width in extra small, small, and medium screens, and a fixed width in large and above.

2 Flex Display

- `d-flex` - Display flex
- `row` - It has a flex display with `flex-wrap` set to `wrap`, it also has a negative margin to counter the padding of the container and if you remove the container you will notice a horizontal scroll bar.

If you want to test the negative margin try the following code:

```
<div class="negMargin">
</div>
```

```
.negMargin {
  height: 100px;
  background-color: teal;
  margin: 0 -15px;
}
```

You will see a horizontal scroll bar no matter how wide your screen is. Try to remove the margin or make it positive and it will disappear.

Now try putting it inside a container that has a padding like the following:

```
<div class="container">
  <div class="negMargin"></div>
</div>
```

And give the container a padding:

```
.container {
  background-color: gold;
  padding: 15px;
}
```

The horizontal scroll bar will disappear and you will see the padding of the container only from upper and lower sides since the negative margin is only on the left and right sides.

It will look like this:

Figure 1: Negative Margin

And if you give the container a padding larger than the negative margin (for example 25px) it will look like this:

Figure 2: Side Padding

`row` also allows us to specify the number of items in the row using the classes `col-{NUMBER-OF-ROWS}` followed by the number of columns you want to divide the row into. You can also specify them based on each screen size, for example, `col-sm-6` will make the item take half the width of the row in small screens.

Bootstrap grid system has 12 columns so you can divide the row into 12 columns.

By default if you specify the number of columns an element inside a container spans in a specific screen size, the larger screen sizes will also have the same number of columns and smaller screen sizes will have 12 columns.

For example if your element has the class `col-lg-4` it will span 4 columns in large, extra large, and extra extra large screens, and 12 columns in medium, small, and extra small screens.

The `row` must be the direct parent of the items that have the `col-` class.

`row` has a default right and left padding (gutter) for all its direct children. If you want to change it you can use `gx-{NUMBER}` The number should be between 0 and 5.

With `row` `gy-` is used for vertical **margin** not padding.

So in summary:

- Gutter is used with `row`.
- `gx-{NUMBER}` is used for horizontal **padding** (it's not margin because margin would have caused the elements to go to next line and `box-sizing: border-box` will not be able to fix it).
- `gy-{NUMBER}` is used for vertical **margin**.
- `g-{NUMBER}` allows you to change both `gx-` and `gy-` at the same time.

When using `row` you can make all the elements inside the row have the same width by using the class `col` without specifying the number of columns. You can also specify at which screen size you want the element to have the same width by adding the screen size to the class like `col-lg` so this will make the element have the same width in large screens and above then you can use `col-sm-6` to make the element take half the width in small and medium screens, and the extra small screen will have 12 columns.

2.1 Offset

`offset` is used to give the elements a `margin-left` that is equal to a number of columns you specify. For example, `offset-2` will give the element a margin-left equal to the width of 2 columns.

`offset` is used when you have extra columns space in the row.

You can also make that offset appear in a specific screen size by adding the screen size to the class like `offset-md-2` so this will make the offset appear only in medium screens or larger.

2.2 Min and Max Width

`min-width` is the minimum width that an element can have, the element can be larger than it but not smaller.

`max-width` is the maximum width that an element can have, the element can be smaller than it but not larger.

3 Summary

- **Containers**:
  - `container` - has a `max-width` depending on the screen size.
  - `container-fluid` - has a `width` of 100% in all screen sizes.
  - `container-{SCREEN-SIZE}` - has a `max-width` in that screen size or larger and 100% in smaller screen sizes.
- `row` & **Flex Display**:
  - `d-flex` - Display flex.
  - `row` - It has a flex display with `flex-wrap` set to `wrap`.
  - `row` has a negative margin to counter the padding of the container.
  - `col-{NUMBER-OF-COLUMNS}` - Specify the number of columns an element will span in a row. -**Gutter**
  - `gx-{NUMBER}` - Change the horizontal padding of items inside the row.
  - `gy-{NUMBER}` - Change the vertical margin of items inside the row.
  - `g-{NUMBER}` - Change both the horizontal padding and vertical margin of items inside the row.
- `col`
  - `col` - Make all the elements inside the row have the same width.
  - `col-{SCREEN-SIZE}` - Make all the elements inside the row have the same width in that screen size or larger.
- `offset`
  - `offset-{NUMBER-OF-COLUMNS}` - Give the element a margin-left equal to the number of columns you specify.
  - `offset-{SCREEN-SIZE}-{NUMBER-OF-COLUMNS}` - Give the element a margin-left equal to the number of columns you specify in that screen size or larger.

Session 13

Mohamed Emary

April 21, 2024

1 Inside head tag

`meta` tag takes: name & content

keywords

```
<meta name="keywords" content="HTML, CSS, JavaScript">
```

author

```
<meta name="author" content="John Doe">
```

description

```
<meta name="description" content="Free Web tutorials">
```

viewport (responsive design)

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Content-Type

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

important for SEO

Link `icon`

```
<link rel="icon" href="favicon.ico">
```

`.ico` is preferred. There is sites to convert `.png` to `.ico`.

Character encoding

```
<meta charset="UTF-8">
```

OR:

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

`html lang`

```
<html lang="en">
```

Before `<html>`

`<!DOCTYPE>` declaration

```
<!DOCTYPE html>
```

Means the document is an HTML5 document.

It enforces some rules on the document CSS like:

- case-sensitive selectors
- having to write `px` after a number
- images inside a div with a border there will be a very small gap between the image and the border since the image is a replaced element and to fix this you can set the image to `display: block;`

2 Other

`svg` tag

It can be used to create shapes and images.

```
<svg width="100" height="100">
  <circle cx="50" cy="50" r="40" stroke="black" stroke-width="3" fill="red"
  ↪  />
</svg>
```

`circle` is the shape, `cx` and `cy` are the center of the circle, `r` is the radius, `stroke` is the color of the border, `stroke-width` is the width of the border, `fill` is the color of the inside of the circle.

- `form-control` and `gy-2` classes
- `btn-outline-warning`

---

svg and another section with the same color to make it look like better

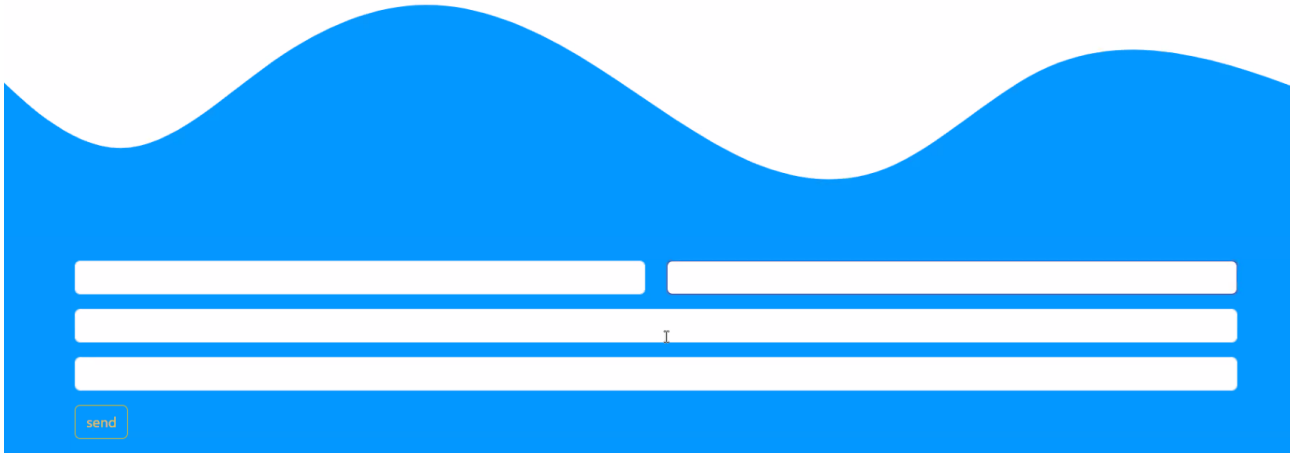

Figure 1: 1713722502526

In this image the waves above appear to be a part of the form however they are not, but since they have the same color they look like they are.

`max-width` and `min-width`

Download fonts from google fonts and create a folder for them in the project. Then move the downloaded font file (`.ttf`, `.woff`, `.woff2`, `.eot`) to the fonts folder.

Then use the `@font-face` rule to use them. `font-family` is the name of the font. `src` is the path to the font.

```
@font-face {
  font-family: 'Roboto';
  src: url('fonts/Roboto-Regular.ttf');
}
```

You can have multiple `src` to support different browsers.

```
@font-face {
  font-family: 'Roboto';
  src: url('fonts/Roboto-Regular.ttf') format('truetype'),
       url('fonts/Roboto-Regular.woff') format('woff');
}
```

Then use the `font-family` in the CSS.

You can have more than one font with the same name and the browser will use the one that is needed.

for example:

fonts with the same name and different weights.

```
@font-face {
  font-family: 'Roboto';
  src: url('fonts/Roboto-Regular.ttf');
}
```

```

@font-face {
  font-family: 'Roboto';
  src: url('fonts/Roboto-Bold.ttf');
}
```

So using:

```
font-family: 'Roboto';
font-weight: bold;
```

will use the bold font.

while using:

```
font-family: 'Roboto';
font-weight: normal;
```

will use the regular font.

Importing font awesome javascript file will make the icons appear as svg. You can see that in dev tools.

`navbar-expand-lg` is used to make the navbar responsive and expand when the screen is large.

In html 5 you can make a tag with any name and it will be valid and it will have inline display by default.

You can also create your own attributes in the form `data-*`. Then you can select it in the CSS using `[data-*]`.

Bootstrap has its own tags like `data-toggle` and `data-target` to make the navbar work.

Slider exercise

---

cards exercise

tabs part of the exercise

- Alert
- Accordion
- Badge
- Breadcrumb

And many Other Components

`popper.min.js` and `bootstrap.min.js` are merged in `bootstrap.bundle.min.js`

Session 14

Mohamed Emary

April 24, 2024

1 Lighthouse

Lighthouse is a tab in chrome dev tools that helps you to test the performance, accessibility, and SEO of your website, then it gives you a score based on these tests.

2 Video & Audio Tags

2.1 Video

```
<video src="video.mp4" controls></video>
```

Any video is a replaced element just like an image, so you can set its width and height.

Video tag has some attributes:

- `controls` attribute: adds a play/pause button, volume control, and a progress bar.
- `autoplay` attribute: plays the video automatically, but some browsers block it.
- `muted` attribute: mutes the video.
- `loop` attribute: plays the video in a loop.

2.2 Audio

```
<audio src="audio.mp3" controls></audio>
```

Audio tag has the same attributes as the video tag.

3 JavaScript

JS story in video and where did its name come from.

4 where to write JS code

- Inline
- Internal
- External
- `window.alert`
- `console.log`
- `document.getElementById`
- `document.getElementById("demo").innerHTML = "Hello JavaScript";`
- `console.log(document.getElementById("demo"));`
- `console.log(document.getElementById("demo").innerHTML);` prints the tag
- Comments one line and multi-line
- variable declaration assignment:
  - in the same line or in different lines
  - since its a variable u can reassign it
  - the name can't start with a number or a special character except for `$` and `_`
  - variable name can't be a reserved word like `var`, `while`, `function`, etc
- Use camelCase for naming
- don't redeclare a variable with the same name
- Data types:
  - Primitive
    1. Number
    2. String
    3. Boolean
    4. Null
    5. Undefined
  - Non-primitive
- To know the data type of a variable use `typeof`
- JS is a loosely typed language which means you don't have to put the data type of a variable when declaring it.
  - Languages that force you to declare the data type of a variable are called statically typed languages.
- strings are written inside `"string"` or `'string'` or `` `String` `` (backticks)
- `typeof null` is `object` which is a bug in JS

Session 15

Mohamed Emary

April 28, 2024

1 Operators In JavaScript

1.1 Arithmetic Operators

Arithmetic operators in JS are: `+`, `-`, `*`, `/`, `**`, and `%`.

The `%` operator returns the remainder of a division, for example if `x = 5` and `y = 2`, then `x % y` is `1`. When the remainder is `0`, it means that `y` is divisible by `x`.

When `+` is used with strings, it concatenates them, for example:

```
var x = "Hello" + "World"; // x is "HelloWorld"
var y = "Hello" + 1 + 2;   // y is "Hello12"
var z = 1 + 2 + "Hello";   // z is "3Hello"
```

1.2 Assignment Operators

Assignment operators in JS are: `=`, `+=`, `-=`, `*=`, `/=`, `**=`, and `%=`.

1.3 Comparison Operators

Comparison operators in JS are: `==`, `===`, `!=`, `!==`, `>`, `<`, `>=`, and `<=`.

```
var x = 5;
var y = 5;
console.log(x >= y);  // true

x = "5";
console.log(x == y);  // true
console.log(x != y);  // false

console.log(x === y); // false
console.log(x !== y); // true
```

So what is the difference between `==` and `===`?

- `==` is used to compare values, while `===` is used to compare values and types.
- `console.log(x == y);` is true because JS converts the string to a number if possible.
- `console.log(x === y);` is false because `===` does not convert the types.

1.4 Logical Operators

Logical operators in JS are: `&&`, `||`, and `!` (AND, OR, and NOT).

Explanation:

- `&&` is true if all conditions are true and false if at least one condition is false.
- `||` is true if at least one condition is true and false if all conditions are false.
- `!` is used to reverse the result, so if a condition evaluates to true, `!` will make it false.

```
var x = 5;
var y = 10;
console.log(x > 3 && y < 20);  // true
console.log(x > 3 || y > 20);  // true
console.log(!(x > 3));         // false
```

Logical operators are commonly used in to make decisions in JS using conditional statements.

2 Conditional Statements

2.1 If Statement

The `if` statement is used to execute a block of code if a condition is true.

```
var x = 5;
if (x > 0) {
  console.log("x is positive");
}
```

The statement `console.log("x is positive");` will only be executed if `x > 0`.

2.2 Else Statement

The `else` statement is used to execute a block of code if the same condition is false.

```
var x = -5;
if (x > 0) {
  console.log("x is positive");
} else {
  console.log("x is negative");
}
```

2.3 Else If Statement

The `else if` statement is used to specify new conditions if the previous conditions are false.

```
var skill = "HTML";
if (skill == "CSS") {
```

```
  console.log("CSS");
} else if (skill == "HTML") {
  console.log("HTML");
} else if (skill == "JavaScript") {
  console.log("JavaScript");
} else {
  console.log("Another skill");
}
```

Here if `skill` is not `CSS` it checks if it is `HTML`, and if not, it checks if it is `JavaScript` and if not, it prints `Another skill`.

2.4 Nesting If Statement

A nested `if` statement is an `if` statement inside another `if` statement.

```
var x = 10;
var y = 20;
if (x == 10) {
  if (y == 20) {
    console.log("x is 10 and y is 20");
  }
}
```

2.5 Switch Statement

The `switch` statement is used to perform different actions based on different conditions.

```
var day = 3;
switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  default:
    console.log("Another day");
}
```

The `break` statement is used to break out of the switch block, because JS will execute the next switch case if a `break` is not found.

Switch statement has a better performance than if-else statement.

2.6 Nested Switch Statement

A nested `switch` statement is a `switch` statement inside another `switch` statement.

```
var day = 3;
var month = 4;
```

```
switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    switch (month) {
      case 4:
        console.log("Wednesday, April");
        break;
      case 5:
        console.log("Wednesday, May");
        break;
      default:
        console.log("Another Month");
    }
    break;
  default:
    console.log("Another day");
}
```

2.7 Falsey Values

Falsey values in JS are values that are considered false when evaluated in a boolean expression. They include: `false`, `0`, `""`, `null`, `undefined`, and `NaN`.

```
var x = "";
var y = "Mohamed";
console.log(x && y);  // prints nothing
console.log(x || y);  // Mohamed
```

The first `console.log` prints nothing because `x` is falsey and we are using the `&&` operator so both conditions must be true to execute the statement, while the second `console.log` prints `Mohamed` because `y` is truthy and we are using the `||` operator so only one condition must be true to execute the statement.

The `&&` stops evaluating with the first false value, while the `||` stops evaluating with the first true value.

3 Loops

Loops are used to execute the same block of code multiple times.

3.1 For Loop

The `for` loop is used to execute a block of code a number of times.

Syntax:

```
for (initialization; condition; step) {
  // code block to be executed
}
```

Example:

```
for (var i = 0; i < 5; i++) {
  console.log(i);
}
```

The loop will print the numbers from `0` to `4`.

`i++` is the same as `i = i + 1`, `i += 1`.

To print even numbers from `0` to `10`:

```
for (var i = 0; i <= 10; i += 2) {
  console.log(i);
}
```

These two code blocks will print forever (infinitive loop):

```
for (;;) {
  console.log("Hello");
}
```

```
for (var i = 0; i < 5;) {
  console.log(i);
}
```

While this one will cause an error because of a missing `;`:

```
for (;) {
  console.log("Hello");
}
```

3.2 While Loop

The `while` loop is used to execute a block of code as long as a condition is true.

Syntax:

```
while (condition) {
  // code block to be executed
}
```

Example:

```
var i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

The loop will print the numbers from `0` to `4`.

3.3 Do While Loop

The `do while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax:

```
do {
  // code block to be executed
} while (condition);
```

Example:

```
var i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

3.4 Using Loops With HTML Elements

You can use loops to manipulate HTML elements.

In HTML:

```
<ul id="list">
</ul>
```

In JS:

```
var list = document.getElementById("list");
for (var i = 0; i < 6; i++) {
  if (i % 2 === 0) {
    var item = "<li class='red'>Item " + i + "</li>";
  } else {
    var item = "<li class='green'>Item " + i + "</li>";
  }
  list.innerHTML += item;
}
```

You can even use the classes to style the elements.

```
.red {
  color: red;
}

.green {
  color: green;
}
```

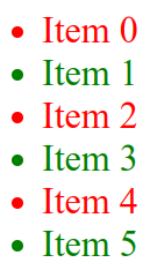

Figure 1: Final Result

4 Summary

- Operators in JS include arithmetic, assignment, comparison, and logical operators.
- Arithmetic operators include `+`, `-`, `*`, `/`, `**`, and `%`.
  - `+` is used to concatenate strings.
  - `%` returns the remainder of a division.
- Assignment operators include `=`, `+=`, `-=`, `*=`, `/=`, `**=`, and `%=`.
- Comparison operators include `==`, `===`, `!=`, `!==`, `>`, `<`, `>=`, and `<=`.
  - `==` is used to compare values, while `===` is used to compare values and types.
- Logical operators include `&&`, `||`, and `!`.
- Conditional statements include `if`, `else`, `else if`, and `switch`.
  - `if` is used to execute a block of code if a condition is true.
  - `else` is used to execute a block of code if the same condition is false.
  - `else if` is used to specify new conditions if the previous conditions are false.
  - `switch` is used to perform different actions based on different conditions.
    - `break` is used with `switch` to break out of a case, otherwise JS will execute the next cases till the end.
    - `default` is used with `switch` to execute a block of code if no case is true.
- Falsey values are values that are considered false in a boolean expression, they include `false`, `0`, `""`, `null`, `undefined`, and `NaN`.
- Loops include `for`, `while`, and `do while`.
  - `for` is used to execute a block of code a number of times.
  - `while` is used to execute a block of code as long as a condition is true.
  - `do while` is used to execute a block of code once before checking the condition.

Session 16

Mohamed Emary

May 5, 2024

Important Note

This session discusses a lot of important and tricky concepts that gets asked frequently in interviews. So, make sure you understand the concepts well.

1 Implicit Conversion

Implicit conversion is the automatic conversion of a value from one data type to another. This is done by the JavaScript engine when the data type of the operands in an expression are different.

For example when you add a number and a string, JavaScript will convert the number to a string and concatenate the two strings.

```
var a = 20;
var b = "10";
var c = a + b;
console.log("c = " + c);
var f = a - b;
console.log("f = " + f);
var d = a * b;
console.log("d = " + d);
var e = a / b;
console.log("e = " + e);
```

```
c = 2010
f = 10
d = 200
e = 2
```

As you see the results below the code:

- `c` equals `2010` because `a` is converted to a string and concatenated with `b`.
- `f` equals `10` because `b` is converted to a number and subtracted from `a`.
- `d` equals `200` because `b` is converted to a number and multiplied by `a`.
- `e` equals `2` because `b` is converted to a number and divided by `a`.

But what if the value can't be converted to the desired data type? In this case, the result will be `NaN` (Not a Number).

```
var a = 20;
var b = "Hello";
var c = a - b;
console.log("c = " + c);
```

```
c = NaN
```

The result will be `NaN` because the string `Hello` can't be converted to a number to be subtracted from `a`.

Lets see the result of string to number conversion of some strings.

```
console.log( 'true = ' + Number(true))
console.log( 'false = ' + Number(false))
console.log( '"0" = ' + Number('0'))
console.log( 'null = ' + Number(null))
console.log( 'undefined = ' + Number(undefined))
console.log( '"" = ' + Number(''))
console.log( 'NaN = ' + Number(NaN))
```

```
true = 1
false = 0
"0" = 0
null = 0
undefined = NaN
"" = 0
NaN = NaN
```

- `true` will be converted to `1` and `false` will be converted to `0`.
- Since the string `0` can be converted to a number, the result will be `0`.
- `null`, `""` will be converted to `0`.
- `undefined` will be converted to `NaN`.
- `NaN` stays as `NaN`.

What about adding these values to a number?

```
console.log( 'true + 1 = ' + (true + 1))
console.log( 'false + 1 = ' + (false + 1))
console.log( 'null + 1 = ' + (null + 1))
console.log( 'undefined + 1 = ' + (undefined + 1))
console.log( 'NaN + 1 = ' + (NaN + 1))
```

```
true + 1 = 2
false + 1 = 1
null + 1 = 1
undefined + 1 = NaN
NaN + 1 = NaN
```

What about adding these values to a string?

```
console.log( 'true + "1" = ' + (true + "1"))
console.log( 'false + "1" = ' + (false + "1"))
console.log( 'null + "1" = ' + (null + "1"))
console.log( 'undefined + "1" = ' + (undefined + "1"))
console.log( 'NaN + "1" = ' + (NaN + 1))
```

```
true + "1" = true1
false + "1" = false1
null + "1" = null1
undefined + "1" = undefined1
NaN + "1" = NaN
```

Lets try some extra examples.

```
console.log( 'true + true = ' + (true + true))
console.log( 'true + false = ' + (true + false))
console.log( 'false + false = ' + (false + false))
console.log( 'true + null = ' + (true + null))
console.log( 'true + undefined = ' + (true + undefined))
console.log( 'true + NaN = ' + (true + NaN))
```

```
true + true = 2
true + false = 1
false + false = 0
true + null = 1
true + undefined = NaN
true + NaN = NaN
```

Some extra examples:

```
console.log( '"3" * false = ' + ("3" * false))
console.log( '"3" * true = ' + ("3" * true))
console.log( '"3" / false = ' + ("3" / false))
console.log( '"3" / true = ' + ("3" / true))
```

```
"3" * false = 0
"3" * true = 3
"3" / false = Infinity
"3" / true = 3
```

2 Function

Functions are reusable blocks of code that can be called multiple times. They can take parameters and return values.

Functions are defined using the `function` keyword followed by the function name and a list of parameters in parentheses. The function body is enclosed in curly braces `{}`.

Functions parameters are created without using `var` keyword.

```
function FunctionName (parameter1, parameter2, ...) {
    // function body
}
```

And to use the function you have to call it, just write the function name followed by parentheses and any arguments you want to pass to the function.

The arguments are the values of the parameters that the function will use.

Example:

```
function greet(name) {
    console.log("Hello " + name);
}
greet("World");
```

```
Hello World
```

The `greet("World")` is called a function call or invocation. The value inside the parentheses is called an argument.

For reusability purposes, it's preferred that each function does only one thing, this is called the single responsibility principle.

Functions can access already defined variables in the global scope.

```
var name = "World";
function greet() {
    console.log("Hello " + name);
}
greet();
```

```
Hello World
```

2.1 Return Value

Functions can return a value using the `return` keyword followed by the value to return.

```
function add(a, b) {
    return a + b;
}
var result = add(10, 20);
console.log("result = " + result);
```

```
result = 30
```

Any code after the `return` statement will not be executed.

The function that doesn't have a `return` statement will return `undefined`.

```
function greet(name) {
    console.log("Hello " + name);
}
var result = greet("World");
console.log("result = " + result);
```

```
Hello World
result = undefined
```

The first output line here `Hello World` is because of the `console.log` inside the function, and the second output line `result = undefined` is because the function doesn't have a `return` statement.

A return function is the function that have a return statement.

Return is used when you want to get a value from a function to use it in another part of the code.

You can even return another function.

2.2 Function Types

2.2.1 Declaration Function

This is the most common way to define a function. It is defined using the `function` keyword followed by the function name.

Declaration functions always start with the `function` keyword.

```
function greet(name) {
    console.log("Hello " + name);
}
greet("World");
```

```
Hello World
```

2.2.2 Expression Function

This is when you assign a function to a variable.

```
var greet = function(name) {
    console.log("Hello " + name);
}
greet("World");
```

```
Hello World
```

In this example, the function is assigned to the `greet` variable and then called using the variable.

When you print the value of the `greet` variable you will get the function definition.

```
console.log(greet);
```

```
ƒ (name) {                                    main.js:14
  console.log("Hello " + name);
}
```

Figure 1: Console Output

3 User Input

To get a value from user you can use the `prompt` function. This function will show a dialog box to the user with a message and an input field, and it will return the value entered by the user.

```
var name = prompt("Enter your name");
console.log("Hello " + name);
```

This code will make a dialog box appear with the message “Enter your name” and an input field. The value entered by the user will be stored in the `name` variable then it will be printed to the console.

4 Some Interview Notes

What will be the result of this code?

```
function salaryBonus(salary) {
    console.log(salary + 100);
}
salaryBonus();
salaryBonus(100);
salaryBonus(100, 200);
```

The results will be:

1. `NaN` because `salary` is not defined so `undefined + 100` is `NaN`.
2. `200` because `salary` is `100`.
3. `200` because `salary` is `100` and `200` is ignored.

5 Hoisting

JavaScript hoisting is a mechanism where variable and function declarations are moved to the top of their containing scope during the compilation phase.

That is the reason why when you print the value of a variable before declaring it, you will get `undefined` instead of an error.

```
console.log(a);
var a = 10;
```

```
undefined
```

But if you just print the value of a variable without declaring it, you will get an error.

```
console.log(a);
```

With the code above you will get: `ReferenceError: a is not defined`

Since hoisting also works with functions, you can call a function before declaring it.

```
greet("World");
function greet(name) {
    console.log("Hello " + name);
}
```

```
Hello World
```

These weird behaviors happen because of hoisting.

You should notice that only the declaration is hoisted not the initialization, so if you have `var a = 10;` the `var a;` part will be hoisted but the `a = 10;` part will not and that is why you get `undefined` when you print the value of `a`.

So these are equivalent and both will print `undefined`.

```
console.log(a);
var a = 10;
```

```
undefined
```

```
var a;
console.log(a);
a = 10;
```

```
undefined
```

6 Hoisting With Function Types

6.1 Declaration Functions Hoisting

Declaration functions get hoisted so you can call the function before declaring it.

6.2 Expression Functions Hoisting

Expression functions don't get hoisted since only the variable declaration gets hoisted and not the initialization (which is the function definition)

For example in:

```
var greet = function(name) {
    console.log("Hello " + name);
}
```

only `var greet;` gets hoisted.

so you can't call an expression function before the line it was assigned to the variable. In other words to make a function that can only be called after its definition use expression function.

7 Scope

Scope is the context in which a variable is defined. JavaScript has two types of scope: **global scope** and **local scope**.

7.1 Global Scope

A variable is in the global scope if it's declared outside of any function or block.

```
var a = 10;
function getNum() {
    console.log(a);
}
getNum();
```

In this example, the variable `a` is declared in the global scope and **can be accessed** from the `getNum` function.

7.2 Local Scope

A variable is in the local scope if it's declared inside a function or block.

```
function getNum() {
    var a = 10;
}
console.log(a);
```

In this example, the variable `a` is declared in the local scope of the `getNum` function and **can't be accessed** from outside the function, so you will get `ReferenceError: a is not defined`.

Inside a function you can access the variables of the global scope but in the global you can't access the variables of the function local scope.

The normal behavior assumes that each `{ }` creates a new scope but this is not the case when using `var` keyword, so if you create a loop or an if condition for example and declared variables inside their `{ }` using `var` keyword, you still can access these variables from outside the `{ }`, and the only exception is the function scope. The same happens with whole functions if you declared a function inside another function you can only access it from inside the function.

7.3 Qestions

What will be the output of each of these codes?

Code 1

```
var a = 10;
function myFunction() {
    var a = 20;
    console.log(a);
}
myFunction();
console.log(a);
```

```
20
10
```

The output of the function call will be 20 and the output of the second `console.log` will be 10 because the function accessed the variable in the local scope which had the value `20` while the `console.log` accessed the variable in the global scope which had the value `10`.

Code 2

```
var a = 10;
function myFunction() {
    console.log(a);
    a = 20;
}
myFunction();
console.log(a);
```

```
10
20
```

The output of the function call will be 10 and the output of the second `console.log` will be 20 because the function accessed the variable in the global scope which had the value `10` and changed it to `20` then the `console.log` accessed the same variable after it was changed.

Code 3

```
var a = 10;
function myFunction() {
    console.log(a);
    var a = 20;
}
myFunction()
console.log(a);
```

```
undefined
10
```

The output of the function call will be `undefined` and the output of the second `console.log` will be `10` because the function accessed the variable in the local scope which was hoisted to the top of the function but not initialized yet so it was `undefined` while the `console.log` accessed the variable in the global scope which had the value `10`.

Notice that the variable inside the function gets hoisted to the top of the function not to the top of the global scope.

Code 4

```
function foo() {
    function bar(){ return 3;}
    function bar(){ return 8;}
    return bar();
}
console.log(foo())
```

```
8
```

The output will be `8` because the second function definition will override the first one.

Code 5

```
function foo() {
    var bar = function(){ return 3;}
    return bar();
    var bar = function(){ return 8;}
}
console.log(foo())
```

```
3
```

The output will be `3` because the second function definition will be ignored because it's an expression function that is not hoisted so only the variable declaration will be hoisted and the value of the variable will be the first function.

Code 6

```
function foo() {
    function bar(){ return 3;}
    return bar();
    function bar(){ return 8;}
}
console.log(foo())
```

```
8
```

The output will be `8` because both functions are declaration functions so both will be hoisted (in the same order they were defined) and the second function will override the first one.

Its equivalent to this code:

```
function foo() {
    function bar(){ return 3;}
    function bar(){ return 8;}
    return bar();
}
console.log(foo())
```

Code 7

```
function foo(){
    return bar();
```

```
    function bar(){ return 3;}
    var bar = function(){ return 8;}
}
console.log(foo())
```

3

The output will be `3` because the declaration function definition will be hoisted to the top before the return statement while in the expression function definition only the variable declaration will be hoisted but not the function assigned to it, you should also know that the value returned here is the result of the function call since we are using `()` after the function name but if we remove the `()` the value returned will be the function definition similar to 1.

The code above is equivalent to this code which will also print `3`:

```
function foo(){
    function bar(){ return 3;}
    var bar;
    return bar();
    bar = function(){ return 8;}
}
console.log(foo())
```

Code 8

```
console.log(foo())
var foo = function(){
    return bar();
    function bar(){ return 3; }
    var bar = function(){ return 8; }
}
```

The output will be `TypeError: foo is not a function` because the variable `foo` is hoisted to the top but not initialized yet so it's value is `undefined` and you can't call `undefined` as a function. (if you `console.log(foo)` you will get `undefined`)

8 Self Inovked Function

It's also called IIFE (Immediately Invoked Function Expression). It's a function that is called immediately after it's defined.

Syntax:

```
(function() {
    // function body
})();
```

We use IIFEs to create a new scope for our code so we can use variables without polluting the global scope.

Session 17

Mohamed Emary

May 24, 2024

1 Object

Object in JS is a collection of key-value pairs. Keys are also called properties and values can be any data type.

Objects are used to store multiple values in a single variable, these values are related to each other.

```
var student = {
    name: "Mohamed",
    age: 25,
    id: 12345,
    courses: ["Math", "Physics", "Chemistry"],
};
```

Objects are non-primitive data types, and when you try `console.log(typeof student)` it will return `object`.

Being a *non-primitive* data type means that the object can store **multiple values**, and these values can be of **different data types**.

Printing the object will show all the properties and their values.

```
console.log(student);
// Output:
// {
//   name: 'Mohamed',
//   age: 25,
//   id: 12345,
//   courses: [ 'Math', 'Physics', 'Chemistry' ]
// }
```

To access the properties of an object, you can use the `.` operator.

> Note:
>
> When you see the dot operator with any variable, it means that the variable is an object.

```
console.log(student.name); // Mohamed
console.log(student.age); // 25
```

When you try to access a property that doesn't exist, it will return `undefined`.

```
console.log(student.city); // undefined
student.city = "Cairo";    // Set a new property
console.log(student.city); // Get the new property
```

Now when you try to print the object, you will see the new property.

```
console.log(student);
// Output:
// {
//   name: 'Mohamed',
//   age: 25,
//   id: 12345,
//   courses: [ 'Math', 'Physics', 'Chemistry' ],
//   city: 'Cairo'
// }
```

You can put an object inside another object.

```
var student = {
    name: "Mohamed",
    age: 25,
    id: 12345,
    courses: ["Math", "Physics", "Chemistry"],
    address: {
        city: "Cairo",
        street: "Tahrir",
    },
};
```

To access the nested object, you can use the dot operator.

```
console.log(student.address.city);   // Cairo
console.log(student.address.street); // Tahrir
```

You can also put functions inside an object.

```
var student = {
    name: "Mohamed",
    age: 25,
    id: 12345,
    courses: ["Math", "Physics", "Chemistry"],
    address: {
        city: "Cairo",
        street: "Tahrir",
    },
    sayHello: function() {
        console.log("Hello, I'm a student");
    },
};
```

To call the function, you can use the dot operator.

If you don't use the parentheses `()`, it will return the function itself.

```
student.sayHello;
// Output:
// ƒ () {
//   console.log("Hello, I'm a student");
// }
```

To call the function, you should use the parentheses `()`.

```
student.sayHello(); // Hello, I'm a student
```

Using the `sayHello` function inside a `console.log` will print but also return `undefined` because the function doesn't return anything.

```
console.log(student.sayHello());
// Output:
// Hello, I'm a student
// undefined
```

But if you use a function like this:

```
var student = {
    name: "Mohamed",
    age: 25,
    id: 12345,
    courses: ["Math", "Physics", "Chemistry"],
    address: {
        city: "Cairo",
        street: "Tahrir",
    },
    sayHello: function() {
        return `Hello, I'm ${student.name}`;
    },
};
```

It will return the string without `undefined`.

```
console.log(student.sayHello());
// Output:
// Hello, I'm Mohamed
```

Note:

When you put a function inside an object, it's called a **method**.

2 JS Built-in Objects

JavaScript has many built-in objects like `window`, `document`, `console`.

2.1 `window`

`window` is a super global object that contains all global variables, functions, and objects.

Example functions in the `window` object are `alert`, `prompt`.

```
window.alert("Hello, World!");
window.prompt("What's your name?");
```

Since `window` is the super global object, you can access its properties and methods without using the `window` keyword.

This will also work:

```
alert("Hello, World!");
prompt("What's your name?");
```

2.2 `document`

`document` is another object that we can use to manipulate the HTML document.

```
document.getElementById("id");
```

> Note:
>
> Try this code:
> `document.getElementById("id").innerHTML = "Hello, World!";`
>
> Why this code works? isn't `getElementById` a method not an object?
> That is because the method `getElementById` returns an object so using the dot operator `.` allows you to access the properties of the object that the method returns.

2.3 `console`

`console` is also an object that has a method `log` that we use to print messages.

```
console.log("Hello, World!");
```

2.4 `Math`

`math` is an object that has many properties and methods to perform mathematical operations.

```
console.log(Math.PI); // 3.141592653589793
console.log(Math.round(4.7)); // 5
console.log(Math.floor(4.7)); // 4
console.log(Math.ceil(4.4)); // 5
console.log(Math.pow(2, 3)); // 8
console.log(Math.sqrt(64)); // 8
console.log(Math.abs(-4.7)); // 4.7
console.log(Math.min(0, 150, 30, 20, -8, -200)); // -200
console.log(Math.max(0, 150, 30, 20, -8, -200)); // 150
console.log(Math.random()); // Random number between 0 and 1
```

Mathematical expressions of the above methods:

- `Math.PI` $= \pi$
- `Math.round(4.7)` ≈ 5
- `Math.floor(4.7)` $= \lfloor 4.7 \rfloor = 4$
- `Math.ceil(4.4)` $= \lceil 4.4 \rceil = 5$

- `Math.pow(2, 3)` $= 2^3 = 8$
- `Math.sqrt(64)` $= \sqrt{64} = 8$
- `Math.abs(-4.7)` $= |-4.7| = 4.7$

```
console.log(Math.round(Math.random() * 10)); // Random number between 0 and
↪  10
```

Example to show a random number with a button click:

In HTML body:

```
  <div>
      <p id="rand"></p>
      <button id="btn">Get a random Number</button>
  </div>
  <script src="./script.js"></script>
```

In `script.js`:

```
var btn = document.getElementById("btn");
btn.onclick = getRand;
function getRand() {
  var p = document.getElementById("rand");
  p.innerHTML = Math.round(Math.random() * 10);
}
```

3 Array

Array is a collection of elements. Elements can be of any data type.

Array is a special type of object:

```
var nums = [1, 2, 3, 4, 5];
console.log(typeof nums); // object
```

And since objects are non-primitive data types, arrays can store multiple values of different data types.

Array syntax is a pair of square brackets `[]` with elements separated by commas.

```
var fruits = ["Apple", "Banana", "Orange"];
var mix = [
    [1, 2, 3],
    'Apple',
    25,
    true,
    function() { return "Hello, World!"; },
    { name: 'Mohamed', age: 25 }
];

console.log(mix[0][1]); // 2
console.log(mix[1]);    // Apple
console.log(mix[4]);    // [Function]
```

```
14 console.log(mix[4]());  // Hello, World!
15 console.log(mix[5].name); // Mohamed
```

Suppose you have multiple products and each of these products is an object, and you want to store all these products in one variable, you can use an array.

```
1 var products = [
2     { name: "Apple", price: 10 },
3     { name: "Banana", price: 5 },
4     { name: "Orange", price: 7 },
5 ];
```

Each item in the array has an index starting from 0 **(not 1)** and you can access the elements using that index.

```
1 var fruits = ["Apple", "Banana", "Orange"];
2 console.log(fruits[0]); // Apple
3 console.log(fruits[2]); // Orange
4 console.log(fruits);    // ["Apple", "Banana", "Orange"]
```

`fruits[0]` is read as "fruits of 0".

To print each item in the array, you can use a loop.

```
1 var fruits = ["Apple", "Banana", "Orange"];
2 for (var i = 0; i < fruits.length; i++) {
3     console.log(fruits[i]);
4 }
```

`fruits.length` returns the number of elements in the array, and since the last index is `fruits.length - 1`, the loop should run from `0` to `fruits.length - 1` so we use `i < fruits.length` instead of `i <= fruits.length` in the loop condition.

4 Object vs Array

	Object	Array
Type	Object & Non-primitive	Object & Non-primitive
Syntax	`{ prop1: val1, prop2: val2, ... }`	`[ elem1, elem2, ... ]`
Element Access	`object.prop`	`array[index]`
Use Case	Store different properties of an element	Store multiple elements
Index	Key	Number
Example	`{ name: 'Mohamed', age: 25 }`	`['Apple', 'Banana']`

See the execrise at the end of video 7: ***`object vs array & exercise`***

5 Functional Programming in JS

JavaScript applies functional programming concepts like:

1. You can assign a function to a variable.

```
var x = function() {
    return "Hello, World!";
};
```

2. Functions can be properties of objects.

```
var obj = {
    sayHello: function() { // method
        return "Hello there";
    },
};
```

3. Functions can be returned from another function.

```
function twoNumAvg(sum) {
    return sum / 2;
}

function getAvg(a, b) {
    var sum = a + b;
    return twoNumAvg(sum);
}

console.log(getAvg(10, 20)); // 15
```

4. Functions can be passed as arguments to other functions.

```
function twoNumAvg(sum) {
    return sum / 2;
}

function sum(a, b) {
    return a + b;
}

console.log(twoNumAvg(sum(10, 20))); // 15
```

6 Summary

Objects:

- Object is a collection of key-value pairs.
- Objects are used to store multiple values in a single variable.
- Objects are non-primitive data types.
- Objects can store multiple values of different data types.

Arrays:

- Array is a collection of elements.
- Array is a special type of object.
- Arrays can store multiple values of different data types.

Built-in Objects:

- `window` is a super global object.
- `document` is an object that we can use to manipulate the HTML document.
- `console` is an object that has a method `log` to print messages.
- `Math` is an object that has many properties and methods to perform mathematical operations.

Functional Programming:

- JavaScript applies functional programming concepts.
    - Functions can be assigned to a variable.
    - Functions can be properties of objects.
    - Functions can be returned from another function.
    - Functions can be passed as arguments to other functions.

Session 18

Mohamed Emary

May 27, 2024

1 Built-in Array Methods

1.1 `push`

Suppose you have this array:

```
var colors = ["red", "orange", "yellow"];
```

And you want to add another color to the end of the array. You can use this way:

```
colors[3] = "green";
console.log(colors); // ["red", "orange", "yellow", "green"]
```

However, there is a better way to do this. You can use the `push` method:

```
var colors = ["red", "orange", "yellow"];
colors.push("green");
console.log(colors); // ["red", "orange", "yellow", "green"]
```

The `push` method is an array method that adds an element to the end of an array, and returns the new length of the array.

You can also use `push` to add multiple elements to an array:

```
var colors = ["red", "orange"];
var newLen = colors.push("blue", "indigo", "violet");
console.log(colors); // ["red", "orange", "blue", "indigo", "violet"]
console.log(newLen);    // 5
```

1.2 `unshift`

`unshift` is similar to `push`, but it adds an element to the beginning of an array instead of the end, and returns the new length of the array.

```
var colors = ["red", "orange", "yellow"];
var newLen = colors.unshift("green");
console.log(colors); // ["green", "red", "orange", "yellow"]
console.log(newLen);    // 4
```

You can also use `unshift` to add multiple elements to an array:

```
var colors = ["red", "orange"];
colors.unshift("blue", "indigo", "violet");
console.log(colors); // ["blue", "indigo", "violet", "red", "orange"]
```

1.3 pop

`pop` removes the last element from an array and returns that element.

```
var colors = ["red", "orange", "yellow"];
var removed = colors.pop();
console.log(removed); // "yellow"
console.log(colors);  // ["red", "orange"]
```

1.4 shift

`shift` removes the first element from an array and returns that element.

```
var colors = ["red", "orange", "yellow"];
var removed = colors.shift();
console.log(removed); // "red"
console.log(colors);  // ["orange", "yellow"]
```

Hovering on functions

When you hover with the mouse cursor over a function in VS Code, you can see a brief description of what the function does. This can be helpful if you're not sure what a function does or how to use it.

To know the length of an array, you can use the `length` ***property***:

> It's a property, because arrays are objects, and `length` is a property of the array object.

```
var colors = ["red", "orange", "yellow"];
console.log(colors.length); // 3
```

`length` is always higher than the highest index in the array by 1 since it starts from 1 and index starts from 0, so if you have an array with 3 elements, the highest index is 2, and the length is 3.

1.5 splice

`splice` can add or remove elements from an array, and returns the removed elements.

It takes three arguments:

1. The index at which to start changing the array.
2. The number of elements to remove.
3. The element***s*** to add.

So this is its syntax:

```
array.splice(start, deleteCount, item1, item2, ...)
```

Example on deleting elements with `splice`:

```
var colors = ["red", "orange", "yellow", "green", "blue"];
// To delete yellow and green
var removed = colors.splice(2, 2); // From index 2, delete 2 elements
console.log(colors);  // ["red", "orange", "blue"]
console.log(removed); // ["yellow", "green"]

colors = ["red", "orange", "yellow", "green", "blue"];
// To delete all elements starting from index 2
removed = colors.splice(2);  // From index 2, delete all elements
console.log(colors);  // ["red", "orange"]
console.log(removed); // ["yellow", "green", "blue"]

colors = ["red", "orange", "yellow", "green", "blue"];
// This have no effect on the array
removed = colors.splice(2, 0);  // From index 2, delete 0 elements
console.log(colors);  // ["red", "orange", "yellow", "green", "blue"]
```

Example on adding elements with `splice`:

```
var colors = ["red", "orange"];
// To add yellow and black
colors.splice(1, 0, "yellow", "black"); // From index 1, delete 0
 ↪  elements, add "yellow" and "black"
console.log(colors); // ["red", "yellow", "black", "orange"]


colors = ["red", "orange", "yellow"];
// To add green and blue and remove orange
colors.splice(2, 1, "green", "blue"); // From index 2, delete 1 element,
 ↪  add "green" and "blue"
console.log(colors); // ["red", "orange", "green", "blue"]
```

1.6 `slice`

`slice` takes two arguments: the start index and the end index (*not included*), and returns a new array with the elements between the start and end indexes.

Not included means that the last index is not included in the new array.

With `slice`, the original array is not affected.

```
var colors = ["red", "orange", "yellow", "green", "blue"];
var subColors = colors.slice(1, 3); // From index 1 to 3 (not included)
console.log(subColors); // ["orange", "yellow"]
```

1.7 `includes`

`includes` checks if an array includes a certain element, and returns `true` or `false`.

```
var colors = ["red", "orange", "yellow"];
console.log(colors.includes("yellow")); // true
console.log(colors.includes("purple")); // false
```

You can also specify a starting index from which to start searching:

```
var colors = ["red", "orange", "yellow", "green", "blue"];
console.log(colors.includes("orange", 2)); // false
```

Here the color `"orange"` already exists in the array, but since we specified the starting index as `2`, it starts searching from index `2`, and `"orange"` is at index `1`, so it returns `false`.

1.8 `indexOf` & `lastIndexOf`

`indexOf` returns the first index at which a given element can be found in the array, or `-1` if it is not present.

```
var colors = ["red", "orange", "yellow", "green", "blue"];
console.log(colors.indexOf("green")); // 3
console.log(colors.indexOf("purple")); // -1
```

If the array contains two similar elements, `indexOf` returns the index of the first one:

```
var colors = ["orange", "red", "yellow", "red"];
console.log(colors.indexOf("red")); // 1
```

What if you want to get the index of the last occurrence of an element? You can use `lastIndexOf`:

```
var colors = ["orange", "red", "yellow", "red"];
console.log(colors.indexOf("red")); // 1
console.log(colors.lastIndexOf("red")); // 3
```

Other Methods

There are many other methods that you can use with arrays. You can find the full list of array methods in the MDN Web Docs.

Some methods like `map`, `filter`, `reduce`, and `forEach` were introduced in ES6, and they are very useful when working with arrays. We will cover them in the next sessions.

2 Exercise

Try to find how many times a certain element appears in an array.

```
function getOccurrences(array, searchElement) {
  var indices = [];
  if (array.includes(searchElement)) {
    for (var i = 0; i < array.length; i++) {
      if (array[i] === searchElement) {
        indices.push(i);
      }
    }
    return indices;
  } else {
    return 0;
  }
}

```

```
var numbers = [1, 2, 3, 4, 1, 1, 1, 1];
console.log(getOccurrences(numbers, 1)); // [0, 4, 5, 6, 7]
```

3 CRUD Operations

CRUD stands for ***C***reate, ***R***ead, ***U***pdate, and ***D***elete. These are the four basic operations that can be performed on data. You can also add ***S*** for ***S***earch.

Any software application that works with data usually performs these operations.

When working making your website always finish the design first (the HTML and CSS), then add the functionality (JavaScript).

Also when solving a problem using JS divide it into smaller tasks, and solve each task separately.

Important:

Watch the CRUD example system videos from video 4 to 9.

When getting an element from the DOM using `document.getElementById("id")`, you can use `console.log(element)` it to make sure you got the right element.

When JS deals with HTML it converts the HTML tags to objects each with its own properties and methods and each attribute in the HTML tag is a property in the object, and you can manipulate these objects using JS.

```
var addBtn = document.getElementById("addBtn");

addBtn.onclick = addProduct; // IMPORTANT: Don't add () after the function
 ↪  name

function addProduct() {
  var prodName = document.getElementById("prodName").value;
  console.log(prodName);
};
```

In this code we assigned the `addProduct` function to the `onclick` event of the `addBtn` button, and we didn't add `()` after the function name, because we don't want to call the function immediately, we want to call it when the button is clicked.

The `addProduct` function gets the value of the input with the id `prodName` and logs it to the console.

Important Note:

You shouldn't put this line before the function:
`var prodName = document.getElementById("prodName").value;`
Because the input field will not have a value when the page loads, and you want to get the value when the button is clicked, as the user will press it after typing the product name.

4 Summary

In this session we learned about the following array methods:

- `push` adds an element to the end of an array, and returns the new length of the array.
- `unshift` adds an element to the beginning of an array, and returns the new length of the array.
- `pop` removes the last element from an array, and returns that element.
- `shift` removes the first element from an array, and returns that element.
- `splice` can add or remove elements from an array, and returns the removed elements.
- `slice` returns a new array with the elements between the start and end indexes.
- `includes` checks if an array includes a certain element, and returns `true` or `false`.
- `indexOf` returns the first index at which a given element can be found in the array, or `-1` if it is not present.
- `lastIndexOf` returns the index of the last occurrence of an element in an array.

We also learned about CRUD operations, and how to manipulate the DOM using JavaScript.

Session 19

Mohamed Emary

May 31, 2024

1 Local Storage

Local storage is a way to store data in the browser (client-side storage). It is a key-value pair storage limited storage (5MB).

When we say it's Local Storage, it means it's local to the browser. It is not stored on the server. It is stored on the client's machine, so it is not shared with other users.

To see the local storage in the browser, open the developer tools and go to the Application tab. Then, click on Local Storage.

You can *only store strings* in the local storage.

To store a value in the local storage, you can use the `setItem()` method. The `setItem()` method takes two parameters: the key and the value.

```
localStorage.setItem('name', 'Mohamed');
```

Keys are unique. If you set a value to a key that already exists, it will overwrite the old value.

```
localStorage.setItem('name', 'Ahmed');
```

Now the value of the key `name` is `Ahmed`.

To get a value from the local storage, you can use the `getItem()` method. The `getItem()` method takes one parameter: the key.

```
var name = localStorage.getItem('name');
console.log(name); // Ahmed
```

To remove a value from the local storage, you can use the `removeItem()` method. The `removeItem()` method takes one parameter: the key.

```
localStorage.removeItem('name');
var name = localStorage.getItem('name');
console.log(name); // null
```

To know how many items are stored in the local storage, you can use the `length` property.

To clear the local storage, you can use the `clear()` method. The `clear()` method takes no parameters.

```
localStorage.setItem('name', 'Mohamed');
localStorage.setItem('age', '25');
console.log(localStorage.length); // 2
localStorage.clear();
var name = localStorage.getItem('name');
var age = localStorage.getItem('age');
console.log(name); // null
console.log(age);  // null
```

To know which key at a specific index, you can use the `key()` method. The `key()` method takes one parameter: the index.

```
localStorage.setItem('name', 'Mohamed');
localStorage.setItem('age', '25');
console.log(localStorage.key(0)); // name
console.log(localStorage.key(1)); // age
```

You shouldn't store sensitive data in the local storage because it's not secure. It's accessible by anyone who has access to the client's machine.

We don't get all data from backend some data that are not sensitive like language. can be stored in the local storage.

Local storage data are not removed even if you close the browser. It will be removed when you clear the local storage or when you delete the browser's data.

2 Session Storage

Session storage is similar to local storage, but it's for the session only which means it's removed when the session is ended like when you close the tab or the browser.

We have a method called `sessionStorage` that works the same as `localStorage` with the same methods and properties like:

- `setItem()`
- `getItem()`
- `removeItem()`
- `length`
- `clear()`
- `key()`

3 Storing Objects

As we mentioned before, you can only store strings in the local storage. If you want to store an object, you need to convert it to a string using `JSON.stringify()`.

```
var person = {
  name: 'Mohamed',
  age: 25
};

localStorage.setItem('person', JSON.stringify(person));
```

To get the object from the local storage, you need to parse the string using `JSON.parse()`.

```
var person = JSON.parse(localStorage.getItem('person'));
console.log(person.name); // Mohamed
console.log(person.age);  // 25
```

The same can be done with arrays:

```
var people = [
  { name: 'Mohamed', age: 25 },
  { name: 'Ahmed', age: 30 },
  { name: 'Ali', age: 35 }
];

localStorage.setItem('people', JSON.stringify(people));
var people = JSON.parse(localStorage.getItem('people'));
console.log(people[1]); // { name: 'Ahmed', age: 30 }
```

4 Accepting Image As Input

With the input element where the user can select an image, you will specify the type as `file` you can also specify the `accept` attribute to specify the type of files that the user can select, for example, `image/png`, `image/jpeg`, or `image/*` to accept all image types, and you can also use the attribute `multiple` to allow the user to select multiple files.

```
<input type="file" accept="image/*" id="imgInput" />
<button id="upload">Upload</button>
```

This will create an input field that accepts all image types.

In your JavaScript code when you `console.log` the value of the file input element, you will get a `C:\fakepath\` followed by the image file name, so for example if your image file name is `my_image.jpg` the console output will be `C:\fakepath\my_image.jpg`

```
var imgInput = document.getElementById('imgInput');
console.log(imgInput.value); // C:\fakepath\my_image.jpg
```

This `C:\fakepath\` is a browser standard that doesn't depend on the operating system and it's used by the browser with any file the user uploads not just images. This is done for security reasons to prevent the website from knowing the user's file system structure.

For example if the real file path was `C:\Users\Ahmed\TopSuperSecretProject\VeryImportantImg.png`, then by uploading it you'd be exposing that your real name is Ahmed and you're working on TopSuperSecretProject which is a security risk.

Since `C:\fakepath\` is a browser standard, you can see it in any operating system even those with no `C:\` partition like macOS or Linux.

So how can you display the image?

You can get the file object from the input element using the `files` property. The `imgInput.files` is a FileList object that contains the multiple files the user selected in case the input element has the `multiple` attribute. If the input element doesn't have the `multiple` attribute, then you can access the one file using `imgInput.files[0]`.

You can access the file name using `name` property.

```
var imgInput = document.getElementById('imgInput');
console.log(imgInput.files[0].name); // my_image.jpg
```

To display the image we get the file object from the input element, then we use the `createObjectURL()` method to create a URL for the file object, then we can use that URL to display the image in the browser using the `src` attribute of an image element.

Consider this example:

In HTML:

```
<input type="file" accept="image/*" id="imgInput" />
<button id="upload">Upload</button>
<img id="img" />
```

In JavaScript:

```
var imgInput = document.getElementById('imgInput');
var upload = document.getElementById('upload');
var img = document.getElementById('img');
upload.onclick = function() {
  var file = imgInput.files[0];
  if (file) {
    var objectURL = URL.createObjectURL(file);
    // set the src attribute of the image element to the object URL
    img.src = objectURL;
  }
};
```

This is how the page will look like:

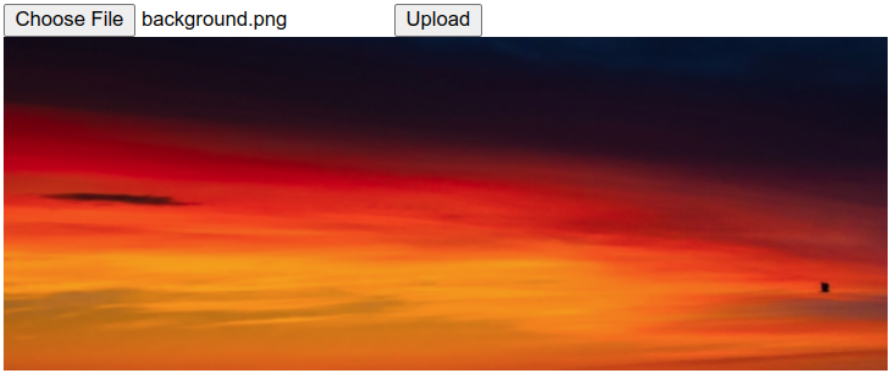

Figure 1: Image Upload

5 String Methods

Strings have many methods that you can use to manipulate strings. Here we will discuss some of the most common methods.

5.1 `charAt()`, `[]`, `at()`

The `charAt()` method returns the character at a specified index (position) in a string.

```
var str = 'Hello, World!';
console.log(str.charAt(0)); // H
console.log(str.charAt(7)); // W
```

You can also use square brackets `[]` to access the character at a specific index.

```
var str = 'Hello, World!';
console.log(str[0]); // H
console.log(str[7]); // W
```

The `at()` method returns the character at a specified index (position) in a string, but it also *supports negative indexes*.

```
var str = 'Hello, World!';
console.log(str.at(0)); // H
console.log(str.at(7)); // W
console.log(str.at(-1)); // !
console.log(str.at(-3)); // l
```

5.2 `slice()`

The `slice()` method extracts a part of a string and returns a new string.

The `slice()` method takes two parameters: the start index and the end index. The `slice()` method extracts up to *but not including the end index*.

If you don't specify the end index, the `slice()` method will extract to the end of the string.

The `slice()` method also supports negative indexes.

Syntax:

```
string.slice(start, end(optional))
```

Example:

```
var str = 'Hello, World!';
console.log(str.slice(3, 6)); // lo,
console.log(str.slice(3)); // lo, World!
console.log(str.slice(-6, -1)); // World
console.log(str.slice(-6)); // World!
```

5.3 `substring()`

The `substring()` method extracts the characters in a string between two specified indices.

The `substring()` method takes two parameters: the start index and the end index.

The `substring()` method is similar to the `slice()` method, but it doesn't support negative indexes.

Syntax:

```
string.substring(start, end(optional))
```

Example:

```
var str = 'Hello, World!';
console.log(str.substring(3, 6)); // lo,
console.log(str.substring(3)); // lo, World!
```

5.4 toUpperCase(), toLowerCase()

The `toUpperCase()` method converts a string to uppercase letters.

The `toLowerCase()` method converts a string to lowercase letters.

Example:

```
var str = 'Hello, World!';
console.log(str.toUpperCase()); // HELLO, WORLD!
console.log(str.toLowerCase()); // hello, world!
```

5.5 toLocaleUpperCase(), toLocaleLowerCase()

The `toUpperCase()` and `toLowerCase()` methods in JavaScript convert a string to uppercase and lowercase respectively, without considering the locale of the user's environment.

On the other hand, `toLocaleUpperCase()` and `toLocaleLowerCase()` methods also convert a string to uppercase and lowercase respectively, but they take into account the locale of the user's environment. This means that they respect the language rules for casing.

For example, in Turkish, the lowercase `I` is `ı` and the uppercase `i` is `İ`. The `toUpperCase()` and `toLowerCase()` methods do not handle this correctly, while `toLocaleUpperCase()` and `toLocaleLowerCase()` do.

Here's an example:

```
let str = 'i';
console.log(str.toUpperCase()); // I
console.log(str.toLocaleUpperCase('tr-TR')); // İ

str = 'I';
console.log(str.toLowerCase()); // i
console.log(str.toLocaleLowerCase('tr-TR')); // ı
```

The output of both `toUpperCase()` and `toLowerCase()` is wrong for the Turkish language, while the output of both `toLocaleLowerCase()` and `toLocaleUpperCase()` is correct.

5.6 includes()

The `includes()` method checks if a string contains a specified value.

The `includes()` method returns `true` if the string contains the specified value, otherwise it returns `false`.

Syntax:

```
string.includes(searchValue, start(optional))
```

Example:

```
var str = 'Hello, World!';
console.log(str.includes('Hello')); // true
```

```
console.log(str.includes('hello')); // false
console.log(str.includes('Hello', 0)); // true
console.log(str.includes('Hello', 1)); // false
console.log(str.includes('')); // true (empty string is always included)
```

5.7 concat()

The `concat()` method concatenates two or more strings and returns a new string.

Syntax:

```
string.concat(string1, string2, ..., stringN)
```

Example:

```
var str1 = 'Hello ';
var str2 = 'JS ';
var str3 = 'and ';
var str4 = 'the World!';
console.log(str1.concat(str2, str3, str4)); // Hello JS and the World!
```

5.8 trim(), trimStart(), trimEnd()

The `trim()` method removes whitespace from both ends of a string.

The `trimStart()` method removes whitespace from the beginning of a string.

The `trimEnd()` method removes whitespace from the end of a string.

Example:

```
var str = '   Hello, World!   ';
console.log(str.trim());      // 'Hello, World!'
console.log(str.trimStart()); // 'Hello, World!   '
console.log(str.trimEnd());   // '   Hello, World!'
```

5.9 split()

The `split()` method splits a string into an array of substrings.

The `split()` method takes two parameters: the separator and the limit.

The `split()` method splits the string at each occurrence of the separator.

If you don't specify the limit, the `split()` method will split the string into all substrings.

Syntax:

```
string.split(separator, limit(optional))
```

Example:

```
var str = 'Hello JS and the World!';
console.log(str.split(' '));    // ['Hello', 'JS', 'and', 'the', 'World!']
console.log(str.split(' ', 2)); // ['Hello', 'JS']
console.log(str.split(''));     // ['H', 'e', 'l', ..., '!']
console.log(str.split(' ', 0)); // []
```

```
console.log(str.split('', 3));  // ['H', 'e', 'l']
console.log(str.split('and'));  // ['Hello JS ', ' the World!']
```

5.10 join()

If you have an array of strings and you want to join them into a single string, you can use the `join()` method.

Syntax:

```
array.join(separator)
```

Example:

```
var arr = ['Hello', 'JS', 'and', 'the', 'World!'];
console.log(arr.join(' ')); // Hello JS and the World!
console.log(arr.join(''));  // HelloJSandtheWorld!
console.log(arr.join());    // Hello,JS,and,the,World!
console.log(arr.join(',')); // Hello,JS,and,the,World!
```

From the last two lines we can see that if we don't specify the separator, the default separator is a comma.

Example on using `split()` with `slice()` and `join()`:

```
var str = 'Hello JS and the World!';
var res = str.split(' ').slice(1, 4).join('-');
console.log(res); // JS-and-the
```

The result of `split(' ')` is `['Hello', 'JS', 'and', 'the', 'World!']`, then we use `slice(1, 4)` to get the elements from index 1 to index 3 (not including index 4) which are `['JS', 'and', 'the']`, then we use `join('-')` to join them with a hyphen `-` to get `JS-and-the`.

5.11 repeat()

The `repeat()` method returns a new string with a specified number of copies of an existing string.

Syntax:

```
string.repeat(count)
```

Example:

```
var str = 'Hello, World!';
console.log(str.repeat(3)); // Hello, World!Hello, World!Hello, World!
console.log(str.at(-1).repeat(3)); // !!!
```

5.12 replace(), replaceAll()

The `replace()` method searches a string for a specified value, or a regular expression, and returns a new string where the specified values are replaced.

The `replace()` method takes two parameters: the value to search for, and the value to replace it with.

The `replace()` method only replaces the *first occurrence* of the specified value.

The `replaceAll()` method is similar to the `replace()` method, but it replaces all occurrences of the specified value.

Syntax:

```
string.replace(searchValue, replaceValue)
```

Example:

```
var str = 'HTML and CSS and JS';
console.log(str.replace('and', 'AND')); // HTML AND CSS and JS
console.log(str.replaceAll('and', 'AND')); // HTML AND CSS AND JS
```

5.13 `padStart()`, `padEnd()`

The `padStart()` method pads a string with another string until the resulting string reaches the specified length.

The `padEnd()` method pads a string with another string until the resulting string reaches the specified length.

Syntax:

```
string.padStart(targetLength, padString(optional))
string.padEnd(targetLength, padString(optional))
```

Example:

```
var str = '99';
console.log(str.padStart(10)); // '      99'
console.log(str.padEnd(10));   // '99      '
console.log(str.padStart(10, '0')); // '0000000099'
console.log(str.padEnd(10, '0'));   // '9900000000'
```

6 Searching in CRUD System

There is two types of search:

1. Real-time search: The search is done while the user is typing, it provides a better user experience but also comes with a performance cost.
2. Search button: The search is done when the user clicks on the search button.

6.1 Real-time Search

You can handle this by using the `keyup` event which is triggered when the user releases a key, you can also use the `input` event which is better and triggered when the value of the input element changes this is better because not all keys change the value of the input element like the arrow keys or the control keys.

HTML:

```
<input type="text" id="search" />
```

JavaScript:

```
var search = document.getElementById('search');
search.oninput = function() {
  console.log(search.value);
};
```

This will log the value of the input element whenever the user changes it.

6.2 Search Button

You can handle this by using the `onclick` event which is triggered when the user clicks on the search button.

HTML:

```
<input type="text" id="search" />
<button id="searchBtn">Search</button>
```

JavaScript:

```
var search = document.getElementById('search');
var searchBtn = document.getElementById('searchBtn');
searchBtn.onclick = function() {
  console.log(search.value);
};
```

This will log the value of the input element whenever the user clicks on the search button.

6.3 Example of Real-time Search CRUD System

This is an example of a real-time search in a CRUD system where the user inputs some product names and can search for them in real-time.

For simplicity, the JS code is written in the `<script>` tag of the HTML file:

In HTML:

```
<h2>Product Management</h2>
<input type="text" id="productName" placeholder="Enter product name" />
<button onclick="addProduct()">Add Product</button>
<h2>Product List</h2>
<ul id="productList"></ul>
<h2>Search Product</h2>
<input
  type="text"
  id="searchProduct"
  placeholder="Search product"
  oninput="searchProduct()" />
```

In JavaScript:

```
var products = [];

function addProduct() {
  var productName = document.getElementById("productName");
  if (productName.value) { // check if the input is not empty
    products.push(productName.value);
```

```
    productName.value = "";
    displayProducts();
  }
}

function displayProducts() {
  var productList = document.getElementById("productList");
  productList.innerHTML = "";
  for (var i = 0; i < products.length; i++) {
    productList.innerHTML += `<li> ${products[i]} </li>`;
  }
}

function searchProduct() {
  var searchValue =
  ↪  document.getElementById("searchProduct").value.toLowerCase();
  var productList = document.getElementById("productList");
  productList.innerHTML = "";
  for (var i = 0; i < products.length; i++) {
    // toLowerCase() is used to make the search case-insensitive
    if (products[i].toLowerCase().includes(searchValue)) {
      productList.innerHTML += `<li> ${products[i]} </li>`;
    }
  }
}
```

This is how the page will look:

Figure 2: After Adding All Products

Figure 3: While Searching...

You may notice that all elements appear when the search input is empty, this is because when the search input is empty, the `searchValue` is an empty string which is included in all strings.

Code Link to try it yourself.

7 Summary

Local Storage

- Local storage is a way to store data in the browser with a maximum storage of 5MB.
- It's limited to the browser and not shared with other users or the server.
- You can only store strings in local storage.
- Data is not removed when you close the browser tab, but it's removed when you clear the local storage or browser data.

Session Storage

- Session storage is similar to local storage but data is removed when the session is ended (e.g. closing the tab or browser).
- The same methods and properties are used to work with session storage as local storage.

Storing Objects

- To store objects in local storage, you need to convert them to strings using `JSON.stringify()` and convert them back from JSON using `JSON.parse()` when retrieving them.

Accepting Image As Input

- With an input element of type `file`, you can specify what file types the user can select using the `accept` attribute.
- You can also specify if the user can select multiple files using the `multiple` attribute.
- To get the file name, you can use the `files` property of the input element.
- To display the image, you can get the file object from the input element and use the `createObjectURL()` method to create a URL for the file object.

String Methods

- This section covers common string methods including:
  - `charAt()` - returns the character at a specified index.
  - `slice()` - extracts a part of a string and returns a new string.
  - `substring()` - similar to slice but doesn't support negative indexes.
  - `toUpperCase()` - converts a string to uppercase letters.
  - `toLowerCase()` - converts a string to lowercase letters.
  - `includes()` - checks if a string contains a specified value.
  - `concat()` - concatenates two or more strings.
  - `trim()` - removes whitespace from both ends of a string.
  - `split()` - splits a string into an array of substrings.
  - `join()` - joins an array of strings into a single string.
  - `repeat()` - returns a new string with a specified number of copies of an existing string.
  - `replace()` - searches a string for a specified value and replaces it with another value.

- `padStart()` - pads a string with another string to a specified length from the left side.
- `padEnd()` - pads a string with another string to a specified length from the right side.

Searching in CRUD System

- There are two types of search: real-time search and search with a button.
- Real-time search is done while the user is typing using the `input` event.
- Search with a button is done when the user clicks on a search button using the `onclick` event.
- The provided code shows an example of a real-time search for products in a CRUD system.

Session 20

Mohamed Emary

June 3, 2024

1 DOM (Document Object Model)

When the browser loads an HTML document, it creates a tree-like structure in memory. This structure is called the Document Object Model (DOM). The DOM represents the document as nodes and objects, allowing you to interact with the document using JavaScript.

For example if you have an `<img />` tag in your HTML, this tag will be represented as an object in the DOM, and its attributes (e.g., `src`, `alt`, `width`, `height`) will become properties of this object and can be accessed and modified using JavaScript.

But what is the difference between HTML and DOM? In short, HTML represents the *initial page content* and the DOM (Document Object Model) represents the *current content* in a tree of objects. If you have a html page and add a tag with javascript, the actual HTML of the page is still the same, but the "DOM" however has changed.

2 Selecting Elements in the DOM

Suppose you have this HTML element:

```
<div id="myElement" class="myClass"></div>
```

To select that element using JavaScript there are several ways to do that. Here are some common methods:

1. `getElementById`: This method returns the element with the specified ID. (Note: IDs must be unique within the document.)

```
var element = document.getElementById('myElement');
```

2. `getElementsByClassName`: This method returns a *collection* of elements with the specified class name.

```
var elements = document.getElementsByClassName('myClass');
```

3. `getElementsByTagName`: This method returns a *collection* of elements with the specified tag name.

```
var elements = document.getElementsByTagName('div');
```

4. `getElementsByName`: This method returns a *node list* of elements with the specified name attribute.

```
var elements = document.getElementsByName('myName');
```

5. `querySelector`: This method returns *the first element* that matches the specified CSS selector.

```
var element = document.querySelector('.myClass');
```

6. `querySelectorAll`: This method returns a *node list* of elements that match the specified CSS selector.

```
var elements = document.querySelectorAll('.myClass');
```

`getElementsByClassName`, `getElementsByTagName` return a *collection* of elements called `HTMLCollection`, which is an *array-like object*. If you want to access a specific element, you can use the index like `elements[0]`.

Since `HTMLCollection` is not an actual array, you can't use array methods like `join`, `push`, `pop`, etc. To convert it to an array, you can use the `Array.from` method.

```
var elements = document.getElementsByClassName ('myClass');
var elementsArray = Array.from(elements);
```

Now you can use array methods on `elementsArray`.

`getElementsByName`, `querySelectorAll` return a `NodeList`, which is also an array-like object that you can loop through and access elements by index and you can't use array methods on it. You can also convert it to an array using `Array.from`.

The difference between `HTMLCollection` and `NodeList` is that `NodeList` is a list of nodes, not just elements. For example, it can contain text nodes, comment nodes, etc. While `HTMLCollection` only contains elements.

There is some elements that are built-in properties of the `document` object:

1. `document.documentElement`: Returns the `<html>` element.
2. `document.head`: Returns the `<head>` element.
3. `document.body`: Returns the `<body>` element.
4. `document.title`: Returns the title of the document.
5. `document.images`: Returns a collection of all `<img>` elements in the document.
6. `document.links`: Returns a collection of all `<a>` elements with a `href` attribute in the document.
7. `document.forms`: Returns a collection of all `<form>` elements in the document.
8. `document.scripts`: Returns a collection of all `<script>` elements in the document.
9. `document.styleSheets`: Returns a collection of all `<link>` and `<style>` elements that have a `rel` attribute with the value `stylesheet`.

3 Event Listeners

Event listeners are used to listen for events on a specific element and execute a JavaScript function when that event occurs. You can add event listeners to any element in the DOM.

Syntax:

```
element.addEventListener(event, function);
```

- `element`: The element to attach the event listener to.
- `event`: The event to listen for (e.g., `click`, `mouseover`, `keydown`, etc.).
- `function`: The function to execute when the event occurs.

This way of adding event listeners is better than using the `onEvent` attribute in the HTML because it allows you to add multiple event listeners to the same element and separate the JavaScript code from the HTML.

Here is an example of adding an event listener to a button element:

In HTML:

```
<button id="myButton">Click me</button>
```

In JavaScript:

```
function sayHello() {
  console.log('Hello!');
}

var button = document.getElementById('myButton');
// Don't use () after function name
button.addEventListener('click', sayHello);
```

Using `()` after the function name will execute the function immediately once the event listener is added. You should only pass the function name without `()`.

But what if that function has parameters? You can use an anonymous function to pass the parameters:

```
function sayHello(name) {
  console.log('Hello, ' + name + '!');
}

button.addEventListener('click', function() {
  sayHello('John');
});
```

The same applies to `element.event` way of adding event listeners:

```
function sayHello() {
  console.log('Hello!');
}
button.onclick = sayHello;
```

And if the function has parameters:

```
function sayHello(name) {
  console.log('Hello, ' + name + '!');
}
button.onclick = function() {
  sayHello('John');
};
```

What is the difference between `addEventListener` and `element.event`? The main difference is that `addEventListener` allows you to add multiple event listeners to the same element, while `element.event` can only have one event listener per event type.

```
var button = document.getElementById('myButton');
button.onclick = function() {
  console.log('Hello!');
};

button.onclick = function() {
  console.log('Goodbye!');
};
```

In this example, only the second event listener will be executed because the first one will be overwritten.

```
var button = document.getElementById('myButton');
button.addEventListener('click', function() {
  console.log('Hello!');
});

button.addEventListener('click', function() {
  console.log('Goodbye!');
});
```

In this example, both event listeners will be executed in the order they were added.

4 Event Object

When an event occurs, the browser creates an event object that contains information about the event. This object is passed as an argument to the event listener function.

Here is an example of using the event object to get information about a click event:

```
var button = document.getElementById('myButton');
button.addEventListener('click', function(event) {
  console.log(event);
});
```

The event object contains information such as:

- `type`: The type of event (e.g., `click`, `dblclick`, `mouseover`, `keydown`, etc.).
- `target`: The element that triggered the event.
- `clientX`, `clientY`: The coordinates of the mouse pointer when the event occurred.
- `keyCode`: The key code of the key that was pressed (for keyboard events).

Example using the event object with the whole document:

```
document.addEventListener('keydown', function(event) {
  console.log(event.keyCode);
});

document.addEventListener('click', function(event) {
  console.log(event.clientX, event.clientY);
  console.log(event.target);
  console.log(event.type);
});
```

5 Some Common Events

Here are some common events that you can listen for:

Mouse Related Events:

- `click`: The user clicks an element.
- `dblclick`: The user double-clicks an element.
- `mousemove`: The user moves the mouse.
- `mouseenter`: The user moves the mouse over an element.
- `mouseleave`: The user moves the mouse out of an element.
- `mouseup`: The user releases a mouse button.
- `mousedown`: The user presses a mouse button.
- `mouseover`: The user moves the mouse over an element.
- `mouseout`: The user moves the mouse out of an element.
- `scroll`: The user scrolls the page.
- `drag`: The user is dragging an element. (Note: you should add `draggable="true"` to the element HTML code to make it draggable.)
- `dragstart`: The user starts dragging an element.
- `dragend`: The user stops dragging an element.

Keyboard Related Events:

- `keyup`: The user releases a key on the keyboard.
- `keydown`: The user presses a key on the keyboard.
- `keypress`: The user presses a key on the keyboard.

Input and Form Related Events:

- `input`: The user inputs text into an input element.
- `change`: The user focuses out of an input element after changing its value.
- `submit`: The user submits a form. This for example can be used to prevent the page from reloading when submitting a form using `event.preventDefault()` (`event` here is the event object passed to the event listener function not the event type).

Focus Related Events:

- `focus`: The user focuses on an input element.
- `blur`: The user focuses out of an input element.

There are many more events that you can listen for. You can find a complete list of events in the MDN Web Docs.

6 Changing Element Styles

You can change the style of an element using JavaScript by accessing its `style` property. This property contains all the CSS properties of the element.

Here is an example of changing the background color of a `div` element:

```
var element = document.getElementById('myElement');
element.style.backgroundColor = 'red';
```

You can also change multiple styles at once using the `cssText` property:

```
element.style.cssText = 'background-color: red; color: white; font-size:
↪  20px;';
```

These properties are useful for changing styles dynamically based on user interactions or other events.

The styles applied using the `style` property are inline styles, which have the highest specificity and override any other styles defined in external CSS files or internal styles except ones with `!important`.

When a style has `!important` and you want to override it using JavaScript, you can use the `cssText` property with `!important`:

```
element.style.cssText = 'background-color: red !important;';
```

6.1 Example of Making an Element Draggable

To make an element draggable, you need to add the `draggable` attribute to the element and set it to `true`. You can then listen for the `dragend` event to get the mouse coordinates and move the element to that position.

Here is an example of making a `div` element draggable:

In HTML:

```
<div id="myElement" draggable="true">Drag me</div>
```

In JavaScript:

```
var element = document.getElementById("myElement");
element.style.cssText = `
  width: 100px;
  height: 100px;
  line-height: 100px;
  text-align: center;
  background-color: gold;`;
element.addEventListener("dragend", function (event) {
```

```
  element.style.position = "absolute";
  element.style.left = event.clientX + "px";
  element.style.top = event.clientY + "px";
  element.style.transform = "translate(-50%, -50%)";
});
```

In this code we applied some styles to the element using the `cssText` property, then we listened for the `dragend` event to get the mouse coordinates and move the element to that position.

Don't forget to add `px` after because the `clientX` and `clientY` properties return the mouse coordinates in pixels but the unit is not specified so you need to add `px` after the value.

In the example above you can also use `mousemove` event instead of `dragend` to move the element while dragging it but that will make you not able to drop it in a new position. You can for example use that to make a simple icon that is always following the mouse cursor or to make a simple drawing app where you draw by dragging the mouse.

7 Get, Set, and Remove Attributes

You can use the `setAttribute` method to set an attribute of an element and the `getAttribute` method to get the value of an attribute.

Here is an example of setting and getting the `src`, `alt` attributes of an `img` element:

```
var img = document.getElementById('myImage');
img.setAttribute('src', 'image.jpg');
img.setAttribute('alt', 'My Image');

var src = img.getAttribute('src');
var alt = img.getAttribute('alt');
console.log(src); // Output: image.jpg
console.log(alt); // Output: My Image
```

You can also use these methods to set and get styles:

```
<div id="myElement" style="background-color: red;">lorem</div>
```

```
var element = document.getElementById('myElement');
var backgroundColor = element.getAttribute('style');
console.log(backgroundColor); // Output: background-color: red;

element.setAttribute('style', 'background-color: blue;');
backgroundColor = element.getAttribute('style');
console.log(backgroundColor); // Output: background-color: blue;
```

To remove an attribute, you can use the `removeAttribute` method:

```
element.removeAttribute('style');
```

This will remove the `style` attribute from the element.

8 Class List

The `classList` property allows you to `add`, `remove`, `toggle`, `replace`, and check if it `contains` classes on an element.

Here is an example of adding, removing, and toggling classes on an element:

- `add`: Adds a class or more to the element.
- `remove`: Removes a class from the element.
- `toggle`: Toggles a class on the element (adds the class if it doesn't exist, removes it if it does).
- `replace`: Replaces a class with another class.
- `contains`: Checks if the element has a specific class.

```
var element = document.getElementById('myElement');
element.classList.add('myClass');
element.classList.remove('myClass');
element.classList.toggle('myClass'); // adds the class again
var hasClass = element.classList.contains('myClass');
console.log(hasClass); // Output: true
element.classList.add('oldClass');
element.classList.replace('oldClass', 'newClass');
console.log(element.classList.contains('oldClass')); // Output: false
console.log(element.classList.contains('newClass')); // Output: true
```

The `classList` property is useful for adding and removing classes dynamically based on user interactions or other events.

You can pass multiple classes to the `add` method by separating them with a comma:

```
element.classList.add('class1', 'class2', 'class3');
```

9 Summary

- The DOM (Document Object Model) is a tree-like structure that represents the document as nodes and objects.
- Each element in the DOM is represented as an object with properties and methods that allow you to interact with it using JavaScript.
- HTML is the initial page content, and the DOM represents the current content in a tree of objects.
- You can select elements in the DOM using methods like:
  - `getElementById` - Returns the element with the specified ID.
  - `getElementsByClassName` - Returns a collection of elements with the specified class name.
  - `getElementsByTagName` - Returns a collection of elements with the specified tag name.
  - `getElementsByName` - Returns a node list of elements with the specified name attribute.
  - `querySelector` - Returns the first element that matches the specified CSS selector.
  - `querySelectorAll` - Returns a node list of elements that match the specified CSS selector.
- `HTMLCollection` is an array-like object that contains elements with the same class name or tag name, while `NodeList` contains nodes, not just elements.
- Event listeners are used to listen for events on elements and execute JavaScript functions when those events occur.
- You can add event listeners using the `addEventListener` method.
- Event listeners can also be added using the `element.event` syntax but it can only have one event listener per event type.
- Some common events include `click`, `dblclick`, `mouseover`, `keydown`, `input`, `change`, `submit`, `focus`, `blur`, etc.
- The event object contains information about the event that occurred, such as the type of event, the target element, and the mouse coordinates.
- You can change the style of an element using the `style` property and the `cssText` property.
- The `setAttribute`, `getAttribute`, and `removeAttribute` methods are used to get, set, and remove attributes of an element.
- The `classList` property allows you to `add`, `remove`, `toggle`, `replace`, and check if it `contains` classes on an element.

Session 21

Mohamed Emary

June 7, 2024

1 Image Slider Examples

In the first part of the session Eng. Shimaa showed us two examples of image sliders. The first one was a simple slider that only changes the main image with the image that is clicked on. The second one was a more complex slider with next and previous buttons, and a close button. This one was more complex in JavaScript.

See the sliders code here.

2 Event Propagation

Event propagation is the process in which the browser determines which event handler to execute first. There are two types of event propagation: bubbling and capturing.

- **Bubbling** is the default propagation method, and it starts from the target element and bubbles up to the root element. For example if you have a parent element and a child element inside the parent element, and you click on the child element, the event will first be handled by the child element's event handler, then by the parent element's event handler.
- **Capturing** is the opposite of bubbling, and it starts from the root element and goes down to the target element. For example if you have a parent element and a child element inside the parent element, and you click on the child element, the event will first be handled by the parent element's event handler, then by the child element's event handler.

To control whether the event propagation is bubbling or capturing, you can use the `addEventListener()` method with the `useCapture` parameter. If `useCapture` is `true`, the event propagation is capturing, and if it is `false` (which is the default), the event propagation is bubbling.

Consider the following example:

HTML:

```
<div
  id="parent"
```

```
  style="width: 200px; height: 200px; background-color: lightblue">
  <div
    id="child"
    style="width: 100px; height: 100px; background-color: lightcoral">
    Click me!
  </div>
</div>
```

JavaScript:

```
document.getElementById('parent').addEventListener('click', function() {
  console.log('Parent clicked!');
}, true);

document.getElementById('child').addEventListener('click', function() {
  console.log('Child clicked!');
}, true);
```

This how it will look:

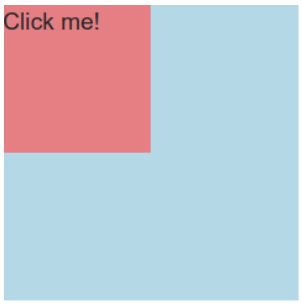

Figure 1: Event Propagation

In this example, if you click on the child element, the output will be:

```
Parent clicked!
Child clicked!
```

This is because the event propagation is capturing, and the parent element's event handler is executed first.

But if you remove the `true` parameter from the `addEventListener()` method or set it to `false` (which is the default so you don't have to write it), the output will be:

```
Child clicked!
Parent clicked!
```

2.1 Stop Propagation

You can stop the event propagation by using the `stopPropagation()` method. This method stops the event from bubbling up or capturing down the DOM tree.

Consider the same example but with this JavaScript code:

```
document.getElementById('parent').addEventListener('click', function() {
  console.log('Parent clicked!');
});
```

```
4
5  document.getElementById('child').addEventListener('click', function(e) {
6    console.log('Child clicked!');
7    e.stopPropagation();
8  });
```

In this example, if you click on the child element, the output will be:

```
Child clicked!
```

This is because the event propagation is stopped by the `stopPropagation()` method, and the parent element's event handler is not executed.

3 Regular Expressions (Regex)

A regular expression is a sequence of characters that define a search pattern. They are used for pattern matching in strings to find or replace text and to validate text inputs like emails, phone numbers, etc so the user can only input the correct format.

That validation should be done in both front-end and the back-end to ensure the data is correct and secure. The front-end and back-end developers should agree on the regular expression pattern to use.

Regex is implemented in the front-end to validate the user input before sending it to the back-end so we can reduce the number of requests to the server to save the user time and the server resources.

When working with Regex you can use this website to test your regular expression pattern to see if it matches the text you want to validate.

There is also a site called I Hate Regex that has a collection of common regex patterns with explanations.

This website provides Regex visualization so you can understand your Regex better.

3.1 Example Regular Expressions With Explanation

- `/a/` matches any string that contains the character `a`.
- `/abc/` matches any string that contains the characters `abc` in order.
- `/(a|b|c)/`, `/[abc]/`, or `/[a-c]/` match any string that contains either `a` OR `b` OR `c`.
- `/[a-z]/` matches any string that contains any lowercase letter.
- `/[A-Z]/` matches any string that contains any uppercase letter.
- `/[A-Z][a-z]/` matches any string that contains an uppercase letter followed by a lowercase letter.
- `/^[A-Z][a-z]/` matches any string that starts with an uppercase letter followed by a lowercase letter.
- `/^[A-Z][a-z]$/` matches any string that starts with an uppercase letter followed by a lowercase letter and nothing else after the lowercase letter.
- `/^[A-Z][a-z]{3}$/` matches any string that starts with an uppercase letter followed by three lowercase letters and nothing else after the lowercase letters.

- `/^[A-Z]{2}[a-z]{3}$/` matches any string that starts with two uppercase letters followed by three lowercase letters and nothing else after the lowercase letters.
- `/^[A-Z]{2,}[a-z]{3,6}$/` matches any string that starts with 2 or more uppercase letters followed by 3 to 6 lowercase letters and nothing else after the lowercase letters.
- `/[0-9]/` matches any string that contains any digit.
- `/^([0-9]|10)$/` matches any string that starts with a digit from 0 to 9 or is equal to 10.
- `/^(Mr|Mrs|Ms)?[A-Z][a-z]+$/` matches any string that starts with an optional title (Mr, Mrs, or Ms) followed by an uppercase letter followed by one or more lowercase letters.
- `/[ah-uz]/` matches any string that contains `a`, or any letter from `h` to `u`, or `z`.

3.2 Real-World Regex Examples

- `/^((\+20)|0)1[0125][0-9]{8}$/` matches any string that starts with either `+20` or `0` followed by `1` then a digit from `0` to `2` or `5` followed by 8 digits (Egyptian phone number format).
- `/^[a-z0-9_-]{3,16}$/` Username validation (only alphanumeric characters, underscore and hyphen, between 3 and 16 characters):

3.3 Some Characters Used in Regex

- `^`: Matches the start of a string. OR negation when used inside `[]` of a character set.
- `$`: Matches the end of a string.
- `*`: Matches zero or more of the preceding element.
- `+`: Matches one or more of the preceding element.
- `?`: Matches zero or one of the preceding element.
- `.`: Matches only one character of any type (Digit, Letter, Special Character).
- `{n}`: Matches exactly `n` of the preceding element.
- `{n,}`: Matches `n` or more of the preceding element.
- `{n,m}`: Matches between `n` and `m` of the preceding element.
- `\d`: Matches any digit character (only one character). Equivalent to `[0-9]`.
- `\D`: Matches any non-digit character (only one character). Equivalent to `[^0-9]`.
- `\w`: Matches any word character (alphanumeric character plus underscore). Equivalent to `[0-9a-zA-Z_]`.
- `\W`: Matches any non-word character. Equivalent to `[^0-9a-zA-Z_]`.
- `\s`: Matches any whitespace character. You can also just use a space character.
- `\S`: Matches any non-whitespace character.

If you want your pattern to contain any of these characters, you should escape them with a backslash `\` for example if you want to match a string that contains the `$` character you should use `/\$/`.

3.4 Some Regex Flags

- `i`: Case-insensitive matching.
- `g`: Global matching (find all matches).

3.5 How to Use Regex

There are two ways to create a regular expression:

1. Using the `RegExp` object constructor:

```
var re = new RegExp('pattern', 'flags');
```

2. Using the literal notation:

```
var re = /pattern/flags;
```

Where `pattern` is the regular expression pattern, and `flags` are optional flags that can be used to change the behavior of the regular expression.

Then we can use the `test()` method to test if the pattern matches a string. The `test()` method returns `true` if the pattern matches the string, and `false` otherwise.

```
var re = /[A-Z][a-z]{3,}/;

// true Starts with uppercase letter followed by 3 or more lowercase
↪  letters
console.log(re.test('Hello'));

// false Starts with a lowercase letter
console.log(re.test('hello'));

// false No lowercase letters after the uppercase letter
console.log(re.test('HELLO'));

// false lowercase letters are less than 3
console.log(re.test('Hi'));
```

Example using Regex with `replace()` method:

```
var re1 = /and/ig;
var re2 = /and/i;
var re3 = /and/g;
var re4 = /and/;

var Str = 'Sand And wind'

// Replace all occurrences of 'and' with 'or'
newStr = Str.replace(re1, 'or');
console.log(newStr); // Sor or wind

// Replace the first occurrence of 'and' or 'And' with 'or'
newStr = Str.replace(re2, 'or');
console.log(newStr); // Sor And wind

```

```
// Replace all occurrences of 'and' with 'or'
newStr = Str.replace(re3, 'or');
console.log(newStr); // Sor And wind

// Replace the first occurrence of 'and' with 'or'
newStr = Str.replace(re4, 'or');
console.log(newStr); // Sor And wind
```

Using `replace()` with a Regex that has the `g` is equivalent to using the `replaceAll()` method.

3.6 Example Regex With User Input

This is an example of how to use Regex with user input to validate an email address while the user is typing:

HTML:

```
<input type="text" id="email" placeholder="Enter your email">
<p id="result"></p>
```

JavaScript:

```
var email = document.getElementById('email');
var result = document.getElementById('result');

email.addEventListener('input', function() {
  var re = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;
  if (re.test(email.value)) {
    result.textContent = 'Valid email';
  } else {
    result.textContent = 'Invalid email';
  }
});
```

4 Operators

4.1 Conditional (Ternary) Operator

Ternary operator `condition ? value1 : value2` can be used to write an if-else statement in one line.

It returns `value1` if `condition` is true, and `value2` if `condition` is false.

```
var a = 5;
var b = 10;
var x = (a > b) ? a : b;
```

In this example, if `a` is greater than `b`, `x` will be equal to `a`. Otherwise, `x` will be equal to `b`.

4.2 Nullish Coalescing Operator

Nullish Coalescing Operator `??` is a new feature in JavaScript that allows you to provide a default value for a variable if the variable is `null` or `undefined`. If the variable is `null` or `undefined`, the result will be the default value. Otherwise, the result will be the variable itself.

```
var x = y ?? z;
```

In this line of code, if `y` is `null` or `undefined`, `x` will be `z`. If `y` is not `null` or `undefined`, `x` will be `y`.

This is useful because it allows you to provide a default value for a variable without having to check if the variable is `null` or `undefined`.

Without the nullish coalescing operator, you would have to do something like this:

```
var x = (y !== null && y !== undefined) ? y : z;
```

4.3 Chaining Operator

Chaining Operator or **Safe Navigation Operator** `?.` is a new feature in JavaScript that allows you to access a property of an object that may be `null` or `undefined` without causing an error. If the property is `null` or `undefined`, the result will be `undefined`. Otherwise, the result will be the property itself.

```
var x = obj?.prop;
```

In this line of code, if `obj` is `null` or `undefined`, `x` will be `undefined` and no error will be thrown. If `obj` is not `null` or `undefined`, `x` will be equal to `obj.prop`.

This is useful because it eliminates the need to check each object in the chain to avoid a `TypeError` being thrown when trying to access a property of `null` or `undefined`.

Without the safe navigation operator, you would have to do something like this:

```
var x = (obj !== null && obj !== undefined) ? obj.prop : undefined;
```

As you can see, the safe navigation operator makes the code cleaner and easier to read.

5 Extras

- `is-valid` and `is-invalid` classes in Bootstrap can be used to style the input fields based on the validation result.
- To access the next sibling of an element you can use the `nextElementSibling` property.

6 Summary

In this session we have covered the following topics:

- We learned about image sliders and how to create them using JavaScript.
- Event propagation is the process in which the browser determines which event handler to execute first. There are two types of event propagation: bubbling and capturing.
- We can stop the event propagation by using the `stopPropagation()` method.
- Regular expressions (Regex) are used for pattern matching in strings to find or replace text and to validate text inputs.
- We learned about some common regex patterns and how to use regex in JavaScript.
- Conditional (Ternary) Operator `condition ? value1 : value2` can be used to write an if-else statement in one line.
- Nullish Coalescing Operator `??` allows you to provide a default value for a variable if the variable is `null` or `undefined`.
- Chaining Operator `?.` allows you to access a property of an object that may be `null` or `undefined` without causing an error.
- `is-valid` and `is-invalid` classes in Bootstrap can be used to style the input fields based on the validation result.
- To access the next sibling of an element you can use the `nextElementSibling` property.

Session 22

Mohamed Emary

June 10, 2024

1 innerHTML and innerText

1.1 innerHTML

- `innerHTML` returns the HTML content of an element.
- When assigning a value with HTML tags to `innerHTML`, the browser will render the HTML tags as HTML elements.
- When printing the value of `innerHTML` of an element that contains HTML tags, the browser will show the HTML tags in the output.

Example:

HTML:

```
<p id="example">My <strong>example</strong> paragraph</p>
```

JavaScript:

```
var example = document.getElementById('example');
console.log(example.innerHTML); // My <strong>example</strong> paragraph

example.innerHTML = 'My <strong>new</strong> paragraph';
console.log(example.innerHTML); // My <strong>new</strong> paragraph
```

1.2 innerText

- `innerText` returns the text content of an element.
- When assigning a value with HTML tags to `innerText`, the browser will render the HTML tags as plain text.
- When printing the value of `innerText` of an element that contains HTML tags, the browser will show the HTML tags in the output.

Example:

HTML:

```
<p id="example">My <strong>example</strong> paragraph</p>
```

JavaScript:

```
var example = document.getElementById('example');
console.log(example.innerText); // My example paragraph

example.innerText = 'My <strong>new</strong> paragraph';
console.log(example.innerText); // My <strong>new</strong> paragraph
```

Here the `<strong>new</strong>` appear in the web page as it is because `innerText` does not render HTML tags.

2 Creating Elements

To create an element, you can use the `document.createElement()` method. This method creates a new element with the specified tag name.

Example:

```
var newElement = document.createElement('div');
```

To set the attributes of the new element, you can use the `setAttribute()` method or the `.` notation.

Example:

```
// Using the setAttribute() method
newElement.setAttribute('id', 'new-element');
newElement.setAttribute('class', 'new-class');

// Or using the . notation
newElement.id = 'new-element';
newElement.className = 'new-class';
```

2.1 Appending & Prepending Elements (Child)

To append an element inside another element in the DOM, you can use the `append()` method, and to prepend an element, you can use the `prepend()` method.

Example:

HTML:

```
<div id="parent" style="background-color: gold">
  <p>First paragraph</p>
  <p>Second paragraph</p>
</div>
```

JavaScript:

```
var parent = document.getElementById('parent');
var newElement = document.createElement('p');
newElement.innerText = 'New paragraph';

```

```
// Append the new element inside the parent element
parent.append(newElement);
```

Now the result will look like this:

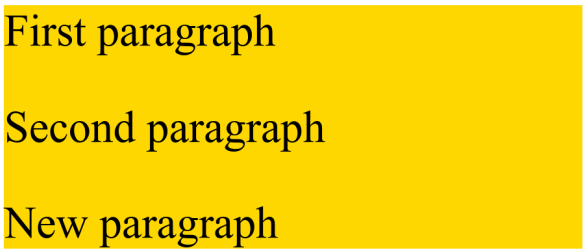

Figure 1: Using `append`

2.2 Add Element Before or After Another (Sibling)

To add an element before or after another element in the DOM, you can use the `before()` and `after()` methods.

Example:

HTML:

```
<div id="parent" style="background-color: gold">
  <p>First paragraph</p>
  <p>Second paragraph</p>
</div>
```

JavaScript:

```
var parent = document.getElementById('parent');
var newElement = document.createElement('p');
newElement.innerText = 'New paragraph';

// Add the new element after the second paragraph
parent.after(newElement);
```

Now the result will look like this:

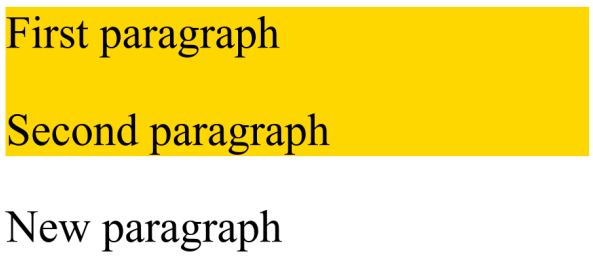

Figure 2: Using `after`

Note

You can only send elements as arguments to the `append()`, `prepend()`, `before()`, and `after()` methods. If you send HTML tag or text, it will be treated as a string and not as an element.

3 Traversing the DOM

Traversing the DOM which is a way to move around the DOM tree and select elements based on their relationship to other elements.

Some useful properties and methods for traversing the DOM are:

1. `parentElement`: returns the parent **element** of an element.
2. `parentNode`: returns the parent **node** of an element.
3. `firstElementChild`: returns the first child **element** of an element.
4. `lastElementChild`: returns the last child **element** of an element.
5. `children`: returns an **HTML collection** of an element's child elements.
6. `childNodes`: returns a **NodeList** of an element's child nodes.
7. `nextElementSibling`: returns the next sibling **element** of an element.
8. `previousElementSibling`: returns the previous sibling **element** of an element.
9. `nextSibling`: returns the next sibling **node** of an element.
10. `previousSibling`: returns the previous sibling **node** of an element.

Example:

HTML:

```
<div id="parent" style="background-color: gold">
  <p id="p1">First paragraph</p>
  <p>Second paragraph</p>
</div>
```

JavaScript:

```
var parent = document.getElementById('parent');
var p1 = document.getElementById('p1');

// Get the parent element of the first paragraph
var parentElement = p1.parentElement;
console.log(parentElement.id); // parent

// Get the parent node of the first paragraph
var parentNode = p1.parentNode;
console.log(parentNode.id); // parent

// Get the first child of the parent element
var firstChild = parent.firstElementChild;
console.log(firstChild.innerText); // First paragraph

// Get the last child of the parent element
var lastChild = parent.lastElementChild;
console.log(lastChild.innerText); // Second paragraph

// Get all the child elements of the parent element
var children = parent.children;
```

```
console.log(children.length); // 2
console.log(children[1]); // <p>Second paragraph</p>

// Get all the child nodes of the parent element
var childNodes = parent.childNodes;
console.log(childNodes.length); // 3
console.log(childNodes[1]); // #text

// Get the next sibling element of the first paragraph
var nextSibling = p1.nextElementSibling;
console.log(nextSibling.innerText); // Second paragraph

// Get the previous sibling element of the second paragraph
var previousSibling = lastChild.previousElementSibling;
console.log(previousSibling.innerText); // First paragraph

// Get the next sibling node of the first paragraph
var nextNode = p1.nextSibling;
console.log(nextNode); // #text

// Get the previous sibling node of the second paragraph
var previousNode = lastChild.previousSibling;
console.log(previousNode); // #text
```

4 Important Differences

4.1 `previousSibling`, `nextSibling` VS `previousElementSibling`, `nextElementSibling`

- `previousSibling` and `nextSibling` return nodes and these nodes include both element nodes and non-element nodes (like text and comment nodes).
- `previousElementSibling` and `nextElementSibling` return only element nodes and ignore text and comment nodes.

```
<div>
  <p>Paragraph 1</p>
  Text
  <p>Paragraph 2</p>
</div>
```

If the current node is the first `<p>` element, `nextSibling` would return the text node `Text`, while `nextElementSibling` would return the second `<p>` element.

4.2 NodeList VS HTML Collection

NodeList	HTML Collection
Static (Does not update when the DOM changes)	Live (Updates when the DOM changes)
Returns a list of nodes	Returns a list of elements

NodeList	HTML Collection
Nodes can be of any type like element, text, comment, etc.	Elements only
Returned by methods like `querySelectorAll`, `childNodes`	Returned by methods like `getElementsByTagName`, `children`

By saying that the `NodeList` is static, it means that if you add an element to the DOM after getting the `NodeList`, the `NodeList` will not include the new element. On the other hand, the `HTML Collection` is live, which means that it will include the new element even after getting the `HTML Collection`.

Example:

HTML:

```
<div>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</div>
```

JavaScript:

```
var div = document.querySelector('div');
var paragraphsCollection = div.getElementsByTagName('p');
var paragraphsNodeList = div.querySelectorAll('p');

console.log(paragraphsCollection.length); // 2
console.log(paragraphsNodeList.length);   // 2

var newParagraph = document.createElement('p');
newParagraph.innerText = 'New paragraph';

div.append(newParagraph);

console.log(paragraphsCollection.length); // 3
console.log(paragraphsNodeList.length);   // 2
```

In this example, the `paragraphsCollection` will have a length of 3, while the `paragraphsNodeList` will have a length of 2 because the `NodeList` is static and does not change when the DOM changes while the `HTML Collection` is live and changes when the DOM changes.

5 Browser Object Model (BOM)

Browser Object Model or BOM is a set of objects provided by the browser to interact with the browser itself.

5.1 DOM VS BOM

The DOM can be accessed via the BOM through the `window.document` property. So, you can say that the DOM is part of the BOM in a browser environment.

`window` is a super global object in the browser environment.

DOM is concerned with the content of the web document, while the BOM is concerned with the browser environment.

5.2 BOM Methods & Properties

Some of the BOM methods include:

5.2.1 `setInterval`

`setInterval()`: Calls a function or evaluates an expression each time a specified number of milliseconds elapses.

For example, to display the value of a counter every second:

```
function incrementCounter() {
  console.log(counter);
  counter++;
}

var counter = 0;
var interval = setInterval(incrementCounter, 1000);
```

5.2.2 `clearInterval`

`clearInterval()`: Stops the intervals set by `setInterval()`.

For example, to stop the counter we made earlier when the user clicks a button:

```
var button = document.getElementById('stop');

button.addEventListener('click', function() {
  console.log("Counter stopped");
  clearInterval(interval);
});
```

5.2.3 `setTimeout`

`setTimeout()`: Calls a function or evaluates an expression **once** after a specified number of milliseconds.

For example, to display a message after 3 seconds:

```
function showMessage() {
  console.log("Hello, world!");
}

setTimeout(showMessage, 3000);
```

5.2.4 `alert`

`alert()`: Displays an alert box with a message and an OK button.

For example:

```
alert("Hello, world!");
```

5.2.5 `open`

`open()`: Opens a new browser window or a new tab.

For example, This will open a new tab with Google's homepage.

```
var googleBtn = document.getElementById('open');
googleBtn.addEventListener('click', function() {
  open('https://www.google.com', '_blank');
});
```

`_blank` is the name of the target window. It specifies that the URL should be opened in a new tab and it's the default value. To open the URL in the same tab, you can use `_self`.

`open` also has other parameters like `width`, `height`, `top`, `left`, etc.

```
open('https://www.google.com', '_blank',
 ↪  'width=500,height=500,top=100,left=100');
```

Notice that `width`, `height`, `top`, and `left` are passed as a string with a comma separating them.

`open` is also one of `window`'s methods, so you can use it in the form `window.open()`.

5.2.6 `close`

`close()`: Closes the current window.

For example, to close the current window when the user clicks a button:

```
var closeBtn = document.getElementById('close');
closeBtn.addEventListener('click', function() {
  close();
});
```

5.2.7 `innerWidth` and `innerHeight`

`innerWidth` and `innerHeight` properties return the width and height of the content area of the browser window.

If you resize the browser window, the values of `innerWidth` and `innerHeight` will change accordingly.

```
console.log(window.innerWidth);
console.log(window.innerHeight);
```

5.2.8 `screen` Object

The `screen` object provides information about the user's screen.

Some of the properties of the `screen` object include:

- `screen.width`: Returns the width of the screen.
- `screen.height`: Returns the height of the screen.

Those properties don't change when you resize the browser window because they are related to the user's screen (the hardware) and not the browser window.

```
console.log(screen.width);
console.log(screen.height);
```

`screen` object also has other properties like `availWidth`, `availHeight`.

These are the areas of the screen that you can use to display content, it doesn't include the taskbar or any other system-related areas.

```
console.log(screen.availWidth);
console.log(screen.availHeight);
```

5.3 `location` Object

The `location` object contains information about the current URL.

Some of the properties of the `location` object include:

- `location.href`: Returns the entire URL.
- `location.hostname`: Returns the domain name of the web host.
- `location.pathname`: Returns the path and filename of the current page.
- `location.history`: Returns the history of the current page.
  - `location.history.back()`: Goes back to the previous page.
  - `location.history.forward()`: Goes forward to the next page.

6 API

API (Application Programming Interface) is universal way for different software applications to communicate with each other.

APIs can be used to **recieve** data from a server, **send** data to a server, **modify** data on a server, and **delete** data from a server.

The API comes in the form of a URL that you can send a request to and get a response from.

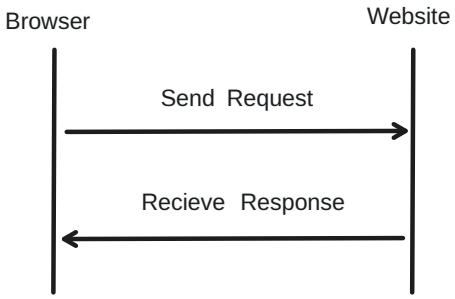

Figure 3: Request & Response

When dealing with APIs the front-end developer takes the API from the back-end developer with the documentation of how to use it. Example API documentation.

Then the front-end developer uses the API to get the data needed to display on the web page.

A good way to test an APIs is to use a tool like Postman, which is a collaboration platform for API development.

6.1 JSON

When you get a response from an API, it's usually in the form of JSON (JavaScript Object Notation).

JSON objects are easy to read and write. They are human-readable and can be parsed by JavaScript. JSON objects are written in key/value pairs and can be either an object or an array of objects.

Example of a JSON object:

```
{
  "name": "Mohamed",
  "age": 30,
  "city": "Cairo"
}
```

6.2 Free APIs

There are many free APIs available that you can use to practice working with APIs.

Example Free APIs:

- JSONPlaceholder
- Forkify Meals API
- Random User Generator API
- Weather API
- News API
- MovieDB API
- Fake Store API

You can find many more on this `public-apis` GitHub repo.

6.3 Terms Related to APIs

There is some terms related to APIs:

Lets use this API to explain the terms: `https://api.github.com/users/Microsoft`

- **Base URL**: The main URL of the API. For example, `https://api.github.com/`
- **Endpoint**: The part of the URL after the base URL that specifies a particular resource or collection of resources. For example, `/users/Microsoft` is the endpoint in the URL `https://api.github.com/users/Microsoft`.
- **Request**: The action you want the API to perform. In this case, a `GET` request to `https://api.github.com/users/Microsoft` to retrieve the data of the user `Microsoft`.

- **Response**: The data you get back from the API. This is typically in the form of a JSON object or array.
- **Status Code**: A number returned by the server that indicates the result of the request. For example, 200 means the request was successful, while 404 means the requested resource could not be found.
- **Method**: The type of request you are making. Common methods include `GET`, `POST`, `PUT`, `DELETE`, and `PATCH`.
  - `GET`: To get data from the server.
  - `POST`: To send data to the server.
  - `PUT`: To update data on the server.
  - `DELETE`: To delete data on the server.
  - `PATCH`: To partially update data on the server.
  - `PUT`: To update data on the server.

6.4 How to Use an API

To use an API, you need to know about AJAX (Asynchronous JavaScript and XML) first.

AJAX allows you to send and receive data from a server asynchronously without reloading the page.

You can use the `XMLHttpRequest` object to interact with the server and get data from it. We don't use the object directly, but we create a `new` instance of it and use its methods.

Example of creating an instance of `XMLHttpRequest`:

```
var xhr = new XMLHttpRequest();
```

Then you can use the `open()` method to establish a connection with the server by specifying the request **method** and the **API URL**.

```
xhr.open('METHOD', 'API_URL');
```

For example to establish a connection with the forkify API using the `GET` method:

```
xhr.open('GET', 'https://forkify-api.herokuapp.com/api/search?q=pizza');
```

After opening the connection, you can use the `send()` method to send the request to the server.

```
xhr.send();
```

To handle the response from the server, you can use the `onload` event handler.

```
xhr.addEventListener('load', function() {
  console.log(xhr.response);
});
```

The `response` property of the `XMLHttpRequest` object contains the response data from the server as a **string**, so you need to parse it to a JSON object.

```
xhr.addEventListener('load', function() {
  var data = JSON.parse(xhr.response);
  console.log(data);
});
```

Now you can access the data returned by the API.

```
xhr.addEventListener('load', function() {
  var data = JSON.parse(xhr.response);
  console.log(data.recipes);
});
```

You can also use `readystatechange` event handler to check the status of the request before accessing the data.

```
xhr.addEventListener('readystatechange', function() {
  if (xhr.readyState === 4 && xhr.status === 200) {
    var data = JSON.parse(xhr.response);
    console.log(data.recipes);
  }
});
```

`readyState` values are:

- `0`: request not initialized
- `1`: server connection established
- `2`: request sent
- `3`: processing request
- `4`: request finished and response is ready

`status` values are:

- `200`: OK (request successful)
- `403`: Forbidden (access denied)
- `404`: Not Found (resource not found)
- `500`: Internal Server Error

Here is the complete code to get data from the forkify API:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'https://forkify-api.herokuapp.com/api/search?q=pizza');
xhr.send();

xhr.addEventListener('load', function() {
  if (xhr.readyState === 4 && xhr.status === 200) {
    var data = JSON.parse(xhr.response);
    console.log(data.recipes);
  }
});
```

We also have `error` event handler to handle errors when the request fails.

```
xhr.addEventListener('error', function() {
  console.log('An error occurred');
});
```

6.5 Displaying Data from an API

This is an example of how to display data from an API on a web page.

HTML:

```
<div class="container">
  <div class="row" id="rowBody"></div>
```

```
</div>
```

JavaScript:

```
var xhr = new XMLHttpRequest();
var allRecipies = [];
xhr.open("get", "https://forkify-api.herokuapp.com/api/search?q=pizza");
xhr.send();
xhr.addEventListener("readystatechange", function () {
  if (xhr.readyState == 4 && xhr.status == 200) {
    allRecipies = JSON.parse(xhr.response).recipes;
    display();
  }
});

function display() {
  var content = ``;
  for (var i = 0; i < allRecipies.length; i++) {
    content += `
      <div class="col-md-4">
        <img
          src="${allRecipies[i].image_url}"
          class="w-100 rounded-2"
          height="200px"
        />
        <h3>${allRecipies[i].title}</h3>
        <p>${allRecipies[i].publisher}</p>
      </div>`;
  }
  document.getElementById("rowBody").innerHTML = content;
}
```

7 Summary

- `innerHTML` returns the HTML content of an element, while `innerText` returns the text content of an element.
- When assigning a value with HTML tags to `innerHTML`, the browser will render the HTML tags as HTML elements.
- When assigning a value with HTML tags to `innerText`, the browser will render the HTML tags as plain text.
- To create an element, you can use the `document.createElement()` method.
- To append an element inside another element in the DOM, you can use the `append()` method, and to prepend an element, you can use the `prepend()` method.
- To add an element before or after another element in the DOM, you can use the `before()` and `after()` methods.
- Traversing the DOM is a way to move around the DOM tree and select elements based on their relationship to other elements.
- `previousSibling` and `nextSibling` return nodes and include both element nodes and non-element nodes, while `previousElementSibling` and `nextElementSibling` return only element nodes.
- The Browser Object Model (BOM) is a set of objects provided by the browser to interact with the browser itself.
- Some of the BOM methods include `setInterval`, `clearInterval`, `setTimeout`, `alert`, `open`, and `close`.
- The `screen` object provides information about the user's screen, and the `location` object contains information about the current URL.
- APIs (Application Programming Interfaces) are used to communicate between different software applications.
- JSON (JavaScript Object Notation) is a common format for data exchange in APIs.
- To use an API, you need to know about AJAX (Asynchronous JavaScript and XML).
- You can use the `XMLHttpRequest` object to interact with the server and get data from it.
- To display data from an API on a web page, you can create an instance of `XMLHttpRequest`, send a request to the API, and handle the response to access the data.
- Common HTTP methods include `GET`, `POST`, `PUT`, `DELETE`, and `PATCH`.
- The `readyState` property of the `XMLHttpRequest` object indicates the state of the request, and the `status` property indicates the status of the response.
  - `readyState` values: `0`, `1`, `2`, `3`, `4`
  - `status` values: `200`, `403`, `404`, `500`

Session 23

Mohamed Emary

June 17, 2024

1 Synchrounous & Asynchronous

A synchronous operation is one that blocks the execution of other code until it is finished, this is also known as **blocking** code.

An asynchronous operation is one that does not block the execution of other code, instead, it allows other code to continue executing while it waits for the operation to complete, this is known as **non-blocking** code.

Example of synchronous code:

```
1 console.log('1'); // synchronous
2 console.log('2'); // synchronous
3 console.log('3'); // synchronous
```

```
1
2
3
```

Example of asynchronous code:

```
1 console.log('1'); // synchronous
2 setTimeout(function () { //
  ↪  asynchronous
3   console.log("2");
4 }, 1000);
5 console.log('3'); // synchronous
```

```
1
3
2
```

In the case of this code example, the `setTimeout` function is an example of an asynchronous operation. It tells the browser to wait for a certain amount of time before executing the callback function. While the browser is waiting, it can continue executing other code, that is why the output of the code is `1`, `3`, and `2`.

To understand the difference and why the output is different, we need to understand how JavaScript works.

1.1 How JavaScript Works

JavaScript is a **non-blocking single-threaded** language. This means that it can only execute one piece of code at a time. Some other languages like Java are **multi-threaded**, which means that they can execute multiple pieces of code at the same time.

JavaScript runtime environment has the following components:

- **Execution stack (Call Stack)**: where code is executed. It has the synchronous methods and global variables
- **Callback queue (Task Queue)**: where asynchronous tasks are placed
- **Web API**: a set of functions provided by the browser to handle long-running tasks that would take a long time to execute
- **Event loop**: checks if there are any tasks in the callback queue

When JavaScript code is executed, it is added to the **execution stack**. The execution stack is a data structure that keeps track of the execution of the code. When a function is called, it is added to the call stack. When the function finishes executing, it is removed from the call stack.

Both **execution stack** and **web API** work at the same time to execute the code.

Synchrounous code gets executed directly in the call stack.

When JavaScript encounters an asynchronous operation, like a `setTimeout` function, it does not execute the code immediately (which means adding it to the call stack), instead, it hands off the operation to the **web API** provided by the browser. The web API handles the operation in the background and when it is finished, it adds the result to the **task queue** then the **event loop** checks if there are any tasks in the task queue and if there are, it adds them to the call stack, but **only when the call stack is empty**.

The **task queue** tasks are divided into two categories:

- **Microtasks**: tasks that have high priority and have the functions that return promises, or uses `await`, `async`.
- **Macrotasks**: tasks like `setTimeout`, `setInterval`.

Microtasks have higher priority than macrotasks, so they are executed first.

This diagram demonstrates what each component of the JavaScript runtime environment takes care of:

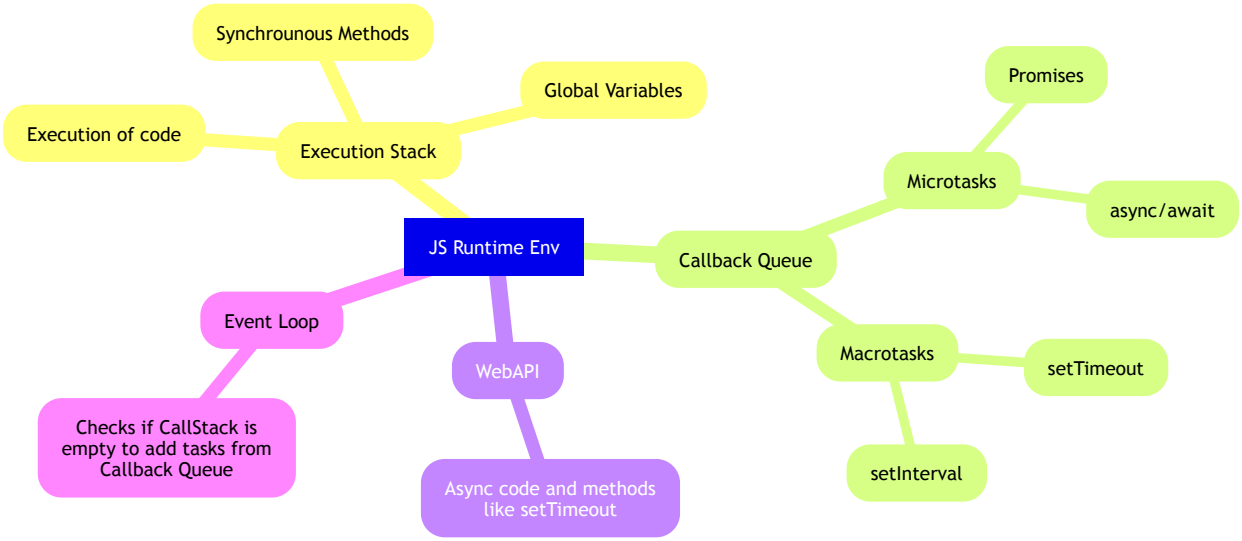

Figure 1: JavaScript Runtime Environment

This image demonstrates the steps of the process:

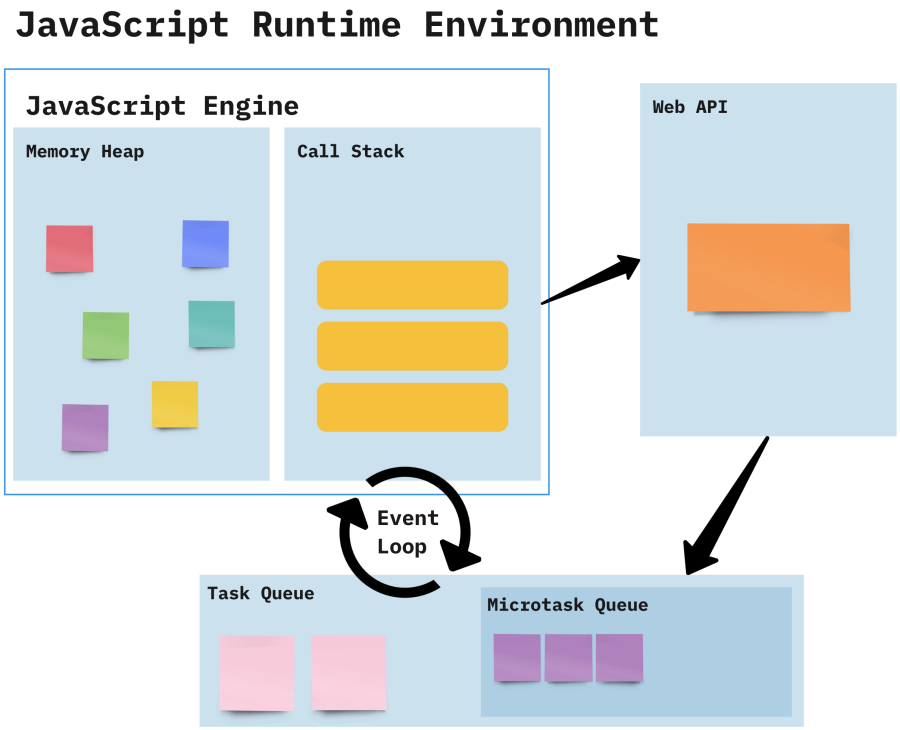

Figure 2: How JavaScript Works

2 Control Code Execution Flow

Sometimes we need to control the flow of the code execution, for example, when we need to execute a piece of code after another because it depends on the result of the first one.

That is when we use **callbacks**, **promises**, and **async/await**.

2.1 Callbacks

A callback is a function that is passed as an argument to another function. The function that receives the callback function will execute first then it will call the callback function.

```
function first(callback) {
  console.log('First function');
  callback();
}

function second(callback) {
  console.log('Second function');
  callback();
}

function third() {
  console.log('Third function');
}

first(function () {
```

```
  second(function () {
    third();
  });
});
```

This is how the code will be executed:

1. The `first` function is called and it logs `First function`.
2. Then it calls the callback function which is the `second` function.
3. The `second` function is called and it logs `Second function`.
4. Then it calls the callback function which is the `third` function.
5. The `third` function is called and it logs `Third function`.

In this code, the callback functions are wrapped in anonymous functions because if we passed the callbacks with arguments directly (like `first(second(third))`), the functions will be executed immediately.

Applying callbacks to the first example:

```
function first(callback) {
  console.log('1');
  callback();
}

function second(callback) {
  setTimeout(function () {
    console.log('2');
    callback();
  }, 1000);
}

function third() {
  console.log('3');
}

first(function () {
  second(function () {
    third();
  });
});
```

Now the functions will be executed in order and the output will be `1`, `2`, `3` just as we want.

We can also check if the callback function exists before calling it so we don't call it if it doesn't exist.

```
function first(callback) {
  console.log('1');
  if (callback) {
    callback();
  }
}
```

2.1.1 Callback Hell

As you can see from the code examples when we have a lot of nested callbacks, the code becomes hard to read and maintain, this is known as **Callback Hell**.

For this reason, callback functions are not used a lot instead we use **promises** and **async/await**.

2.2 Promise

A promise is an object that represents the eventual completion or failure of an asynchronous operation and its resulting value.

A promise has three states:

- **Pending**: the initial state, neither fulfilled nor rejected.
- **Fulfilled (resolved)**: the operation completed successfully. Used with `.then()` method.
- **Rejected**: the operation failed. Used with `.catch()` method.

A promise is created using the `Promise` constructor:

```
var promise = new Promise(function (resolve, reject) {
  // code here
});
```

The `Promise` constructor takes a function as an argument that has two parameters `resolve` and `reject`. These parameters are functions that are used to resolve or reject the promise.

```
var promise = new Promise(function (resolve, reject) {
  setTimeout(function () {
    resolve('Success');
  }, 1000);
});

promise.then(function (value) {
  console.log(value);
});
```

In this example, the promise will be resolved after 1 second and the `then` method will be called with the value `Success`.

If the promise is rejected, the `catch` method will be called:

```
var promise = new Promise(function (resolve, reject) {
  setTimeout(function () {
    reject('Error');
  }, 1000);
});

promise.then(function (value) {
  console.log(value);
}).catch(function (error) {
  console.error(error);
});
```

The value of `error` and `value` parameters are the values passed to the `resolve` and `reject` functions, so in our case `value = 'Success'` and `error = 'Error'`.

In this example, the promise will be rejected after 1 second and the `catch` method will be called with the error `Error`.

Note:

To be able to use `.then()` when calling a function the function must return a promise, and the promise must be resolved.

2.2.1 Promise Chaining

Promises can be chained together to execute code in a specific order.

```
function one(param1) {
  return new Promise(function(resolve, reject) {
    // Do something with param1
    // Resolve or reject based on the result
    resolve('Function One processed ' + param1);
  });
}

function two(param2) {
  return new Promise(function(resolve, reject) {
    // Do something with param2
    // Resolve or reject based on the result
    resolve('Function Two processed ' + param2);
  });
}

function three(param3) {
  return new Promise(function(resolve, reject) {
    // Do something with param3
    // Resolve or reject based on the result
    reject('Function Three encountered an error with ' + param3);
  });
}

one('input1')
  .then(function(result1) {
    console.log(result1);
    return two('input2');
  })
  .then(function(result2) {
    console.log(result2);
    return three('input3');
  })
  .then(function(result3) {
    console.log(result3);
  })
  .catch(function(error) {
```

```
38      console.error('Error:', error);
39    });
```

```
Function One processed input1
Function Two processed input2
Error: Function Three encountered an error with input3
```

This is how the code will be executed:

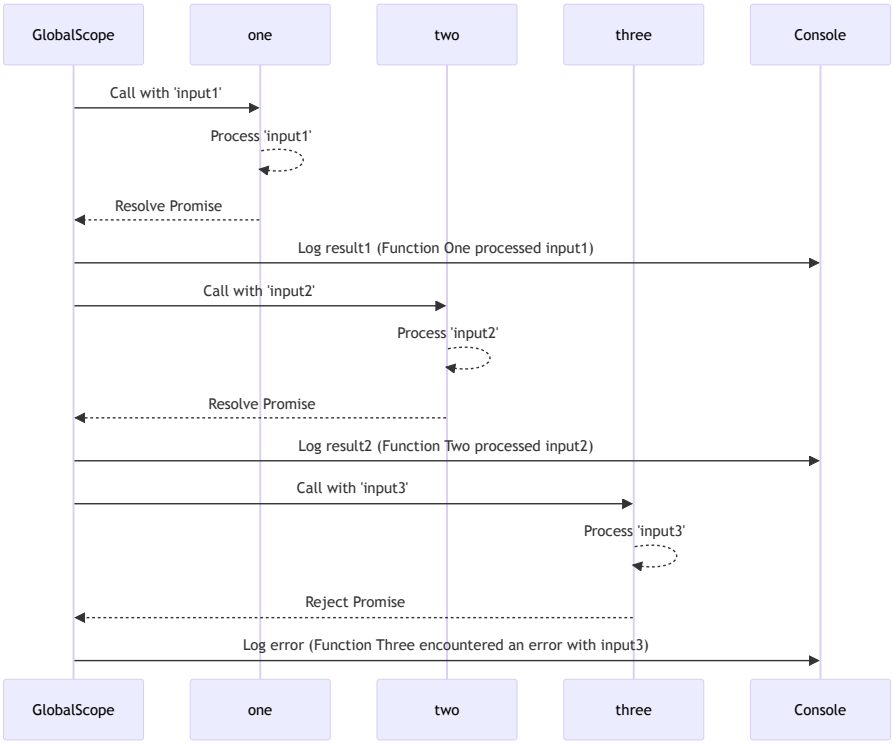

Figure 3: Promise Chaining

1. The `one` function is called with the parameter `'input1'`. This function returns a new Promise. Inside this Promise, some processing is done with `'input1'`, and then the Promise is resolved with a message indicating that `'input1'` has been processed.
2. The `then` method is called on the Promise returned by the `one` function. This `then` method takes a function as an argument, which will be executed when the Promise is resolved. The result of the `one` function (the resolve message) is logged to the console.
3. After the first `then` method has finished executing, it returns a new Promise by calling the `two` function with the parameter `'input2'`. Similar to the `one` function, the `two` function does some processing with `'input2'` and then resolves the Promise with a message.
4. The next `then` method is called on the Promise returned by the `two` function. Again, this `then` method takes a function as an argument, which logs the result of the `two` function to the console.
5. After the second `then` method has finished executing, it returns a new Promise by calling the `three` function with the parameter `'input3'`. However, this time, the Promise is rejected with an error message instead of being resolved.
6. Because the Promise from the `three` function was rejected, the next `then` method is skipped, and the `catch` method is called instead. The `catch` method also takes a function as an argument, which logs the error message to the console.

Note:

If the promise in the chain is rejected, the next `then` method is skipped, and the `catch` method is called instead.

We can use `.catch()` method with the `error` event listener when calling an API so if the API call fails, the promise will be rejected and the `catch` method will be called.

We also have the `finally` method that is called at the end of the promise chain and is called regardless of whether the promise is resolved or rejected.

```
promise.then(function (value) {
  console.log(value);
}).catch(function (error) {
  console.error(error);
}).finally(function () {
  console.log('Finally');
});
```

The Promise way of controlling the flow of the code execution can be hard to read and maintain when we have a lot of promises, that is why we have **async/await**.

2.3 Async/Await

Async/await is a new way to write asynchronous code in JavaScript. It is built on top of promises and provides a more readable and maintainable way to write asynchronous code. But before we dive into async/await, we need to know the `fetch` API.

2.3.1 Fetch API

The `fetch` API is a modern replacement for the `XMLHttpRequest` object. It is used to make network requests to a server and is built into the browser.

The `fetch` function takes a URL as an argument and **returns a promise** that resolves to the `Response` object representing the response to the request.

The `fetch` function have a `GET` method by default, but we can specify the method using the `method` option.

Syntax:

```
fetch(API_URL, options)
  .then(function (response) {
    // Do something with the response
  })
  .catch(function (error) {
    // Handle any errors
  });
```

The `options` object is a JSON object that contains the configuration for the request. Some of the options are:

- `method`: the HTTP method to use for the request (e.g., `GET`, `POST`, `PUT`, `DELETE`).
- `headers`: an object containing the headers to include in the request.

- `body`: the body of the request (e.g., JSON data).

Example:

```
fetch("https://jsonplaceholder.typicode.com/posts/1", {
  method: "GET", // Default value you don't have to specify it
})
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  })
  .catch(function (error) {
    console.error(error);
  });
```

This is how the code will be executed:

1. `fetch("https://jsonplaceholder.typicode.com/posts/1", { method: "GET" })`: This line sends a GET request to the specified URL. The `fetch` function returns a Promise that resolves to the Response object representing the response to the request.
2. `.then(function (response) { return response.json(); })`: This is a Promise chain. When the Promise from the `fetch` function resolves, it passes the Response object to this function. The `response.json()` method reads the response body and returns another Promise that resolves with the result of parsing the body text as JSON.
3. `.then(function (data) { console.log(data); })`: This is another link in the Promise chain. When the Promise from the `response.json()` method resolves, it passes the parsed JSON data to this function, which logs the data to the console.
4. `.catch(function (error) { console.error(error); })`: This is the error handling part of the Promise chain. If any of the Promises in the chain reject (i.e., an error occurs), this function will be called with the error as its argument. It logs the error to the console.

But that is still hard to read and unclear, that is why we have **async/await**.

2.3.2 Using Async/Await

The `async` and `await` keywords were introduced in ES8 (ECMAScript 2017) to make asynchronous code easier to read and write.

The `async` keyword is used to define an asynchronous function, which returns a promise. The `await` keyword is used to pause the execution of an asynchronous function until a promise is resolved.

```
async function fetchData() {
  var response = await
  ↪  fetch("https://jsonplaceholder.typicode.com/posts/1");
  var data = await response.json();
  console.log(data);
}

fetchData();
```

This is how the code will be executed:

1. `fetchData()`: Calls the function. `fetchData` returns a promise because it is an asynchronous function.
2. `await fetch(...)`: Sends a request and `await` pauses the function until the promise returned by `fetch` is resolved.
3. `await response.json()`: reads the response body and waits for the parsing of the body text as JSON.

Note:

If you remove the `await` keyword from any of the lines then log the `response` or `data` variables, you will get `Promise<pending>` because the `fetch` and `response.json()` methods return promises.

If you have many asynchronous functions and you want to execute them in order, you can put them inside an `async` function and use `await` keyword to wait for each function to finish before executing the next one.

This is a more readable and maintainable way to write asynchronous code compared to promises.

3 try, catch, and finally

The `try`, `catch`, and `finally` statements are used to handle errors in JavaScript code.

The `try` statement allows you to define a block of code to be **tested for errors** while it is being executed.

The `catch` statement allows you to define a block of code to be **executed if an error occurs** in the `try` block.

The `finally` statement allows you to define a block of code to be **executed** after the `try` and `catch` blocks, **regardless of whether an error occurred or not**.

```
try {
  // Code to be executed
} catch (error) {
  // Code to handle the error
} finally {
  // Code to be executed after the try and catch blocks
}
```

The `try` and `catch` statements are often used together with asynchronous code to handle errors that occur during the execution of the code.

```
try {
  var response = await
   ↪  fetch("https://jsonplaceholder.typicode.com/posts/1");
  var data = await response.json();
  console.log(data);
} catch (error) {
```

```
  console.error(error);
}
```

In this example, the `try` block contains the asynchronous code that fetches data from a URL. If an error occurs during the execution of the code, the error is caught by the `catch` block and logged to the console.

Another example:

```
try {
  console.log(x); // we didn't define x
} catch (error) {
  console.error(error); // ReferenceError: x is not defined
  console.log(error.name); // ReferenceError
  console.log(error.message); // x is not defined
} finally {
  console.log("Finally block");
}
```

When using `throw` to throw an error, you don't always use `Error()`, you can also use a specific error type like `ReferenceError()`, `TypeError()`, `RangeError()`, etc.

```
try {
  throw new ReferenceError("This is a reference error");
} catch (error) {
  console.error(error); // ReferenceError: This is a reference error
  console.error(error.name); // ReferenceError
  console.error(error.message); // This is a reference error
}
```

4 Summary

- **Synchronous** code blocks the execution of other code until it is finished.
- **Asynchronous** code allows other code to continue executing while it waits for the operation to complete.
- JavaScript is a **non-blocking single-threaded** language.
- JavaScript runtime environment has the **execution stack**, **callback queue**, **web API**, and **event loop**.
- Code execution flow can be controlled using **callbacks**, **promises**, and **async/await**.
- **Callbacks** are functions that are passed as arguments to other functions.
- **Promises** are objects that represent the eventual completion or failure of an asynchronous operation.
- **Async/await** is a new way to write asynchronous code in JavaScript.
- The `fetch` API is used to make network requests to a server.
- The `try`, `catch`, and `finally` statements are used to handle errors in JavaScript code.

Session 24

Mohamed Emary

June 27, 2024

1 "use strict"

When JavaScript was first introduced, it was a very forgiving language. It would try to make sense of whatever code you gave it, even if it was poorly written. This could lead to bugs that were hard to track down.

Examples of common coding problems that JavaScript would allow when it was first introduced include:

- Using a variable and assigning a value to it without declaring it first. `x = 5;`
- Duplicating a parameter name. `function sum(x, x) { /* function body */ }`
- Using a reserved word as a variable or function name. `var let = 5;`

In 2009, ECMAScript 5 (ES5) introduced a new feature called "strict mode" that would help developers catch these bugs earlier. Strict mode is a way to use to a restricted variant of JavaScript that would catch common coding problems and throw exceptions.

Using strict mode in cases like the ones above would throw an error, which would help you catch the bugs earlier in the development process.

To enable strict mode, you can add the following line to the top of your JS code:

```
'use strict'; // be sure to include the "quotes"
```

You can also enable strict mode for just a single function by adding the same line at the top of the function.

```
function doSomething() {
  'use strict';
  // This code is in strict mode
}
```

Strict mode is supported in all modern browsers, and it's a good practice to use it in your code.

```
'use strict';

function doSomething() {
```

```
  // This code is in strict mode
}

function doSomethingElse() {
  // This code is also in strict mode
}
```

Classes and modules which have strict mode enabled by default.

2 let and const

ES6 introduced two new ways to declare variables: `let` and `const`.

2.1 let

`let` is similar to `var`, but it has a few key differences:

- Variables declared with `let` are **block-scoped**, while variables declared with `var` are **function-scoped**.
  - This will save memory because the variable will only be available within the block where it was declared, and after the block ends, the variable will be removed from memory, this will free up memory from unnecessary variables.
- Variables declared with `let` are not **hoisted to the top of the block**, while variables declared with `var` are **hoisted**.
- Variables declared with `let` **cannot be redeclared** in the same scope, while variables declared with `var` can be.

2.2 const

`const` is similar to `let`, but it has one key difference:

- Variables declared with `const` **cannot be reassigned** to a new value.

Example of using `const` is to store a value that you know will not change, like the value of $\pi = 3.14159$.

It can also be used in DOM manipulation to store references to elements that you know will not change

```
const p = document.getElementById('myParagraph');

// Even if you change any property of the element, no problem
// the reference to the element is still the same
p.textContent = "Hello, World"
```

Here are some examples of using `let` and `const`:

```
// Block scope
{
  let x = 5;
  var y = 10;
  console.log(x); // 5
  console.log(y); // 10
```

```
}
// console.log(x); // ReferenceError: x is not defined, because `let` is block-scoped
console.log(y);    // 10

// reassignment
let x = 15;
const z = 20;
x = 25;
// z = 30; // TypeError: Assignment to constant variable, because `const` does not allow reassignment

// redeclaration
var y = 30;
var y = 35;

let a = 40;
// let a = 45; // SyntaxError: Identifier 'a' has already been declared
```

Temporal Dead Zone (TDZ) with `let`

Variables declared with `let` get hoisted to the top of the block, but they are not initialized until the line where they are declared is reached. This is called the Temporal Dead Zone (TDZ).

Example:

```
console.log(x); // ReferenceError: Cannot access 'x' before initialization
let x = 5;
```

2.2.1 `for of` & `const`

When using `const` with `for of`, it will not throw an error because in a `for...of` loop, each iteration creates a new block scope, allowing `const` to be safely used without reassignment errors.

This means it doesn't reassign the variable, but it creates a new variable in a new scope in each iteration.

```
const arr = [1, 2, 3, 4, 5];

for (const item of arr) {
  console.log(item);
}
```

Note: *Now you should always use either* ***`let`*** *or* ***`const`****, never use* ***`var`*** *again.*

3 Default Parameter Value

ES6 introduced a new feature called default parameter values. This allows you to specify a default value for a parameter in a function if no argument is provided.

```
function greet(name = 'World') {
  console.log(`Hello, ${name}!`);
}

greet(); // Hello, World!
greet('Mohamed'); // Hello, Mohamed!
```

This feature is useful when you want to provide a default value for a parameter if no argument is provided.

The old way to do this was to use the `||` operator or an `if` statement:

```
// Using the || operator
function greet(name) {
  name = name || 'World';
  console.log(`Hello, ${name}!`);
}

greet(); // Hello, World!
greet('Mohamed'); // Hello, Mohamed!

// Using an if statement
function greet(name) {
  if (name === undefined) {
    name = 'World';
  }
  console.log(`Hello, ${name}!`);
}

greet(); // Hello, World!
greet('Mohamed'); // Hello, Mohamed!
```

4 Template Literal `String`

ES6 introduced a new way to create strings called template literals. Template literals are enclosed by backticks (`) instead of single quotes (') or double quotes (").

Template literals can contain placeholders, which are indicated by the dollar sign and curly braces (`${expression}`). The expression inside the curly braces is evaluated and the result is inserted into the string.

```
let name = 'Mohamed';
let age = 30;

// Old way
let message = 'Hello, ' + name + '! You are ' + age + ' years old.';
console.log(message); // Hello, Mohamed! You are 30 years old.

// New way
let message = `Hello, ${name}! You are ${age} years old.`;
console.log(message); // Hello, Mohamed! You are 30 years old.
```

Template literals can span multiple lines without the need for escape characters:

```
let message = `This is a
multi-line
string.`;
console.log(message); // This is a
                      // multi-line
                      // string.
```

5 Destruction Assignment

Destructuring assignment is a feature introduced in ES6 that allows you to extract values from arrays or objects and assign them to variables in a more concise way.

5.1 Array Destructuring

Array destructuring allows you to extract values from an array and assign them to variables in a single statement.

```
let numbers = [1, 2, 3, 4, 5];

[a, b, c, d, e] = numbers;
console.log(a, b, c, d, e); // 1 2 3 4 5

[f, g, h] = numbers;
console.log(f, g, h); // 1 2 3
```

You can also skip elements in the array by leaving empty spaces:

```
let numbers = [1, 2, 3, 4, 5];

let [a, , c, , e] = numbers;

console.log(a, c, e); // 1 3 5
```

You can also use the rest operator `...` to capture the remaining elements of an array:

```
let numbers = [1, 2, 3, 4, 5];

let [a, b, ...rest] = numbers;

console.log(a, b); // 1 2
console.log(rest); // [3, 4, 5]
```

5.2 Object Destructuring

Object destructuring allows you to extract values from an object and assign them to variables in a single statement.

```
let person = { name: 'Mohamed', age: 30 };

let { name, age } = person;

console.log(name, age); // Mohamed 30
```

You can also use different variable names for the extracted values:

```
let person = { name: 'Mohamed', age: 30 };

let { name: personName, age: personAge } = person;

console.log(personName, personAge); // Mohamed 30
```

You can also provide default values for the variables:

```
let person = { name: 'Mohamed' };

let { name, age = 30 } = person;

console.log(name, age); // Mohamed 30
```

Lets try a more complex example:

```
let person = {
  name: 'Mohamed',
  age: 30,
  address: {
    country: 'USA',
    city: {
      name: 'New York',
      zip: 10001,
    }
  }
};

let {
  name,
  age,
  address: {
    country,
    city: { name: cityName, zip },
  },
} = person;

console.log(name, age, country, cityName, zip); // Mohamed 30 USA New York
↪  10001
```

You can also combine both dot notation and object destructuring:

```
let person = {
  name: 'Mohamed',
  age: 30,
  address: { country: 'USA', city: { name: 'New York', zip: 10001 } },
};

let { zip } = person.address.city;

console.log(zip); // 10001
```

6 this Keyword

The `this` keyword in JavaScript refers to the object it belongs to. It has different values depending on where it is used:

- In a method, `this` refers to the owner object.
- Alone, `this` refers to the global object. In a browser, it refers to the `window` object.
- In a function, `this` refers to the global object too.
- In a function, in strict mode, `this` is `undefined`.
- In an event, `this` refers to the element that received the event. For example, `e.target` is equivalent to `this.target`.
- In an object, `this` refers to the object itself.

In JavaScript, `this` always refers to the "owner" of the function we're executing, or rather, to the object that a function is a method of.

```
let person = {
  firstName: 'Mohamed',
  lastName: 'Ahmed',
  fullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};

console.log(person.fullName()); // Mohamed Ahmed
```

In the example above, `this` refers to the `person` object because the `fullName` function is a method of the `person` object.

If you were to call the `fullName` function without the `person` object:

```
let person = {
  firstName: 'Mohamed',
  lastName: 'Ahmed',
  fullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};

let fullName = person.fullName;
console.log(fullName()); // TypeError: Cannot read properties of undefined
↪  (reading 'firstName')
```

In this case, `this` refers to the global object because the `fullName` function is not a method of the `person` object. Since the global object does not have `firstName` and `lastName` properties, it throws an error.

6.1 this In A Function Inside An Object Method

When strict mode is not used, if we use this in a function inside an object method, it will refer to the global object.

```
let obj = {
  getThis: function () {
    let innerFunc = function () {
      console.log(this);
    };
    innerFunc();
  },
};

obj.getThis(); // window
```

And if we `"use strict"`, `this` will be `undefined`.

People used to solve this problem by using a variable to store the value of `this` before entering the function.

```
let obj = {
  that: this,
  getThis: function () {
    let that = this;
    let innerFunc = function () {
      console.log(that);
    };
    innerFunc();
  },
};

obj.getThis(); // The object itself
```

7 Arrow Functions

Arrow functions are a new way to write functions introduced in ES6. They provide a more concise syntax for writing functions compared to traditional function expressions.

Arrow functions have the following syntax:

```
let add = (a, b) => a + b;
```

This is equivalent to the following traditional function expression:

```
let add = function(a, b) {
  return a + b;
};
```

Arrow functions have the following features:

- They have a more concise syntax compared to traditional function expressions.
- They do not have their own `this`. They inherit these from the surrounding code.

Here are some examples of arrow functions:

```
// Single parameter
let square = x => x * x;

// Multiple parameters
```

```
let add = (a, b) => a + b;

// No parameters
let greet = () => 'Hello, World!';

// Multiple statements
let sum = (a, b) => {
  let result = a + b;
  return result;
};
```

Some Notes:

- With single parameter you can ignore the parentheses of the parameter.
- With one statement you can ignore the curly braces and the `return` keyword.
- With no parameters you can use empty parentheses.
- With multiple statements you need to use curly braces and the `return` keyword.
- With multiple parameters you need to use parentheses.

7.1 `this` & Arrow Functions

We mentioned earlier that arrow functions do not have their own `this`, they inherit `this` from the surrounding code.

This will help us solve the problem mentioned earlier in the section about `this` in a function inside an object method.

```
let obj = {
  getThis: function () {
    let innerFunc = () => {
      console.log(this);
    };
    innerFunc();
  },
};

obj.getThis(); // The object itself
```

Now we don't need to use a variable to store the value of `this` before entering the function.

8 Set

`Set` was introduced in ES6. A `Set` is a collection of **unique values**. It is similar to an array, but it **does not allow duplicate elements**.

You can create a `Set` by passing an array of values to the `Set` constructor:

```
let set = new Set([1, 2, 3, 4, 5, 1, 2, 3]); // Duplicate values are
↪  removed

console.log(set); // Set(5) { 1, 2, 3, 4, 5 }
```

8.1 Set & Array

You can convert a `Set` to an array using the `Array.from` method:

```
let set = new Set([1, 2, 3]);

let arr = Array.from(set);

console.log(arr); // [1, 2, 3]
```

You can also convert an array to a `Set` using the `Set` constructor:

```
let arr = [1, 2, 3, 4, 5, 1, 2, 3];

let set = new Set(arr);

console.log(set); // Set(5) { 1, 2, 3, 4, 5 }
```

8.2 Set Methods

8.2.1 add

You can add values to a `Set` using the `add` method:

```
let set = new Set();

set.add(1);
set.add(2);
set.add(3);

// Or you can chain the add method
set.add(1).add(2).add(3);

console.log(set); // Set(3) { 1, 2, 3 }
```

8.2.2 size

You can get the number of elements in a `Set` using the `size` property:

```
let set = new Set([1, 2, 3]);

console.log(set.size); // 3
```

8.2.3 has

You can check if a `Set` contains a value using the `has` method:

```
let set = new Set([1, 2, 3]);

console.log(set.has(1)); // true
console.log(set.has(4)); // false
```

8.2.4 delete

You can remove values from a `Set` using the `delete` method:

```
let set = new Set([1, 2, 3]);

set.delete(2);

console.log(set); // Set(2) { 1, 3 }
```

9 Map

`Map` was introduced in ES6. A `Map` is a collection of **key-value pairs**. It is similar to an object, but it has some key differences:

- The keys in a `Map` can be of any type, while the keys in an object are always strings.
- The keys in a `Map` preserve the order in which they were inserted, while the keys in an object do not.
- The size of a `Map` can be easily determined using the `size` property.
- You can easily iterate over the keys and values in a `Map`.
- You can remove an entry from a `Map` using the `delete` method.

Similar to `Set`, you can create a `Map` by passing an array of key-value pairs to the `Map` constructor:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

console.log(map); // Map(2) { 'name' => 'Mohamed', 'age' => 30 }
```

9.1 Map & Object

You can convert an object to a `Map` using the `Map` constructor and the `Object.entries` method:

```
let obj = { name: 'Mohamed', age: 30 };

let map = new Map(Object.entries(obj));

console.log(map); // Map(2) { 'name' => 'Mohamed', 'age' => 30 }
```

You can also convert a `Map` to an object using the `Object.fromEntries` method:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

let obj = Object.fromEntries(map);

console.log(obj); // { name: 'Mohamed', age: 30 }
```

9.2 Map Methods

9.2.1 set

You can add key-value pairs to a Map using the set method:

```
let map = new Map();

map.set('name', 'Mohamed');
map.set('age', 30);

// Or you can chain the set method
map.set('name', 'Mohamed').set('age', 30);

console.log(map); // Map(2) { 'name' => 'Mohamed', 'age' => 30 }
```

9.2.2 size

You can get the number of key-value pairs in a Map using the size property:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

console.log(map.size); // 2
```

9.2.3 keys & values

You can get the keys and values of a Map using the keys and values methods:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

console.log(map.keys()); // MapIterator { 'name', 'age' }
console.log(map.values()); // MapIterator { 'Mohamed', 30 }
```

9.2.4 has

You can check if a Map contains a key using the has method:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

console.log(map.has('name')); // true
console.log(map.has('gender')); // false
```

9.2.5 delete

You can remove key-value pairs from a Map using the delete method:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

map.delete('age');

console.log(map); // Map(1) { 'name' => 'Mohamed' }
```

9.2.6 clear

You can remove all key-value pairs from a `Map` using the `clear` method:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

map.clear();

console.log(map); // Map(0) {size: 0}
```

9.2.7 entries

You can get the key-value pairs of a `Map` using the `entries` method:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);


console.log(map.entries()); // MapIterator {'name' => 'Mohamed', 'age' =>
↪  30}
```

9.3 Map Iteration

You can iterate over the key-value pairs of a `Map` using the `for...of` method:

9.3.1 Iterating Over Entries

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);


for (const entry of map) {
  console.log(entry);
}
```

9.3.2 Iterating Over Keys

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

for (const key of map.keys()) {
  console.log(key);
}
```

9.3.3 Iterating Over Values

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

for (const value of map.values()) {
  console.log(value);
}
```

9.3.4 Iterating With Destructuring

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

for (const [key, value] of map) {
  console.log(key, value);
}
```

10 Summary

In this session, we covered the following topics:

- `"use strict"` which is a way to use a restricted variant of JavaScript that would catch common coding problems and throw exceptions.
- `let` and `const` which are new ways to declare variables in ES6.
- Default parameter values which allow you to specify a default value for a parameter in a function if no argument is provided.
- Template literals which are a new way to create strings in ES6.
- Destructuring assignment which allows you to extract values from arrays or objects and assign them to variables in a more concise way.
- Arrow functions which are a new way to write functions in ES6.
- `this` keyword which refers to the object it belongs to.
- `Set` which is a collection of unique values.
- `Map` which is a collection of key-value pairs.

Session 25

Mohamed Emary

July 1, 2024

1 Spread Operator

Spread operator is a new feature in ES6 that allows you to expand an iterable like an array or an object into individual elements. It is denoted by three dots `...` and can be used in a variety of ways.

1.1 Spread in Arrays

The spread operator can be used to expand an array into individual elements. This is useful when you want to pass the elements of an array as arguments to a function.

```
function sum (a, b, c) {
  return a + b + c;
}
const numbers = [1, 2, 3];
console.log(sum(...numbers)); // 6
```

In this example if you didn't use the spread operator and passed the array directly to the `sum` function, the values of `a, b, c` would be `a = [1, 2, 3]`, `b = undefined`, `c = undefined`.

You can also use the spread operator to combine arrays.

```
const numbers1 = [1, 2, 3];
const numbers2 = [4, 5, 6];
const combined = [...numbers1, ...numbers2];
console.log(combined); // [1, 2, 3, 4, 5, 6]
```

1.2 Spread in Objects

The spread operator can also be used to copy the properties of an object into a new object.

```
const obj1 = {
  name: 'Mohamed',
  age: 30
};
const obj2 = {
  city: 'Cairo',
```

```
  country: 'Egypt'
};

const combined = { ...obj1, ...obj2 };
console.log(combined); // { name: 'Mohamed', age: 30, city: 'Cairo',
↪  country: 'Egypt' }
```

1.3 Rest Parameter

The spread operator can also be used to collect multiple arguments into an array. This is called the rest parameter.

```
function sum (...numbers) {
  let total = 0;
  for (let number of numbers) {
    total += number;
  }
  return total;
}

let numbers = [1, 2, 3, 4, 5];
console.log(sum(...numbers)); // 15
```

2 Shallow Copy vs Deep Copy

To understand the difference between shallow copy and deep copy, let's first understand how JavaScript stores values in memory.

JavaScript uses two data structures to store values: the **stack** and the **heap**.

- The **stack** is used to store **primitive** values like numbers, strings, and booleans.
- The **heap** is used to store **non-primitive** values like objects, arrays, and functions.

When you assign a **primitive** value to a variable, the variable **stores the actual value**. When you assign a **non-primitive** value to a variable, the variable **stores a reference** to the value.

This image shows the difference between the stack and the heap:

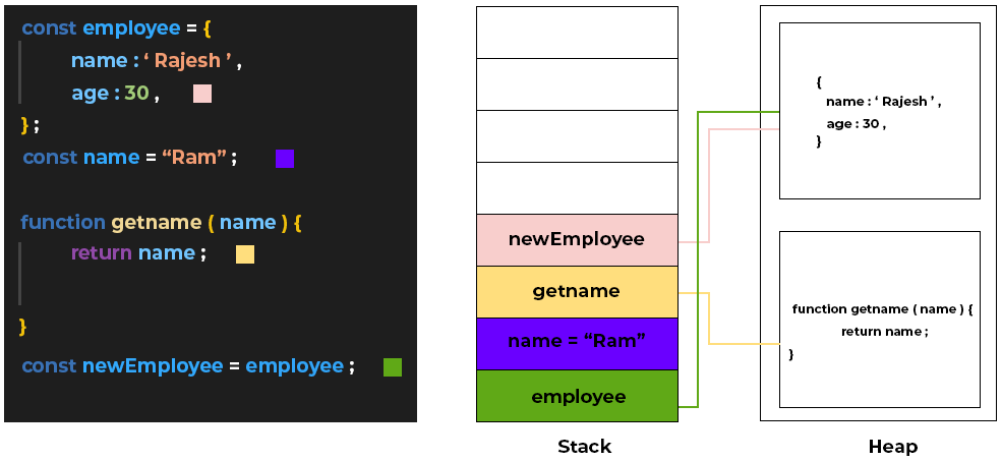

Figure 1: Stack & Heap

Notice that the primitive value `name` is stored directly in the stack, while other non-primitive values like `employee`, `newEmployee` objects, and `getName` function are stored in the heap and the stack stores a reference to them.

Also notice that since the statement `const newEmployee = employee;` is a shallow copy, both `employee` and `newEmployee` point to the same memory location in the heap.

2.1 Shallow Copy

A shallow copy creates a new object that has just a reference to the values of the original object. This means that both objects point to the same memory location and share the same values in memory. So if you change something in the new object, the original object will also change and vice versa.

> We mean by object here both arrays and objects.

```
let original = { name: "Mohamed", age: 30 };
let copied = original;

copied.age = 31;
console.log(original); // { name: "Mohamed", age: 31}
console.log(copied);   // { name: "Mohamed", age: 31}
```

2.2 Deep Copy

A deep copy creates a new object that has a new memory location for each value of the original object. This means that both objects are completely independent of each other. So if you change something in the new object, the original object will not change.

2.2.1 Rest Parameter in Deep Copy

You can use the rest parameter to create a deep copy of an object. This will create a new object with a copy of all **primitive** values.

Consider the following example:

Deep copy:

```
const numbers = [1, 2, 3];
const copy = [...numbers];
numbers[0] = 100;
console.log(copy); // [1, 2, 3]
console.log(numbers); // [100, 2,
↪  3]
```

Shallow copy:

```
const numbers = [1, 2, 3];
const copy = numbers;
numbers[0] = 100;
console.log(copy); // [100, 2, 3]
console.log(numbers); // [100, 2,
↪  3]
```

2.2.2 Non-Primitive Values Inside Non-Primitive Values

If we have a non-primitive value inside another non-primitive value (like another object or an array), the spread operator will only create a shallow copy of the non-primitive value.

```
const obj1 = { name: "Mohamed", address: { city: "Cairo" } };
const obj2 = { ...obj1 };

// Changing a primitive value
```

```
obj1.name = "Ali";
console.log(obj1); // { name: "Ali", address: { city: "Cairo" } }
console.log(obj2); // { name: "Mohamed", address: { city: "Cairo" } }

// Changing a non-primitive value
obj1.address.city = "Alex";
console.log(obj1); // { name: "Ali", address: { city: "Alex" } }
console.log(obj2); // { name: "Mohamed", address: { city: "Alex" } }
```

Notice that the primitive value `name` was a deep copy, while the non-primitive value `address` was a shallow copy.

2.2.3 Deep Copy Using `JSON.parse` and `JSON.stringify`

To create a deep copy of an object that contains non-primitive values, you can use `JSON.parse` and `JSON.stringify`. This will create a new object with a copy of all values (both primitive and non-primitive).

This method works by converting the object to a string and then back to an object.

```
const obj1 = { name: "Mohamed", address: { city: "Cairo" } };
const obj2 = JSON.parse(JSON.stringify(obj1));

obj1.address.city = "Alex";

console.log(obj1); // { name: "Mohamed", address: { city: "Alex" } }
console.log(obj2); // { name: "Mohamed", address: { city: "Cairo" } }
```

2.2.4 Deep Copy Using `structuredClone`

Another way to create a deep copy of an object is to use the `structuredClone` method.

```
const obj1 = { name: "Mohamed", address: { city: "Cairo" } };
const obj2 = structuredClone(obj1);

obj1.address.city = "Alex";

console.log(obj1); // { name: "Mohamed", address: { city: "Alex" } }
console.log(obj2); // { name: "Mohamed", address: { city: "Cairo" } }
```

3 Higher-Order Functions

A higher-order function is a function that takes one or more functions as arguments or returns a function as its result.

Higher-order functions take anonymous functions or arrow functions as arguments and use them to perform some operation.

Examples of higher-order functions in JavaScript include:

- `forEach`
- `map`
- `filter`

- `reduce`
- `find`

3.1 `forEach`

The `forEach` method is used to iterate over an array and execute a function for each element.

```
const numbers = [1, 2, 3, 4, 5];

// Using anonymous function
numbers.forEach(function (number) {
  console.log(number);
});

// Using arrow function
numbers.forEach(number => console.log(number));
```

The code above is equivalent to the following:

```
for (let number of numbers) {
  console.log(number);
}
```

Example of getting the sum of an array using `forEach`:

```
const numbers = [1, 2, 3, 4, 5];
let sum = 0;

numbers.forEach(number => sum += number);

console.log(sum); // 15
```

Example using it with `getElementsByTagName`:

Suppose you have the following HTML:

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```

You can use `querySelectorAll` to select all the list items and then use `forEach` with `addEventListener` to add a click event to each item.

```
const items = document.getElementsByTagName('li');

items.forEach(item => item.addEventListener('click', () => {
  console.log(item.textContent);
}));
```

If we use another parameter with `item` in the arrow function, it will be the index of the item in the array `(item, index) => ...`.

Note:

The `forEach` works with NodeLists but not with HTMLCollections. If you want to use `forEach` with `getElementsByTagName`, you need to convert the HTMLCollection to an array first or just use `querySelectorAll` instead.

3.2 map

The `map` method is used to create a new array by applying a function to each element of an existing array. The new array will have the same length as the original array.

```
let numbers = [1, 2, 3, 4, 5];
let doubled = numbers.map(number => number * 2);
console.log(doubled); // [2, 4, 6, 8, 10]
```

Notice that the original array `numbers` has not been modified.

Another Example with objects:

```
let products = [
  { name: 'iPhone', price: 1000 },
  { name: 'iPad', price: 500 },
  { name: 'MacBook', price: 2000 }
];

let prices = products.map(product => {
  return `${product.name} Price is $${product.price}`;
});

console.log(prices); // ["iPhone Price is $1000", ....]
```

3.3 filter

The `filter` method is used to create a new array with all elements that pass the test implemented by the provided function.

```
let numbers = [1, 2, 3, 4, 5];
let even = numbers.filter(number => number % 2 === 0);
console.log(even); // [2, 4]
```

3.4 reduce

The `reduce` method is used to reduce an array **to a single value**. It executes a reducer function on each element of the array, resulting in a single output value.

The reducer function takes four arguments:

1. Accumulator
2. Current Value
3. Current Index
4. Source Array

```
let numbers = [1, 2, 3, 4, 5];
let sum = numbers.reduce((acc, curr) => acc + curr, 0);
console.log(sum); // 15
```

Notice that the `reduce` method takes an initial value as the second argument. In this case, the initial value is `0`, if you don't provide an initial value, the first element of the array will be used as the initial value.

The reducer function which is `(acc, curr) => acc + curr` takes two arguments: `acc` which is the accumulator and `curr` which is the current value.

3.5 `find`

The `find` method is used to return the first element in an array that satisfies a provided function. It returns `undefined` if no element satisfies the function.

It's similar to the `filter` method, but the difference is that `filter` returns an array of all elements that satisfy the function, while `find` returns only the first element that satisfies the function.

```
let words = ['apple', 'banana', 'cherry'];
let found = words.find(word => word.length > 5);
console.log(found); // 'banana'
```

4 Prototype

When you log an object in JavaScript, you may have noticed a property you didn't create called `[[Prototype]]`. This property is related to JavaScript's prototype-based inheritance system.

1. Every object in JavaScript has an internal property called `[[Prototype]]`.
2. This property is a reference to another object, which is the prototype of the current object.
3. The prototype object is used in the prototype chain, which is a mechanism for implementing inheritance in JavaScript.
4. When you try to access a property or method on an object, JavaScript first looks for it on the object itself. If it's not found, it looks up the prototype chain until it finds the property or reaches the end of the chain (usually `Object.prototype`).
    1. So if there is a property with the same name in the object and its prototype, the object's property will be used.

For example:

```
let obj = {};
console.log(obj);
```

Figure 2: Prototype Object

The `[[Prototype]]` you see here is actually pointing to `Object.prototype`, which is the base prototype for all JavaScript objects.

4.1 Prototypal Inheritance

Suppose you have the following objects:

```
let person = {
  name: 'Mohamed',
  age: 30
};

let employee = {
  salary: 1000
};
```

If you want to make `employee` inherit the properties of `person`, you can set the prototype of `employee` to `person` using the `Object.setPrototypeOf` method.

Syntax of `Object.setPrototypeOf`: `Object.setPrototypeOf(object, prototype)`

```
Object.setPrototypeOf(employee, person);
```

Now, `employee` will have access to the properties of `person`.

```
console.log(employee.name); // 'Mohamed'
console.log(employee.age);  // 30
console.log(employee.salary); // 1000

console.log(employee);
```

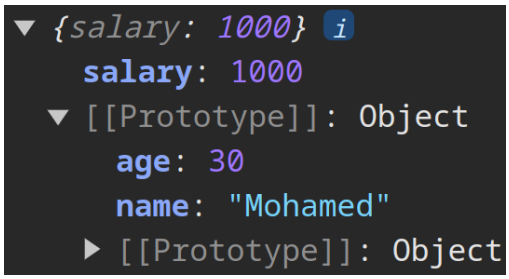

Figure 3: Inheritance Example

Important Notes:

When using `Object.setPrototypeOf`, you can only set one prototype for an object, and if you set another prototype, it will override the previous one.

You can make a **chain of prototypes** by setting the prototype of an object to another object that has a prototype. The resulting object will have the properties of all the prototypes in the chain.

```
let person = { name: "Mohamed" };
let employee = { salary: 1000 };
let manager = { department: "IT" };

Object.setPrototypeOf(employee, person);
Object.setPrototypeOf(manager, employee);
```

```
console.log(manager.name); // Mohamed
console.log(manager.salary); // 1000
console.log(manager.department); // IT
```

Another important thing to note is that you can't set two objects as prototypes for each other because it will create a circular reference.

```
let person = { name: 'Mohamed' };
let employee = { salary: 1000 };

Object.setPrototypeOf(employee, person);
Object.setPrototypeOf(person, employee);
```

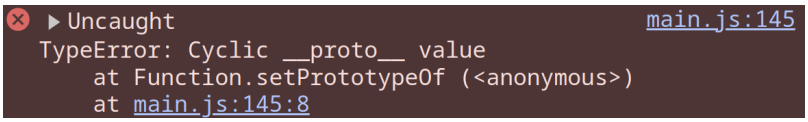

Figure 4: Circular Refference Error

4.2 `Object`

When you create an array in JavaScript, it inherits from `[[Prototype]]` by default.

The `[[Prototype]]` gives the array access to all the methods and properties of the `Array` object.

You can override the prototype of an array object making functions like `push`, `pop`, etc. unavailable.

```
let arr = [];
let obj = {};

object.setPrototypeOf(arr, obj);
arr.push(1); // TypeError: arr.push is not a function
```

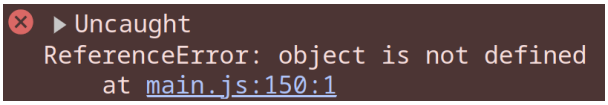

Figure 5: Array Prototype

The same happens with strings, numbers, and booleans. They all have a prototype that gives them access to methods and properties.

5 Summary

- The spread operator `...` is used to expand an iterable like an array or an object into individual elements.
- The spread operator can be used to pass the elements of an array as arguments to a function or to combine arrays.
- The spread operator can also be used to copy the properties of an object into a new object.
- The rest parameter is used to collect multiple arguments into an array.
- A shallow copy creates a new object that has just a reference to the values of the original object, while a deep copy creates a new object with a new memory location for each value of the original object.
- You can use `JSON.parse` and `JSON.stringify` to create a deep copy of an object that contains non-primitive values.
- You can use the `structuredClone` method to create a deep copy of an object.
- Higher-order functions are functions that take one or more functions as arguments or return a function as their result.
- Examples of higher-order functions in JavaScript include `forEach`, `map`, `filter`, `reduce`, and `find`.
    - The `forEach` method is used to iterate over an array and execute a function for each element.
    - The `map` method is used to create a new array by applying a function to each element of an existing array.
    - The `filter` method is used to create a new array with all elements that pass the test implemented by the provided function.
    - The `reduce` method is used to reduce an array to a single value.
    - The `find` method is used to return the first element in an array that satisfies a provided function.
- Every object in JavaScript has an internal property called `[[Prototype]]`, which is a reference to another object that is the prototype of the current object.
- The prototype object is used in the prototype chain, which is a mechanism for implementing inheritance in JavaScript.
- You can use `Object.setPrototypeOf` to set the prototype of an object to another object.
- You can create a chain of prototypes by setting the prototype of an object to another object that has a prototype.
- You can't set two objects as prototypes for each other because it will create a circular reference.
- When you create an array in JavaScript, it inherits from `[[Prototype]]` by default, giving it access to all the methods and properties of the `Array` object.

Session 26

Mohamed Emary

July 4, 2024

1 Revision

1.1 Higher Order Functions

Suppose I have a collection of products:

1. To get the total cost of all products we can use the `reduce` function.
2. To get products with a price greater than 1000 we can use the `filter` function.
3. To get the first product with a price greater than 1000 we can use the `find` function.
4. To add 10% tax to all products we can use the `map` function.

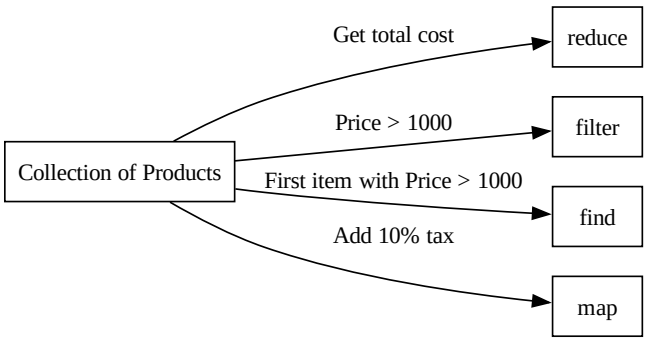

Figure 1: Higher Order Functions

1.2 Prototype

Objects in JS inherit properties from other objects. This is called prototypal inheritance. Every object has a prototype object, which acts as a template object that it inherits methods and properties from.

The prototype object can either be another object or `null`. If it is `null`, the object has no prototype and therefore does not inherit any properties or methods.

Array object inherits from `Array.prototype` and `Array.prototype` inherits from `Object.prototype` which is the root object and its prototype is `null`.

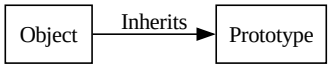

Figure 2: Prototype

The same with numbers, strings, and booleans.

2 Object Oriented Programming (OOP)

OOP is a programming paradigm based on objects. It makes your code more organized, easier to read, and maintain.

The most important advantage of OOP is that it simulates the real world. It allows you to break down your software into smaller parts, making it easier to solve complex problems.

For example to model a hospital management system, you can create classes for `Doctor`, `Patient`, `Nurse`, `Receptionist`, etc.

There are two main ways to implement OOP:

- **Class based OOP** (Most Programming Languages): In this paradigm, we use classes to define objects. A class is a blueprint for creating objects. An object is an instance of a class.
- **Prototype based OOP** (Used in JS): In this paradigm, we use prototypes to define objects. A prototype is a template object that an object inherits properties and methods from.

2.1 Class based OOP

In class based OOP, we create classes for the important entities in our application, and then we create objects from these classes.

For example, to model the hospital management system, we can create a class for `Doctor` and then create an object for each doctor, same with `Patient`, `Nurse`, etc.

Each doctor should have a `name`, `age`, and `salary`. We can define these properties in the `Doctor` class.

2.2 Prototype based OOP

Consider this function:

```
1 function Doctor(name, age, salary) {
2   let doctor = {};
```

```
  doctor.name = name;
  doctor.age = age;
  doctor.salary = salary;
  return doctor;
}
let drAhmed = Doctor('Ahmed', 30, 5000);
```

In this example, `Doctor` is a function that creates a doctor object.

2.2.1 Constructor Functions

In JS, we use constructor functions to create objects. A constructor function is like a blueprint for creating objects.

```
function Doctor(name, age, salary) {
  this.name = name;
  this.age = age;
  this.salary = salary;
}
let drAhmed = new Doctor('Ahmed', 30, 5000);
```

This is the same as the previous example, but we use the `new` operator to create the object, and `this` to refer to the object being created.

To create functions that are shared between all objects created from a constructor function, we use the `prototype` property.

```
function Doctor(name, age, salary) {
  this.name = name;
  this.age = age;
  this.salary = salary;
}

Doctor.prototype.sayHi = function() {
  console.log('Hi, I am ' + this.name);
};

Doctor.prototype.getSalary = function() {
  console.log('My salary is ' + this.salary);
};

let drAhmed = new Doctor('Ahmed', 30, 5000);
let drAli = new Doctor('Ali', 35, 6000);

drAhmed.sayHi(); // Hi, I am Ahmed
drAli.sayHi(); // Hi, I am Ali
```

But why not to just add the functions directly to the object like what we did with the properties?

- Because if we have many objects, each object will have its own copy of the function which is a waste of memory.
- With properties it's fine because each object has its own values for the properties, for example the `name` is different for each object, but with functions it's the same implementation

for all objects, so no need to have a copy for each object.

2.2.2 Sugar Syntax

In ES6, we have a sugar syntax for creating classes, but under the hood, it's the same as the constructor functions.

```
class Doctor {
  constructor(name, age, salary) {
    this.name = name;
    this.age = age;
    this.salary = salary;
  }

  sayHi() {
    console.log('Hi, I am ' + this.name);
  }

  getSalary() {
    console.log('My salary is ' + this.salary);
  }
}

let drAhmed = new Doctor('Ahmed', 30, 5000);
let drAli = new Doctor('Ali', 35, 6000);

drAhmed.sayHi(); // Hi, I am Ahmed
drAli.sayHi(); // Hi, I am Ali
```

This is the same as the previous example, but with a different syntax.

Important things to notice:

- The constructor function is called `constructor` in the class.
- We don't use the `function` keyword before the functions.
- If we want to use arrow functions, we can use for example `sayHi = () => { ... }`.
- When creating an object from a class, we use the `new` keyword.
- The first thing to get executed when creating an object from a class is the `constructor`.

3 Main OOP Concepts

3.1 Inheritance

Inheritance is a mechanism that allows you to eliminate redundant code by reusing existing classes. It allows you to create a new class that is based on an existing class.

For example, we can create a class for `Employee` and then create a class for `Doctor` that inherits from `Employee`.

```
class Employee {
  constructor(name, age) {
    this.name = name;
```

```
    this.age = age;
  }

  sayHi() {
    console.log('Hi, I am ' + this.name);
  }
}

class Doctor extends Employee {
  constructor(name, a   ge, salary) {
    super(name, age);
    this.salary = salary;
  }

  getSalary() {
    console.log('My salary is ' + this.salary);
  }
}

let drAhmed = new Doctor('Ahmed', 30, 5000);
drAhmed.sayHi(); // Hi, I am Ahmed
drAhmed.getSalary(); // My salary is 5000
```

In this example, `Doctor` inherits from `Employee`. The `Doctor` class has access to the `sayHi` function from the `Employee` class.

Notice that `Doctor` has its own constructor function, but it calls the `Employee` constructor function using `super`.

If you log the `drAhmed` object, you will see that it has all the properties of `Employee` and `Doctor` directly in the object. You will also notice that the prototype of `Doctor` has both functions `sayHi` and `getSalary`.

Multiple Inheritance

You can't inherit from multiple classes in JS, however you can make a class inherit from another class that inherits from another class.

---

for example if you have `Employee` and `Doctor` and you want to create a class `Surgeon` that inherits from both, you can make `Doctor` inherit from `Employee` and then make `Surgeon` inherit from `Doctor`.

```
class Surgeon extends Doctor, Employee { // Syntax Error
```

If we have the same variable or function in both the parent and child class, the variable or function in the child class will override the variable or function in the parent class (Polymorphism).

3.2 Polymorphism

Polymorphism is a feature that allows you to use a single interface to represent different data types.

For example, we can have a `sayHi` function in the `Employee` class and a `sayHi` function in the `Doctor` class. When we call `sayHi` on a `Doctor` object, it will call the `sayHi` function in the

`Doctor` class, not the one in the `Employee` class.

```
class Employee {
  sayHi() {
    console.log('Hi, I am an employee');
  }
}

class Doctor extends Employee {
  sayHi() {
    console.log('Hi, I am a doctor');
  }
}

let drAhmed = new Doctor();
drAhmed.sayHi(); // Hi, I am a doctor
```

Polymorphism has two types:

- **Overloading**: Same function name with different parameters.
- **Overriding**: Same function name with the same parameters. (In JS we only have overriding).

3.2.1 Access Modifiers

Access modifiers are keywords that set the accessibility of properties and methods in a class.

- **Public**: Accessible from anywhere. (Default).
- **Private**: Not accessible from outside the class. (use `#` before the property or method name ex: `this.#name`).
- **Protected**: Accessible within the class and its subclasses. (Not available in JS).

```
class Employee {
  #name; // Private property

  constructor(name, age) {
    this.#name = name;
    this.age = age;
  }

  #sayHi() { // Private method
    console.log('Hi, I am ' + this.#name);
  }

  sayHi() {
    this.#sayHi();
  }
}

let emp = new Employee('Ahmed', 30);
console.log(emp.age); // 30
```

```
// console.log(emp.#name); // Syntax Error
emp.sayHi(); // Hi, I am Ahmed
```

Notice that we can't access the private property `#name` from outside the class.

4 Modules

Modules are a way to split your code into multiple files. Each file is a module that *exports* some functions or variables that can be *imported* in other files.

To use modules in JS, we use the `export` and `import` keywords, we also use the `type="module"` attribute in the script tag.

To be able to use modules in the browser, you need to use a live server.

```
// variables.js
export let name = 'Ahmed';
export let age = 30;

// script.js
import { name, age } from './variables.js';
console.log(name); // Ahmed
console.log(age); // 30
```

In `index.html`:

```
<script type="module" src="script.js"></script>
```

The same can be done with functions and classes, just add `export` before the function or class, and `import` it in the other file.

When exporting multiple things, you can use `export { name, age }` in the end of the file, and when importing you can use `import * as vars from './variables.js'` to import all the exported things in an object called `vars`.

```
// info.js
let name = 'Ahmed';
let age = 30;
let sayHi = function() {
  console.log('Hi, I am ' + name);
};

export { name, age, sayHi };

// script.js
import * as info from './info.js';
console.log(info.name); // Ahmed
console.log(info.age); // 30
info.sayHi(); // Hi, I am Ahmed
```

We can use `export default` to export something as the default export, and when importing we can import it without the curly braces and with any name we want.

```
// info.js
let name = 'Ahmed';
```

```
let age = 30;
let sayHi = function() {
  console.log('Hi, I am ' + name);
};

export { name, age, sayHi };
export default sayHi;

// script.js
import greet from './info.js'; // We can use any name to import the
 ↪  default export
greet(); // Hi, I am Ahmed
```

To import the other exports along with the default export, you can use `import greet, { name, age } from './info.js';`. Notice that `name`, `age` are in curly braces and have the same name as the exported variables.

5 Summary

- Higher order functions are functions that take other functions as arguments or return functions.
- In JS, objects inherit properties and methods from other objects using prototypal inheritance.
- OOP is a programming paradigm based on objects. It simulates the real world and makes your code more organized.
- There are two main ways to implement OOP: class based OOP and prototype based OOP.
- In class based OOP, we use classes to define objects. A class is a blueprint for creating objects.
- In prototype based OOP, we use prototypes to define objects. A prototype is a template object that an object inherits properties and methods from.
- Inheritance is a mechanism that allows you to eliminate redundant code by reusing existing classes.
- Polymorphism is a feature that allows you to use a single interface to represent different data types.
- Access modifiers are keywords that set the accessibility of properties and methods in a class.
- Modules are a way to split your code into multiple files. Each file is a module that exports some functions or variables that can be imported in other files.
- To use modules in JS, use the `export` and `import` keywords, and use the `type="module"` attribute in the script tag.
- You can export multiple things using `export { name, age }` and import them using `import * as vars from './variables.js'`.
- You can export something as the default export using `export default` and import it without the curly braces and with any name you want.