

WEB BASICS & HTML

Web-Design und Programmierung



What's the Difference?



HTML
Hypertext Markup Language

Create the structure

- Controls the layout of the content
- Provides structure for the web page design
- The fundamental building block of any web page



CSS
Cascading Style Sheet

Stylize the website

- Applies style to the web page elements
- Targets various screen sizes to make web pages responsive
- Primarily handles the "look and feel" of a web page



Javascript

Increase interactivity

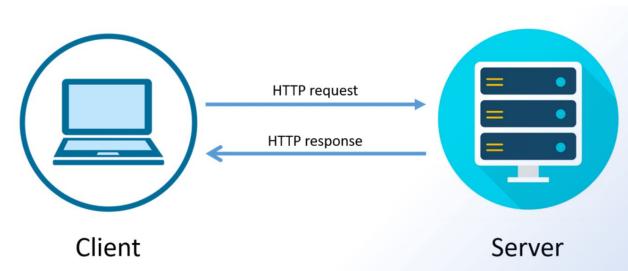
- Adds interactivity to a web page
- Handles complex functions and features
- Programmatic code which enhances functionality

HOW DOES THE WEB WORK?

- World Wide Web
 - Request Response Model
 - HTTP
-

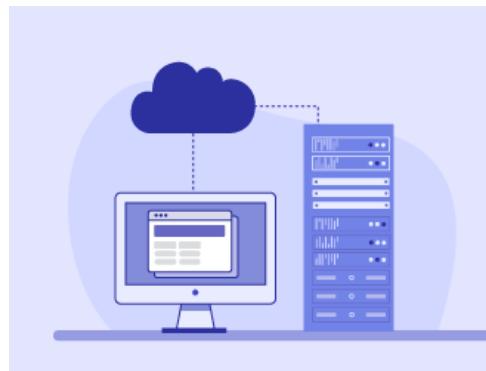
WORLD WIDE WEB

- Web is a subset of the Internet
- Web contains of **pages** which can be **requested** from **servers** and viewed with a browser
- Web contains of multiple protocols/technologies like HTML, CSS, JS, HTTP(S), ...
- Request/Response communication model
 - Client sends request
 - Server processes request and returns response
 - No “push messages” from server



WEB SERVER

- Web server: Software that provides information via HTTP/HTTPS (port 80/443)
- Web server directory
 - Default directory of the domain (htdocs in case of Apache)



What is a
Web Server?

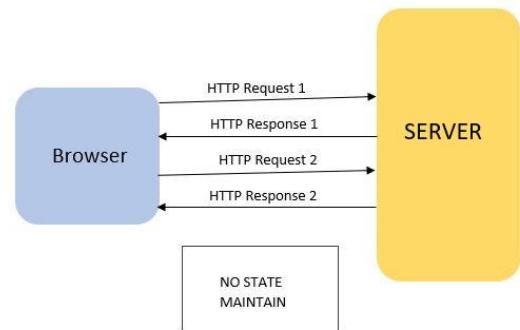
[*Web sur· vr*] ☞

A software program that hosts and delivers web content to clients via HTTP.

HYPertext Transfer Protocol



- De facto standard for communication protocol on the web
- **Text-based** (= non-binary)
- Request/response concept
 - Client sends a request to the server
 - Server sends response back to client
- **Stateless**
 - Server handles all client requests completely independently of each other
- **Connectionless**
 - No permanent connection between client and server required



HYPERTEXT TRANSFER PROTOCOL - REQUEST

- Request

```
GET /campus-hagenberg HTTP/1.1  
Host: www.fh-ooe.at  
[...]
```

- Request:

- Method („Verb“)
- Request-URI
- Protocol version

- Header

- Blank line (obligatory!)
- Body (optional)



HYPERTEXT TRANSFER PROTOCOL - REQUEST

- **Verbs**
 - Describes action to be taken for resource identified via URL
- HTTP consists of eight verbs
- **GET:** request a ressource
- **POST:** create a new ressource
- **PUT:** edit a ressource
- **DELETE:** delete a ressource from server
- **OPTIONS:** request information for communication path to a resource
- **HEAD:** request a resource, only Header
- **TRACE:** For test purposes, request message is returned in the body of the response
- **CONNECT:** Tunneling with HTTP-Proxies

UNIFORM RESOURCE IDENTIFIER

- Identifies affected resource
- Absolute path, starting with "/"
- Interpreted by the server and reallocated to the corresponding resource
- Examples:
 - "/"
 - "/index.html"
 - "/images/logo.png"
 - "/research/staff/12"

URL Anatomy

https://example.com:80/blog?search=test&sort_by=created_at#header



Protocol



Domain



Port Path



Query Parameters



Fragment/Anchor

HYPertext Transfer Protocol - Response

- **Status line**, consisting of:
 - Protocol version
 - Status code
 - Status message
 - **Header**
 - **Blank line**(obligatory!)
 - **Body**(optional)
- HTTP/1.0 200 OK
Header-4: Value of header 4
Header-5: Value of header 5
- This data here is called the body of the response and is transferred back to the client. It is again optional.

HYPertext Transfer Protocol – STATUS CODES

- Three-digit number (standardized)
- For machine processing of the response
- Indicates whether the server understood the request message and was able to process it.

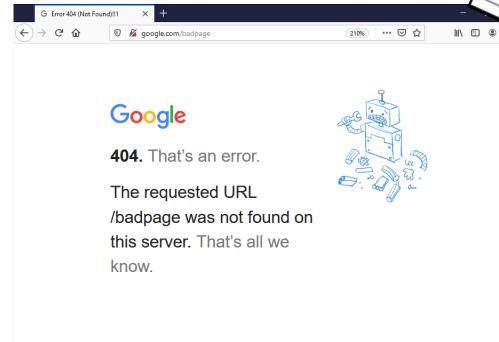
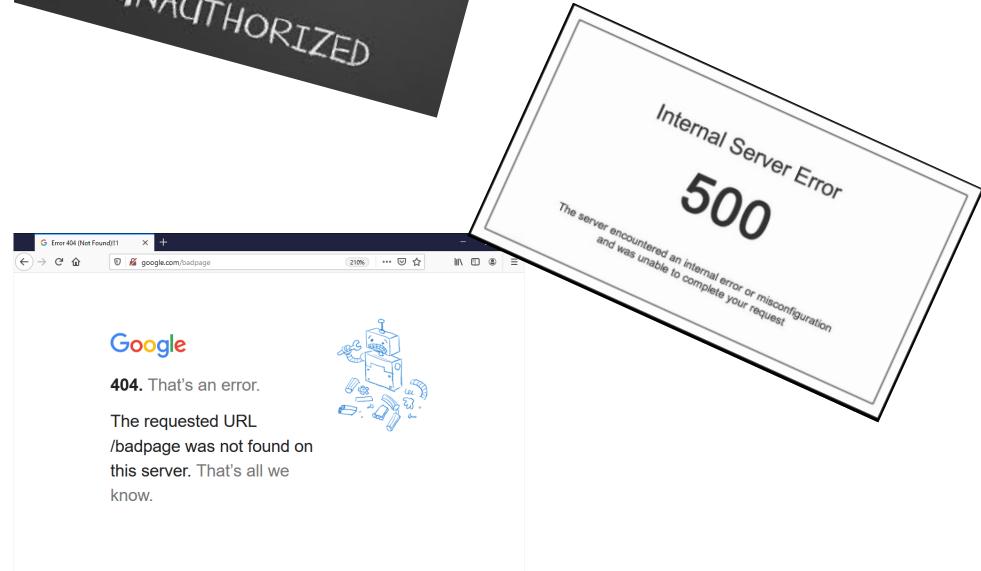
HTTP Status Codes



- **1xx: Information**
 - received Request, Processing (still) in progress
- **2xx: Success**
 - Request received, understood and processed
- **3xx: Forwarding**
 - Request received, additional action required from client
- **4xx: Error on client**
 - Request syntactically incorrect or not valid in terms of content
- **5xx: Error on server**
 - Request valid, error during processing by the server

HYPertext Transfer Protocol – Status Codes

- **HTTP/1.1 400 Bad Request**
 - Syntactically incorrect request message
- **HTTP/1.1 401 Unauthorized**
 - Request requires authentication
- **HTTP/1.1 403 Forbidden**
 - Server has understood request, but refuses execution
- **HTTP/1.1 404 Not Found**
 - Resource is not available
- **HTTP/1.1 405 Method Not Allowed**
 - Used method ("verb") is not supported for the requested resource
- **HTTP/1.1 500 Internal Server Error**
 - Unexpected server error occurred during processing



HYPERTEXT MARKUP LANGUAGE

- Hypertext
 - HTML Elements and attributes
 - Basic elements
-

HYPERTEXT

- Non-linear organization of objects whose net-like structure is created by **logical links between knowledge units** (nodes, e.g., texts or text parts) (reference node concept)
- Hypertext and hypermedia are often used synonymously
- **Hypermedia** often refers to hypertext with multimedia aspects
 - this is the norm today



MARKUP LANGUAGE

- Describes data or procedures for representation
 - Descriptive markup languages
 - Describes **syntax** of data
 - e.g., HTML, XML
 - Procedural markup languages
 - Describes the procedure for generating a representation
 - e.g., TeX, PDF
- Description with “tags”
 - 145 predefined ones in HTML5

HTML VS XML



HTML:

Hypertext Markup Language

Predefined tags

Displays data

In HTML, tags such as '`<h1>`' will always create the same formatted text

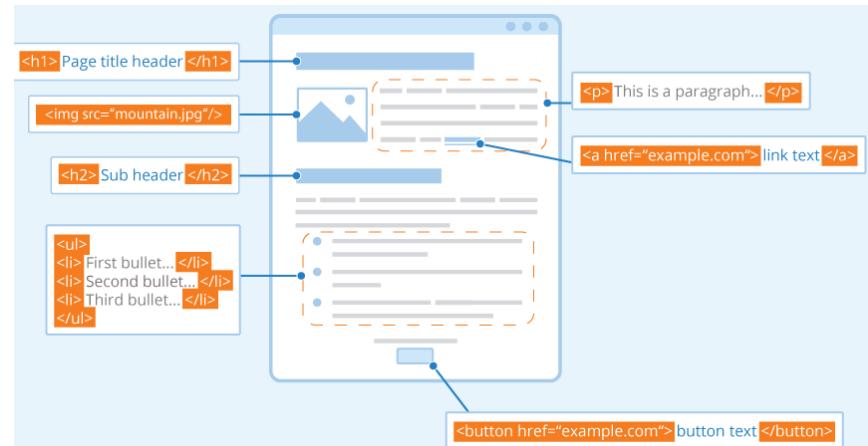
In XML, tags are user defined, meaning that the programmer must define their own tag

XML:

Extensible Markup Language

User defined tags

Carries data



HYPERTEXT MARKUP LANGUAGE

- used to **structure the content** of a web page
- **media-independent** description of interactive content
- HTML holds everything together
 - content, CSS files, JS files, images, ...
- HTML user agents (e.g., browsers) **parse** the markup line by line, turning it into a **DOM** (Document Object Model) tree, which is an in-memory representation of a document.



**HyperText
Markup Language
(HTML)**

[hāp-tē-(,)māk'-el]

The basic scripting language used by web browsers to render pages on the world wide web.

Document type declaration*

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Hello World!</title>
    <meta charset= "utf-8">
  </head>
  <body>
    <p>My first HTML page</p>
  </body>
</html>
```

Head (Title and metatags)

Body (content)

HYPERTEXT MARKUP LANGUAGE

- HTML documents consist of a tree of elements and text.
- Elements have a start tag, such as `<p>`, and an end tag, such as `</p>`.
- Tags have to be nested, such that elements are completely within each other.
- Everything between a start tag and an end tag is called: content of an element.

```
<p>This is some <strong>highlighted</strong> text.</p> 😊  
<p>Please never do <strong>this!</p></strong> 😞
```

The screenshot shows the WebStorm IDE interface. On the left is the project tree with a folder named 'repo_whm' containing 'Hausübungen', 'S24', 'UE', 'VL', 'Animation', 'Bootstrap', 'Einheit 1 - Einführung', 'Einheit 2 - DOM', 'Einheit 3 - jQuery', 'Einheit 4 - UI', 'ES6', 'JSON', 'Objekte', and 'wdp3'. Inside 'wdp3' is an 'index.html' file. The right side shows the code editor with the following content:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>WDP3 HTML Basics</title>
</head>
<body>
    <h1>HTML Basics</h1>
    <p>We are learning some basic HTML and CSS as well as JavaScript!</p>
    <h2>HTML</h2>
    <p>HTML is the standard markup language for creating Web pages.</p>
    <ul>
        <li>This is a very simple list</li>
        <li>It has only two items</li>
    </ul>
    <div>
        <h3>HTML Links</h3>
        <p>HTML links are defined with the a tag:</p>
        <a href="https://www.w3schools.com/html/">Visit our HTML tutorial</a>
    </div>
</body>
</html>
```

At the bottom, the status bar shows 'repo_whm > VL > wdp3 > <index.html>' and the time '19:58'. The bottom right corner indicates 'CRLF'.

Text Editor
(Webstorm)

The screenshot shows a Google Chrome browser window titled 'WDP3 HTML Basics'. The address bar shows 'http://localhost:63342/repo...'. The page content is:

HTML Basics

We are learning some basic HTML and CSS as well as JavaScript!

HTML

HTML is the standard markup language for creating Web pages.

- This is a very simple list
- It has only two items

HTML Links

HTML links are defined with the a tag:

[Visit our HTML tutorial](https://www.w3schools.com/html/)

Browser
(Google Chrome)

HYPERTEXT MARKUP LANGUAGE - HEADER

```
6 <head>
7   <title> html page </title>
8   <meta charset="utf-8">
9   <meta name="viewport" content=
10  <meta name="author" content="johndoe">
11  <meta name="robots" content="index,follow">
12  <link rel="contents" href="#toc">
13 </head>
14 <body>
```

- General data on document
 - Description of content and additional information
 - <meta name="property" name="value" />
 - E.g., <meta name="author" name="maxmustermann" />
 - **description**
 - **keywords** (google and web ranking)
 - **date**
 - Instructions for crawlers: **robots** (follow, index, nofollow, noindex, ...)
 - Inclusion of Stylesheets (good) or JavaScript (bad)
 - <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/meta>

HYPERTEXT MARKUP LANGUAGE - HEADER

```
6 <head>
7   <title> html page </title>
8   <meta charset="utf-8">
9   <meta name="viewport" content=
10  <meta name="author" content=
11  <meta name="robots" content=
12  <link rel="contents" href="#toc">
13 </head>
14 <body>
```

- Web page should contain the following meta information
 - language of the page
 - character encoding of the document.
- The language is defined as lang-attribute of the html-tag:
 - <html lang="en"> or <html lang="de">
- The character set is defined using a meta tag in the page's head section:
 - <meta charset="UTF-8" />
 - Inclusion of special characters (e.g., Emojis) or HTML Entities
 - https://www.w3schools.com/charsets/ref_emoji_smileys.asp

HYPERTEXT MARKUP LANGUAGE – HTML ELEMENTS

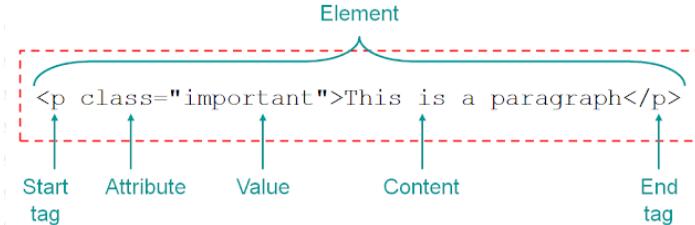
- HTML Body contains the content Elements (**HTML Tags**)
- Types of HTML Elements
 - **Block-Elements** (e.g., `<div>`)
 - Elements start in a new line
 - Structure the content; position can be defined
 - Can contain inline and block elements
 - 100 % width, height according to content
 - **Inline-Elements**
 - Wrap small parts of content
 - Width according to content, height according to line height
 - **Void-Elements or self-closing elements**
 - Elements without “content”
 - E.g., `
`, `<hr/>`
 - Trailing slash is optional
 - **Text**
- Hierarchically structured
- <https://developer.mozilla.org/en-US/docs/Web/HTML/Element>

Block Element vs Inline Element

Block Level Element	Inline Level Element
Begins a new line of text.	Does not begin a new line of text. Text is placed on the same line.
Its width extends beyond the inner content.	Its width only extends as far as the inner content.
You can set the width and height values.	You can't set width and height values.
Can container text, data, inline elements, or other block level element.	Can contain text, data, or other inline elements.

HYPERTEXT MARKUP LANGUAGE – ATTRIBUTES

- Marking objects with properties
 - <`h1 align="center"`>centered headline</`h1`>
- All elements can have attributes
 - Some require specific attributes to actually work (e.g., src-Attribute for `img`)
- Always specified in the **start tag**
- **Types of attributes**
 - Value assignment with predefined values
 - <`input disabled="disabled"`/>
 - Free value assignment, but with data type
 - <`input size="12"`/>
 - Completely free value assignment
 - <`p title="foo"`>
- **Important attributes**
 - id-attribute -> Unique identifier of an Element
 - name-attribute -> Needs not to be unique (used for grouping)
 - Class -> assignment of CSS classes



HYPERTEXT MARKUP LANGUAGE – IMPORTANT GENERAL TAGS

- Headlines
 - <h1>...</h1> - <h6>...</h6>
- Paragraph
 - <p>...</p>
 - May not contain other block elements
- Lists
 - ... (unordered list), ... (ordered list)
 - ... (list item)
 - Change enumeration item
 - For ol → type=“val”, val can be either **I,i,A,a**, or **1** (default 1)
 - For ul type is deprecated (use CSS list-style-type)
 - Can be hierarchically structured

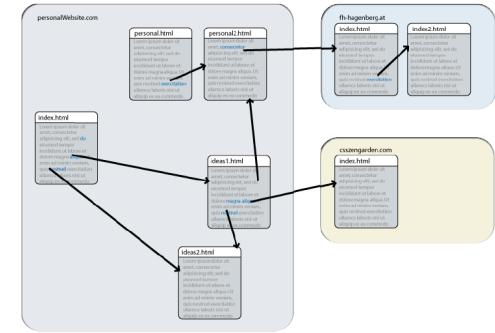
HYPERTEXT MARKUP LANGUAGE – IMPORTANT GENERAL TAGS

- **Line break </br>**
 - Explicit line breaks need to be specified
 - Do not “cascade” line breaks -> use CSS
- **Formatting text**
 - **Logical distinction**
 - ..., ..., <code>...</code>, <samp>...</samp>, <kbd>...</kbd>, <var>...</var>, <cite>...</cite>, <dfn>...</dfn>, <abbr>...</abbr>
 - ... for general marking
 - See <https://wiki.selfhtml.org/wiki/HTML/Tutorials/Textauszeichnung> for details
 - **Physical distinction (presentational)**
 - ..., <i>...</i>, <u>...</u>, <strike>...</strike>, _{...}, ^{...}, <small>...</small>, <big>...</big>
 - Mixes content and presentation -> prefer CSS



HYPERTEXT MARKUP LANGUAGE – IMPORTANT GENERAL TAGS

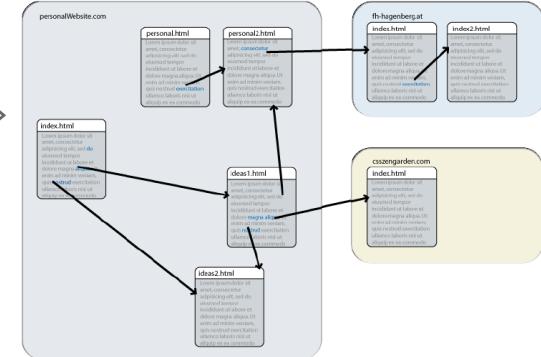
- **Links <a>...**
 - link from one to another HTML document
 - link to an anchor within the same page
 - link to an external resource
 - Requires href Attribute with the links target
 - `This is the anchor text`
- **Reference**
 - **Absolute** (remote or local)
 - Protocol://server.domainname/filename.extension
 - `Link to DuckDuckGo`
 - **Relative**
 - `Introduction`



HYPERTEXT MARKUP LANGUAGE – IMPORTANT GENERAL TAGS

- **Links (cont.)**

- Target-Attribute defines how new page should be presented
 - _self, _parent, _blank, _top
 - Mostly relevant for (old concept of) frames
 - _blank to open new browser tab
- Reference in same document (Anchor)
 - A-tag points to id of an element
 - `Link to anchor`
...
`<h2 id="anchor">Introduction`
- E-Mail
 - `Mail`



HYPertext Markup Language – Important General Tags

- **Horizontal rule** `<hr/>`
- **Images** ``
 - ``
 - Src -> path to image
 - Alt -> Alternative text (accessibility)
 - Check for filetypes
 - https://developer.mozilla.org/en-US/docs/Web/Media/Formats/Image_types
- **Images** can be wrapped with `<figure>...</figure>` tag
 - Allows for according caption

```
<figure>
  
  <figcaption>MDN Logo</figcaption>
</figure>
```

HYPertext Markup Language – Important General Tags

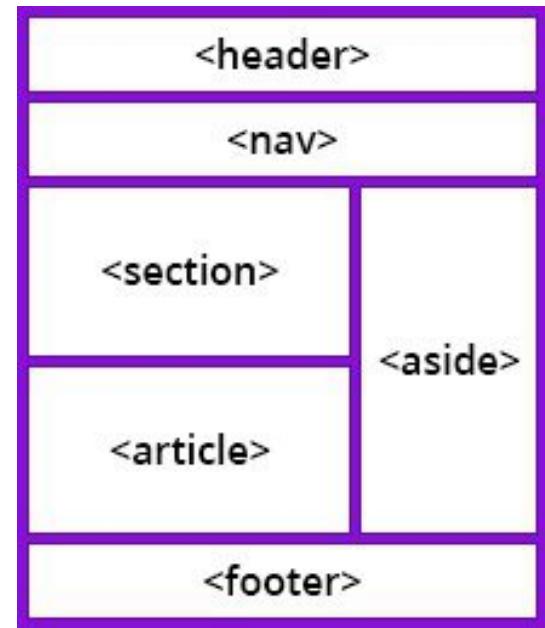
- Picture element <picture>...</picture>
 - Contains zero or more <source> elements and one element to offer alternative versions of an image for different display/device scenarios (responsive design)
 - Browser considers each <source> element and chooses best match
 - No match element → element's src attribute is selected
 - Selected image is then presented in the space occupied by the element
 - Browser checks <source>'s srcset, media, and type attributes to select best match

```
<picture>
  <source srcset="/media/cc0-images/surfer-240-200.jpg"
          media="(orientation: portrait)" />
  
</picture>
```

HYPERTEXT MARKUP LANGUAGE – IMPORTANT GENERAL TAGS

- Elements to (logically) structure page content

- Document Division Element <div>
- Sections
 - Header, Footer <header>, <footer>
 - Section <section>, <article>
 - Sidebar <aside>
 - Navigation <nav>
 - Main <main>



HYPERTEXT MARKUP LANGUAGE – COMMENT

```
<!-- There is nothing more to say here. -->
```

HTML TABLES

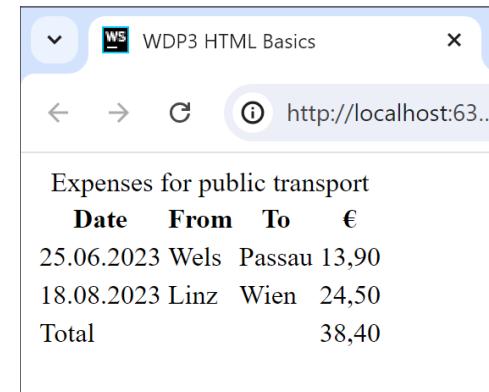
- Table
 - Table header/body
 - Table row/data
-

HTML - TABLES

- Tables <table> are for structuring data
 - Do not use them for layout
- Optional table header <thead>, body <tbody> and footer <tfoot>
- Table rows <tr> and columns <td>
 - For header use <th> (instead of <td>)
- Aggregation of rows/cols using rowspan/colspan attributes
- The (optional) <caption> element is always the first child of a table

```
<table>
  <caption>Expenses for public transport</caption>
  <thead>
    <tr>
      <th>Date</th>
      <th>From</th>
      <th>To</th>
      <th>€</th>
    </tr>
  </thead>
  <tbody>
```

```
    <tr>
      <td>25.06.2023</td>
      <td>Wels</td>
      <td>Passau</td>
      <td>13,90</td>
    </tr>
    <tr>
      <td>18.08.2023</td>
      <td>Linz</td>
      <td>Wien</td>
      <td>24,50</td>
    </tr>
  </tbody>
  <tfoot>
    <tr>
      <td
        colspan="3">Total</td>
      <td>38,40</td>
    </tr>
  </tfoot>
</table>
```



The screenshot shows a web browser window titled "WDP3 HTML Basics" at the URL "http://localhost:63...". The page displays a table with the following data:

Date	From	To	€
25.06.2023	Wels	Passau	13,90
18.08.2023	Linz	Wien	24,50
Total			38,40

HTML FORMS

- Form
 - Form elements
 - Validation and UX
-

HTML FORMS

- Interaction between the user and an application
- Collect and send data to a web server
- Common use cases: login form, contact form, newsletter subscription, conference registration, ordering process, search form, filter...
- Client side => create forms, how to send data
- Server side => receive and process data

HTML FORMS

```
<form action="..." method="post" />  
<!-- form elements -->  
</form>
```

FORM ACTION & METHOD

- Action: URL where form data is sent on submit
- Method: HTTP method to send data with
 - GET: to ask the server for data, to get data
 - Form data is appended to the URL
 - (key-value pairs) => can be bookmarked, will show up in server logs
 - Length of the URL is limited
 - Values are transmitted openly
 - Special characters must be encoded/escaped
 - Use for: search form, filter, pagination,...
 - Example: <https://www.smashingmagazine.com/search/?q=html>
 - POST: to send data to the server
 - Form data is sent via request body
 - Transmitted “invisibly” for the user
 - Own data block – not limited in size like GET
 - Data (files, images, etc.) can also be sent
 - use for: login, contact form, registration,...

INPUT & LABEL

- The `<input>` element is the most common element in a form
- Add the `type` attribute to specify its behavior
 - `text`, `email`, `password`, `number`, `date`, `checkbox`, `radio`, ...
- Add the `<label>` element to explain the input field
 - The input field's `id` and the label's `for` attribute must match

```
<label for="name-input">Enter your username</label>
<input type="text" id="name-input" name="username" />
```

INPUT TYPE

- Inputs with type **text** create basic, single line inputs
- Other types:
 - “date”: Date
 - “datetime”: Date + Time
 - “email”: eMail adress
 - “tel”: Phonenumber
 - “number”: Numerical value
 - “color”: Color chooser
 - “range”: Interval
 - “url”: URL

RADIO BUTTONS

- type="radio"
- Radio buttons are generally used in a set of related options where only one of the options can be selected at a time.
- Grouped radio buttons must have the same **name** attribute, the value is used to identify the selected option.

```
<input type="radio" name="campus" id="hagenberg" value="hagenberg" />
<label for="hagenberg">Hagenberg</label>
<input type="radio" name="campus" id="linz" value="linz" checked />
<label for="linz">Linz</label>
<input type="radio" name="campus" id="steyr" value="steyr" />
<label for="steyr">Steyr</label>
<input type="radio" name="campus" id="wels" value="wels" />
<label for="wels">Wels</label>
```

CHECKBOXES

- type="checkbox"
- Checkboxes are used to turn individual values on or off
- In contrary to radio buttons more than one value can be selected in a checkbox group.

```
<input type="checkbox" name="interests" id="music" value="music" />
<label for="music">Music</label>
<input type="checkbox" name="interests" id="art" value="art" />
<label for="art">Art</label>
<input type="checkbox" name="interests" id="tech" value="tech" />
<label for="tech">Tech</label>
<input type="checkbox" name="interests" id="cooking" value="cooking" />
<label for="cooking">Cooking</label>
```

INPUT ATTRIBUTES

- type: the input's type
- name: value's key when data is sent to server
- value: the value sent to the server
- id: to identify the input and associate a label with the input
- placeholder: a placeholder value, visible if nothing has been entered
- required: whether the input field is required to send the form data
- disabled: whether the input field should be disabled
- Min/max: Min/max value for numerical values

SELECT (DROPDOWN)

- The `select` element (sometimes called dropdown) `<select>` is used to select a value from a number of options
`<option>`
- Options can be grouped using the `<optgroup>` tag
- Attribute `multiple` allows for multiselect (Strg + Click)

```
<label for="countries">Choose your country</label>
<select id="countries">
  <optgroup label="Europe">
    <option value="austria">Austria</option>
    <option value="germany">Germany</option>
    <option value="italy">Italy</option>
  </optgroup>
  <optgroup label="USA">
    <option value="newyork">New York</option>
    <option value="california">California</option>
    <option value="texas">Texas</option>
  </optgroup>
</select>
```

DATALIST

- New form element
 - <datalist> Suggestion list for inputs

```
<input type="text" list="web"/>
<datalist id="web">
    <option value="HTML"/>
    <option value="CSS"/>
    <option value="PHP"/>
    <option value="Javascript"/>
</datalist>
```

TEXTAREA

- A <textarea> allows for multiline input
- A <textarea> is used to let the user enter some free-form text, e.g. a message in a contact form or an additional comment.
- Use the rows and cols attributes to specify a size.

```
<textarea name="test" cols="30" rows="4"></textarea>
```

FIELDSET

- A <fieldset> is used to group several form elements and labels within a form.
- A fieldset can have a nested <legend> attribute providing a caption for the fieldset.
- When a fieldset is disabled, all form controls within the fieldset are disabled.

```
<form>
  <fieldset>
    <legend>Anrede:</legend>
    <input type="radio" name="gender" value="male" />männlich
    <input type="radio" name="gender" value="female" />weiblich
  </fieldset>
</form>
```

BUTTONS

- Click-Button

```
<button type="button" name="Textbutton1" onclick="...">clickme</button>
```

- Submit-Button

- submits form-data

```
<form action="index.php" method="post" id="form1">  
  ...  
  <button type="submit" form="form1" value="Submit">Submit</button>  
</form>
```

- Reset-Button

- resets all form values to its initial values

```
<button type="reset" value="Reset">Reset</button>
```

BUTTON VS INPUT TYPE="BUTTON"

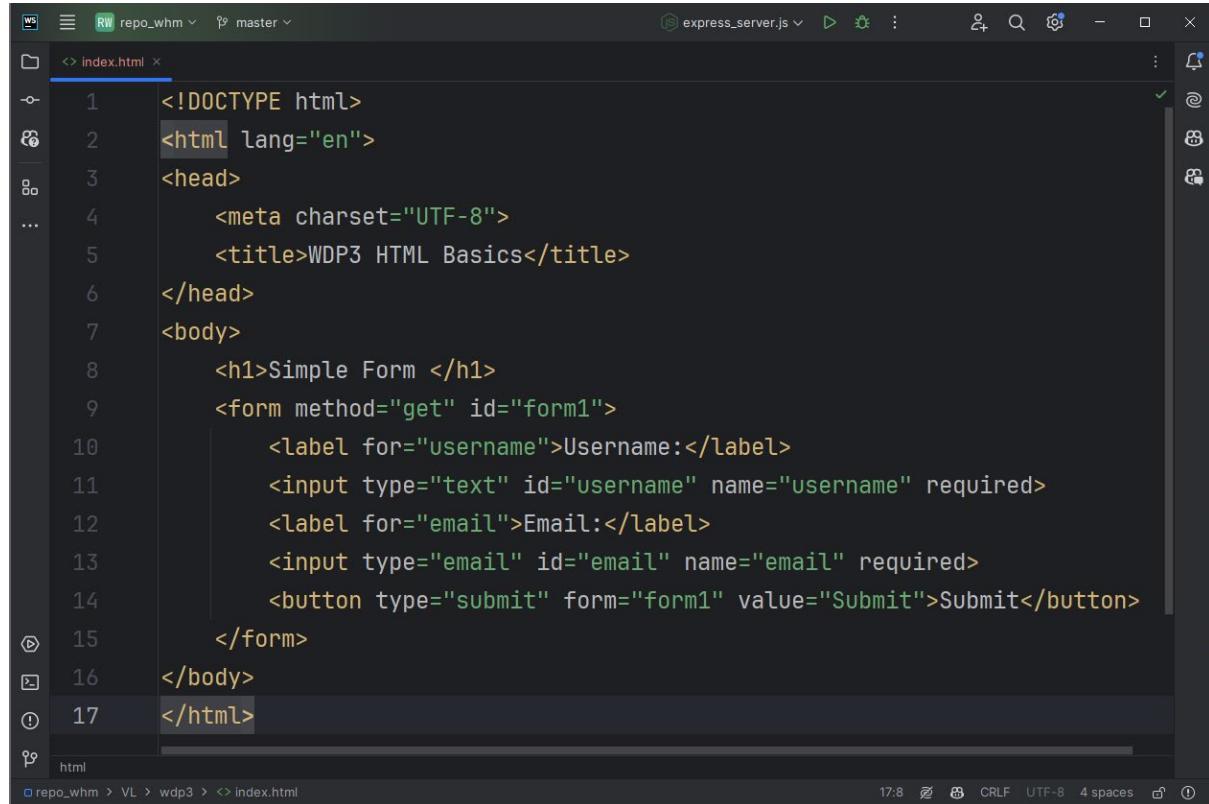
Button

- by default behaves like if it had a "type="submit" attribute
- can be used without a form as well as in forms.
- text or html content allowed
- css pseudo elements allowed (like :before)
- tag name is usually unique to a single form

Input type

- type should be set to 'submit' to behave as a submitting element
- can only be used in forms
- only text content allowed
- no css pseudo elements
- same tag name as most of the forms elements (inputs)

FORM SUBMISSION



The screenshot shows a code editor interface with two tabs open: `index.html` and `express_server.js`. The `index.html` tab is active, displaying the following HTML code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>WDP3 HTML Basics</title>
</head>
<body>
    <h1>Simple Form </h1>
    <form method="get" id="form1">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>
        <button type="submit" form="form1" value="Submit">Submit</button>
    </form>
</body>
</html>
```

The code editor has a dark theme and includes syntax highlighting for HTML. The `express_server.js` tab is visible at the top right but contains no visible code.

FORM SUBMISSION

- If no action is specified, data is sent to the current page
- Use Chrome DevTools to find out what data is sent by the form

method="GET"

X Headers Payload Preview Response Initiator Timing

▼ General

Request URL: http://localhost:63342/repo_whm/VL/wdp3/index.html?username=jschoenb&email=johannes.schoenboeck%40fh-hagenberg.at
Request Method: GET
Status Code: 200 OK
Remote Address: 127.0.0.1:63342
Referrer Policy: strict-origin-when-cross-origin

X Headers Payload Preview Response Initiator Timing

▼ Query String Parameters view source view URL-encoded

username: jschoenb
email: johannes.schoenboeck@fh-hagenberg.at

method="POST"

X Headers Payload Preview Response Initiator Timing

▼ General

Request URL: http://localhost:63342/repo_whm/VL/wdp3/index.html
Request Method: POST
Status Code: 200 OK
Remote Address: 127.0.0.1:63342
Referrer Policy: strict-origin-when-cross-origin

X Headers Payload Preview Response Initiator

▼ Form Data view source view URL-encoded

username: jschoenb
email: johannes.schoenboeck@fh-hagenberg.at

FORM VALIDATION

- **Never trust user input!**
- Client-side validation: JavaScript and built-in form validation (done by the browser)
 - Define required fields and input formats.
- Required attribute
 - `<input type="text" required />`
- Pattern attribute: validate against a regular expression
 - `<input type="text" pattern="[A-Za-z]+" />`
- Min(length) & max(length) attributes
 - `<input type="number" min="5" max="25" />`
- Custom error messages have to be set up via JavaScript.
- Server-side validation (before data is processed)

FORMS AND UX

- Always add labels to input fields
 - Placeholders are not enough!
- Provide (do not remove!) focus states
- Care about the tab order for people using the keyboard only for navigation
- Provide helpful hints and error messages.
- Read more in: [Forms on the Web That Make Me Cry and Why](#)

GRAPHICS

- <canvas>: Drawing area (drawing via JS possible)
 - https://www.w3schools.com/html/html5_canvas.asp
- <svg>: Container for SVG graphics (creation in standard SVG syntax possible)
 - https://www.w3schools.com/html/html5_svg.asp

VALIDATORS

- Check a document against a standard
- Quality assurance
 - ensures functionality/display in the most common browsers (see <https://validator.w3.org/docs/why.html>)
- Recommended validator
 - W3C: <https://validator.w3.org/>

WEB ACCESSIBILITY

- Web Accessibility Initiative (WAI)
 - Web Content Accessibility Guidelines (WCAG)
-

"THE POWER OF THE WEB IS IN ITS
UNIVERSALITY. ACCESS BY EVERYONE
REGARDLESS OF DISABILITY IS AN
ESSENTIAL ASPECT."

Tim Berners Lee

ACCESSIBILITY

- To make a product accessible means to build it in a way that people with disabilities can perceive, understand, navigate, and interact with it
- Accessible websites (and other software products) are not only optimized for blind, deaf or motor-impaired people, but for all of us!
- WHO: ~15% of all people have an impairment

WHAT IS WAI-ARIA? WHAT IS WCAG?

- WAI -> Web Accessibility Initiative
 - Develops standards and support materials to help you understand and implement accessibility
- ARIA -> Accessible Rich Internet Applications
 - A specification defining a set of additional HTML attributes that can be applied to elements to provide additional semantics and improve accessibility wherever it is lacking. ([Source](#))
- WCAG is short for Web Content Accessibility Guidelines
 - Developed with a goal of providing a single shared standard for web content accessibility that meets the needs of individuals, organizations, and governments internationally. ([Source](#))

WAI-ARIA

- A set of additional HTML attributes that can be applied to elements to provide additional semantics and improve accessibility wherever it is lacking.
 - **Roles** to define what an element is or does: search, tablist, navigation, banner ,...
 - **Properties** to provide additional meaning/semantics: aria-labelledby, aria-required, aria-live ,...
 - **States:** aria-disabled, aria-checked ,...
- The first rule of ARIA: don't use it if there are native HTML elements and attributes providing the desired behavior!
 - **No ARIA is better than bad ARIA**
- **Some examples, dos & don'ts**

4 WCAG PRINCIPLES

- **Perceivable** – Information and user interface components must be presentable to users in ways they can perceive.
- **Operable** – User interface components and navigation must be operable.
- **Understandable** – Information and the operation of the user interface must be understandable.
- **Robust** – Content must be robust enough that it can be interpreted by a wide variety of user agents, including assistive technologies.
- **3 Levels:** A, AA, AAA

ACCESSIBILITY – QUICK RULES

ACCESSIBILITY – QUICK RULES

- **Structure/content:** Use headings, titles, lists etc. as such (e.g. heading, not just large font with appropriate formatting)
- **Alt texts:** Alternative text for images (mandatory for HTML5 anyway)
- **Color:** sufficient contrast, not the sole information carrier
- **Navigation** support/accessibility via keyboard: accesskeys, focus & tabindex
- **Labels** for form elements

USE ALTERNATIVE TEXT FOR ICON BUTTONS

- Alternative text for images, videos, and especially for buttons (or links) using icons only
 - Screenreader user is able to perceive what a button is for
 - When a button only contains an icon (or for example an “x” character to indicate it's a button that closes a modal or similar), a screenreader would only read “button”, but not tell the user what the button does.
 - alt="" -> tells screenreader to ignore image

- Use the aria-label attribute:

```
<button aria-label="Beitrag erstellen Dialog schließen">  
  <svg ... />  
</button>
```



- Difference between aria-label, aria-labelledby and aria-describedby

USE MEANINGFUL LINK TEXT

- Instead of using code like this

```
<p>To read more about penguins <a  
 href="...">click here</a>.</p>
```

provide meaningful link texts to indicate what the user can expect when clicking the link:

```
<p>To learn more read this <a  
 href="...">article about penguins</a>.</p>
```

Please find a security questionnaire [here](#)

Fill out a form by the [link](#)

[This](#) is where you can create a new ticket

Always follow [this](#) guideline

Please find a [security questionnaire](#)

Fill out a [new equipment request](#)

[Create a new ticket](#) in the corporate Jira

Always follow the [UI text guideline](#)

Naughty

Better

From this article: [Designing Better Links For The Web](#)
on Smashing Magazine

USE BUTTONS AND LINKS INSTEAD OF DIVS AND SPANS

- Even worse than having a button (or link) without a label is using no `<button>` or `<a>` element at all!
 - Often `<div>` or `` elements with a JavaScript click handler are used instead
- **Potential Problems:**
 - User can't interact with the button/link using the keyboard:
 - Element can't be focused.
 - The element is not part of the tab-order.
 - When the JS code does not work, the action when clicking the button is broken as well.
 - Find more problems and solutions in these two HTMHell articles:
 - <https://www.htmhell.dev/22-the-good-ol-div-link/>
 - <https://www.htmhell.dev/20-close-buttons/>

NEVER REMOVE FOCUS STATES

- When elements on a web page are focused they usually get an outline
- Sometimes devs/designers do not like this outline and add the following CSS code:
 - `*:focus { outline: 0; }`
 - This is very bad when no other way of highlighting focused elements is added instead
 - Users navigating by keyboard will be completely lost on the page because visual feedback is missing

r, or table, form." When a person wanted to enter text (words or text) on a typewriter, there was a repetitive use of the space bar and backspace key. A horizontal bar was placed in the mecha

r, or table, form." When a person wanted to enter text (words or text) on a typewriter, there was a repetitive use of the space bar and backspace key. A horizontal bar was placed in the mecha

r, or table, form." When a person wanted to enter text (words or text) on a typewriter, there was a repetitive use of the space bar and backspace key. A horizontal bar was placed in the mecha

TAB ORDER AND VISUAL ORDER SHOULD BE THE SAME

- Some users prefer navigating through a web page using their keyboard
 - Use “tab” key (\leftarrow) to jump from one interactive element to the next
 - Only works when interactive elements (a, button, input, select,...) are provided by the developer.
- Be careful when the visual order of elements is not the same as the order in the DOM (which is the default tab order)!
 - Examples: using the order property in flexbox or grid, using flex-direction: row-reverse, or similar...
- The default tab order can be changed by using the **tabindex** attribute.

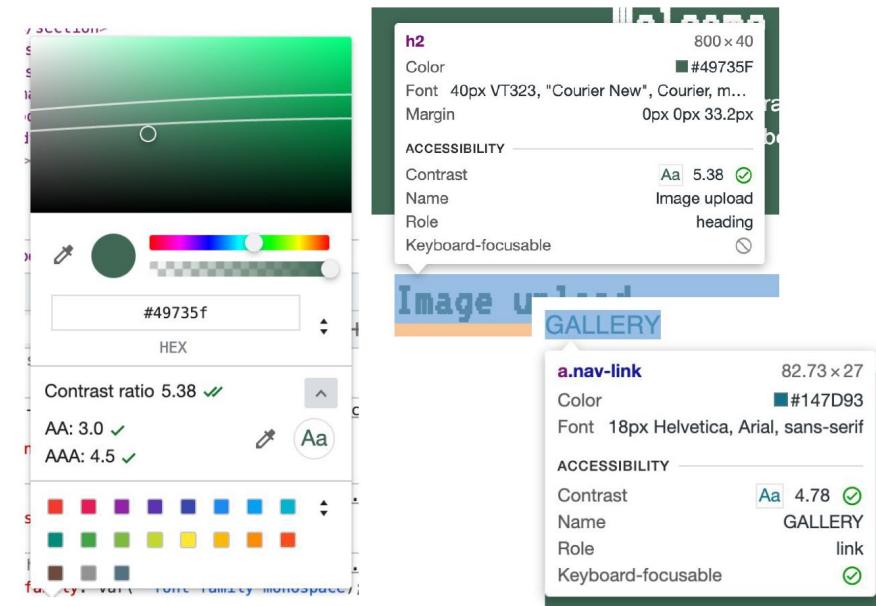
DO NOT RELY ON COLOR ALONE

- It's common to use **green** to indicate success and **red** to indicate errors
- Lot of people cannot distinguish these two colors!
- Make sure to not only rely on color alone when displaying errors or success messages

INACCESSIBLE STATUS VIA COLOR		ACCESSIBLE STATUS VIA ICONS		ERROR STATE INDICATED VIA RED		
FAILS		PASSES		Email	Email	Email required
API	●	API	✓	Default state	Extra cue required FAILS	Icon & text cue added PASSES
Dashboard	●	Dashboard	!			
Gateway	●	Gateway	✓			
● Operational ● Outage		✓ Operational ! Outage		Images from https://uxmovement.com/buttons/the-myths-of-color-contrast-accessibility/		

LET THE BROWSER HELP

- Many useful accessibility tools that can be found in Chrome's dev tools:
 - Inspect element tooltip
 - Contrast ratio checker
 - Lighthouse

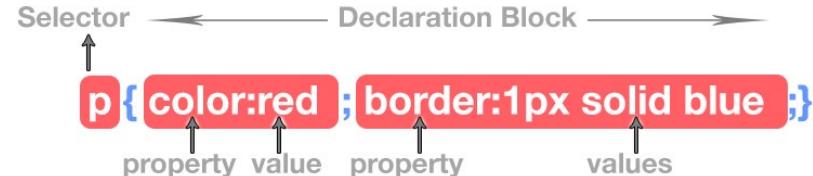
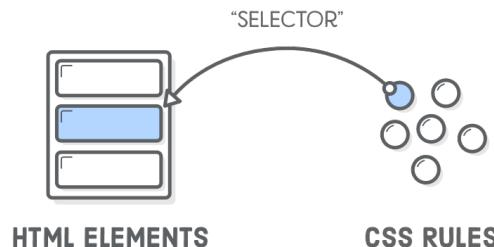


CSS - CASCADING STYLE SHEETS

Web-Design und Programmierung

WHAT IS CSS

- CSS defines how your HTML document is presented to the user, i.e., how elements should be rendered on screen
 - It's all about layout, colors, fonts, sizes, etc...
- CSS is used to separate styling from content
- CSS consists of rules that define how elements of your HTML document look like
 - A rule **selects** an element and applies a set of **properties** to it.



CSS isn't easy

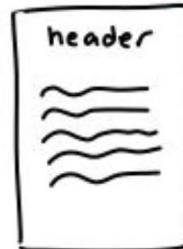
CSS seems simple at first

```
h2 {  
    font-size: 22px;  
}
```

ok this is easy!

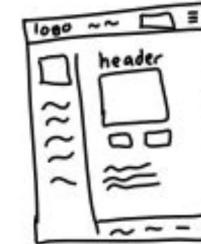


and it is easy for simple tasks



a layout like this is simple to implement!

but website layout is not an easy problem



this needs to adjust to so many screen sizes!

the spec can be surprising

TRY ME!

setting overflow: hidden; on an inline-block element changes its vertical alignment



weird!

and all browsers have bugs



I don't support flexbox for <summary> elements

ok fine

accept that writing CSS is gonna take time



if I'm patient I can fix all the edge cases in my CSS and make my site look great everywhere!

INCLUDING STYLES IN HTML

- **Inline CSS**

- <p style="color: #00a1b2"> ... </p>
- Easy but hardy maintainable
- No means to style “all” elements of a certain type
- Try to avoid inline styles!

- **Internal CSS**

- CSS code between <style> tags in the head of HTML document.
- <style> /* CSS goes here */ </style>
- High effort in case of multiple files

- **External CSS**

- Link to the CSS file in the head of your HTML document
- File ending .css
- <link rel="stylesheet" href="styles.css"/>
- Preferred variant
 - Flexible and reusable across several html files
 - Bandwidth savings

Inline CSS

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

Internal CSS

```
<head>
  <style type = text/css>
    body {background-color: blue;}
    p { color: yellow;}
  </style>
</head>
```

External CSS

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

CSS - COMMENT

```
/* This syntax should be common to you*/
```

CSS VALIDATION

- As for HTML, there are also validators for CSS
- Use is recommended
 - Easier debugging and troubleshooting
 - avoidance of non-obvious problems
- <https://jigsaw.w3.org/css-validator/>
- CSS Reference
 - <https://developer.mozilla.org/en-US/docs/Web/CSS>

SELECTORS

Give me the elements I need

- Basic Selectors
- Combined Selectors

TYPE SELECTOR

- Applies style to all elements of a certain type, i.e., **HTML tag**

```
h1 {  
    color: blue;  
    font-weight: bold;  
}
```

```
<h1>a styled heading</h1>
```

ID-SELECTOR

- Styling of an HTML Element with a specific, unique ID

```
#boldGreen {  
    color: green;  
    font-weight: bold;  
}  
  
<h1 id="boldGreen">a styled heading</h1>
```

CLASS SELECTOR

- Class-Attribute can be used to “categorize” HTML Elements
- **id** vs **class**
 - **id** attribute can be used to uniquely identify elements -> identifier can only be used once per document.
 - **class** is not a unique identifier, but rather an assignment to a group
 - It is also possible for an element to belong to several classes, e.g. class=“important current”

```
.boldGreen {  
    color: green;  
    font-weight: bold;  
}  
<h1 class="boldGreen">a styled heading</h1>  
<p class="boldGreen">a styled paragraph</p>
```

CLASS SELECTOR

- Selects elements that contain **at least** the classes stated (e.g., both classes must be present in the example)

```
.bold.green {  
    color: blue;  
    font-weight: bold;  
}
```

```
<h1 class="bold green">a styled heading</h1>  
<p class="bold">an unstyled paragraph</p>
```

UNIVERSAL SELECTOR

- Selects all elements of an HTML document
- Useful for overriding browser default

```
* {  
    padding: 0px; margin: 0px;  
}
```

BASIC SELECTORS - SUMMARY

```
<body>
  <div id="box">
    <p class="one">Erster Absatz</p>
    <p class="one two">Zweiter Absatz</p>
    <p>Dritter Absatz</p>
  </div>
</body>
```

Selector	Type	Description
*	all	Selects all elements type-independently
1 E	Type	Selects all elements with the identifier E
2 .class	Class	Selects all elements with the named CSS class
3 .class1.class2	Class	Selects elements that contain at least the classes
4 #id	ID-selector	Selects an element with the passed ID

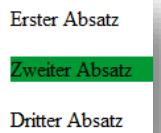
1 `p { background-color: green; }`



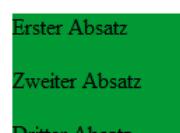
2 `.one { background-color: green; }`



3 `.one.two { background-color: green; }`



4 `#box { background-color: green; }`



GROUP AND CONTEXT SELECTORS

Selector	Description
sel1, sel2 ... sel n	Selects the combination of all nodes selected by the selectors
E F	Selects all elements F that are descendants (not only children) of an element E
E > F	Selects all elements F that are direct child nodes of an element E
E + F	Selects immediate sibling nodes of an element of type E from type F
E ~ F	Selects all the following sibling nodes of an element E that have the type F

- 1
- 2
- 3
- 4
- 5

```
<div id="box">
  <p class="one">Erster Absatz</p>
  <p class="one two">Zweiter Absatz</p>
  <div>
    <p>Dritter Absatz</p>
    <p>Vierter Absatz</p>
  </div>
  </div>
  <p id="p5">Fuenfter Absatz</p>
  <p id="p6">Sechster Absatz</p>
```

```
#box, #p5 {
  background-color: green;
}
```

1

Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz
Sechster Absatz

```
#box p {
  background-color: green;
}
```

2

Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz
Sechster Absatz

```
#box > p {
  background-color: green;
}
```

3

Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz
Sechster Absatz

```
#box + p {
  background-color: green;
}
```

4

Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz
Sechster Absatz

```
#box ~ p {
  background-color: green;
}
```

5

Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz
Sechster Absatz

ATTRIBUTE SELECTORS

```
<body>
  <div id="box">
    <p title="P1">Erster Absatz</p>
    <p name="P2">Zweiter Absatz</p>
    <div title="div2">
      <p>Dritter Absatz</p>
    </div>
  </div>
</body>
```

Selector	Description
1 [name]	Selects all elements of any type that have an attribute name
2 E[name]	Selects all elements of type E that have an attribute name
3 [name=value]	Selects all elements to which all named attribute filters apply
[name]=value	Especially for lang attributes; selects an element if the named attribute has the passed value, or the value followed by any hyphen-separated string.
[name*=value]	Selects an element if the value of the named attribute contains the passed string as substring
[name~=value]	Selects an element if the value of the named attribute contains the passed string as a partial value (separated by spaces)
[name\$=value]	Selects an element if the value of the attribute ends with the passed string
[name!=value]	Selects an element if the value of the named attribute does not correspond to the passed string or the attribute does not exist
[name^=value]	Selects an element if the value of the attribute starts with the passed string

1

```
[title] {
  background-color: green;
}
```

Erster Absatz
Zweiter Absatz
Dritter Absatz

2

```
p[title] {
  background-color: green;
}
```

Erster Absatz
Zweiter Absatz
Dritter Absatz

3

```
[title="div2"] {
  background-color: green;
}
```

Erster Absatz
Zweiter Absatz
Dritter Absatz

PSEUDO-CLASSES

- A CSS pseudo-class is a keyword added to a selector that specifies a special state of the selected element(s).
 - For example, the pseudo-class :hover can be used to select a button when a user's pointer hovers over the button and this selected button can then be styled.
- A pseudo-class consists of a colon (:) followed by the pseudo-class name (e.g., :hover).
 - A functional pseudo-class also contains a pair of parentheses to define the arguments.
 - The element that a pseudo-class is attached to is defined as an anchor element (e.g., button in case button:hover)
- See <https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-classes>

PSEUDO CLASSES

- Pseudo classes often used with links
 - **:link** -> Matches links that have not yet been visited
 - **:visited** -> Matches links that have been visited
 - **:hover** -> Matches when a user designates an item with a pointing device
 - **:active** -> Matches when an item is being activated by the user; for example, when the item is clicked on
 - **:focus** -> Matches when an element has focus

```
a:link { color: green; }
```

```
a:focus { background-color: orange; }
```

TREE-STRUCTURAL PSEUDO-CLASSES

- **:empty** -> Represents an **element with no children** other than white-space characters.
- **:nth-child** -> Uses An+B notation to select **elements** from a **list of sibling elements**.
- **:nth-last-child** -> Uses An+B notation to select **elements** from a **list of sibling elements**, counting backwards from the end of the list.
- **:first-child** -> Matches an element that is the **first** of its **siblings**.
- **:last-child** -> Matches an element that is the **last** of its **siblings**.
- **:only-child** -> Matches an element that has no siblings.
- **:nth-of-type** -> Uses An+B notation to select **elements** from a **list of sibling elements** that match a **certain type** from a list of sibling elements.
- **:nth-last-of-type** -> Uses An+B notation to select **elements** from a **list of sibling elements** that match a certain type from a list of sibling elements counting backwards from the end of the list.
- **:first-of-type** -> Matches an element that is the first of its siblings, and also matches a certain type selector.
- **:last-of-type** -> Matches an element that is the last of its siblings, and also matches a certain type selector.
- **:only-of-type** -> Matches an element that has no siblings of the chosen type selector.

TREE-STRUCTURAL PSEUDO-CLASSES

```
<body>
  <div id="box">
    <p>Erster Absatz</p>
    <p>Zweiter Absatz</p>
    <p>Dritter Absatz</p>
    <p>Vierter Absatz</p>
    <p>Fuenfter Absatz</p>
  </div>
  <div id="box2">
    <p>Erster Absatz</p>
    <p>Zweiter Absatz</p>
    <p>Dritter Absatz</p>
    <p>Vierter Absatz</p>
    <p>Fuenfter Absatz</p>
  </div>
  <div id="box3">
    <p>Erster Absatz</p>
  </div>
</body>
```

```
p:first-child {
  background-color: green;
}
```

Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz

Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz

Erster Absatz

```
p:only-child {
  background-color: green;
}
```

Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz

Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz

Erster Absatz

TREE-STRUCTURAL PSEUDO-CLASSES

```
<body>
  <div id="box">
    <p>Erster Absatz</p>
    <p>Zweiter Absatz</p>
    <p>Dritter Absatz</p>
    <p>Vierter Absatz</p>
    <p>Fuenfter Absatz</p>
  </div>
  <div id="box2">
    <p>Erster Absatz</p>
    <p>Zweiter Absatz</p>
    <p>Dritter Absatz</p>
    <p>Vierter Absatz</p>
    <p>Fuenfter Absatz</p>
  </div>
  <div id="box3">
    <p>Erster Absatz</p>
  </div>
</body>
```

Every 2nd child element, if of type p, starting from the 3rd element

```
p:nth-child(2n+3) {
  background-color: green;
}
```

Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz
Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz
Erster Absatz

First child has index 1 and is therefore not selected

```
p:nth-child(even) {
  background-color: green;
}
```

Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz
Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz
Erster Absatz

Every 4th child element if of type p, starting from behind

```
p:nth-last-child(4n) {
  background-color: green;
}
```

Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz
Erster Absatz
Zweiter Absatz
Dritter Absatz
Vierter Absatz
Fuenfter Absatz
Erster Absatz

TREE-STRUCTURAL PSEUDO-CLASSES

```
<div id="box">
  <p>Erster Absatz</p>
  <p>Zweiter Absatz</p>
  <p>Dritter Absatz</p>
  <p>Vierter Absatz</p>
  <p>Fuenfter Absatz</p>
</div>
<div id="box2">
  <p>Erster Absatz</p>
  <p>Zweiter Absatz</p>
  <h2>Überschrift</h2>
  <p>Dritter Absatz</p>
  <p>Vierter Absatz</p>
  <p>Fuenfter Absatz</p>
</div>
<div id="box3">
  <p>Erster Absatz</p>
</div>
```

Every 2nd p child element

```
p:nth-of-type(2n) {
  background-color: green;
}
```

Erster Absatz

Zweiter Absatz

Dritter Absatz

Vierter Absatz

Fuenfter Absatz

Erster Absatz

Zweiter Absatz

Überschrift

Dritter Absatz

Vierter Absatz

Fuenfter Absatz

Erster Absatz

Every 2nd child element if of type p

```
p:nth-child(2n) {
  background-color: green;
}
```

Erster Absatz

Zweiter Absatz

Dritter Absatz

Vierter Absatz

Fuenfter Absatz

Erster Absatz

Zweiter Absatz

Überschrift

Dritter Absatz

Vierter Absatz

Fuenfter Absatz

Erster Absatz

CSS PROPERTIES

The basic stuff

- Font formatting
 - Alignment
 - Background
 - List
-

FONT FORMATTING

- Deals with
 - Font size, Font type, ...
- Typical properties
 - font-family: Family1, ..., Family_n;
 - font-style: italic/normal/oblique
 - font-variant: normal/small-caps
 - font-size: Value (Value+Unit, xx-small - xx-large)
 - font-weight: lighter/normal:bold/bolder/...
 - font-stretch: normal/narrower/condensed/wider/...
 - word-spacing: Value
 - letter-spacing: Value
 - text-decoration: underline/overline/line-through/blink/none
 - text-transform: capitalize/lowercase/uppercase/normal
 - color: Color-value
- <https://wiki.selfhtml.org/wiki/CSS/Wertetypen>

ALIGNMENT

- For texts
 - Alignment: e.g. left- or right-aligned
 - In tables often also vertical alignment
 - Indentation
 - Line height
 - text-indent: text indentation
 - Value (positive moves to right, negative to left)
 - Normal only first line of a paragraph
 - line-height: Line height (Value)
 - vertical-align: Vertical alignment
 - Baseline, top, middle, bottom,...
 - See <https://developer.mozilla.org/en-US/docs/Web/CSS/vertical-align>
 - text-align: Horizontal alignment
 - start, end, center, justify, justify-all
 - white-space: Text wrap

BACKGROUND-COLOR AND -IMAGES

- **background-color:** Color value
- **background-image:** Url to image
- **background-repeat:**
 - background image can be repeated along the horizontal and vertical axes
 - Values: repeat-x (horizontally only), repeat-y (vertically only), repeat (covers the whole background), space (repeated, but not clipped), round (repeated images will stretch, leaving no gaps, until there is room for another one to be added), no-repeat
- **background-attachment:**
 - Values:
 - scroll: background is fixed relative to the element itself and does not scroll with its contents
 - fixed: background is fixed relative to the viewport; even if an element has a scrolling mechanism, the background doesn't move with the element
 - local: background is fixed relative to the element's contents; if the element has a scrolling mechanism, the background scrolls with the element's contents
- **background-position:** top, bottom, left, right, center, 2 values (x and y)
- **background:** Shorthand (Combination of the above values)

LIST

- For bullet list () and enumerations ()
- **list-style-type:** Element shown
 - see <https://developer.mozilla.org/en-US/docs/Web/CSS/list-style-type> for possible values
- **list-style-position:**
 - position of the marker relative to a list item
 - Inside | outside (default)
- **list-style-image:** URL to own graphics used as marker
- **list-style:** shorthand property

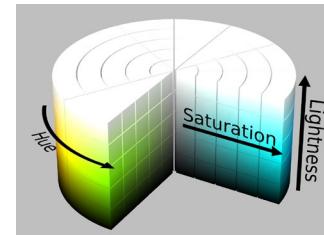
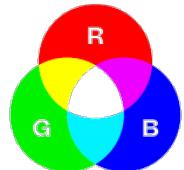
COLORS, FONTS AND SIZES

Make it nicer!



COLOR VALUES

- Color keywords
 - white, black, red, blue, darkslateblue, darkorange, firebrick,...
- Hexadecimal RGB values (#RRGGBB)
 - #FFFFFF, #000000, #FF0000, #0000FF, #483D8B, #FF8C00, #B22222,...
- RGB(A) values or HSL(A) values (hue, saturation, lightness)
 - `rgb(255, 255, 255)`,
 - `rgba(72, 61, 139, 0.5)` -> alpha channel: 0-1 (transparent - opaque)
 - `hsl(0, 100%, 50%)` -> hue between 0 - 360



UNITS

- **Absolute Values**
 - px: pixel (absolute value)
- **Relative Values**
 - Depending on the size of e.g., parent element or font size
 - %: relative to parent's size
 - em: the current element's font size
 - (when used as unit for font-size, em refers to the parent-element's font size)
 - rem: relative to the root font size
 - (font-size of html element; default is 16px)
 - vw/vh: 1% of the viewport width/height
- More about CSS units: [Values and units on MDN](#)

WEB FONTS

- The font-family property takes one or more font names (separated by a comma)
- Browser uses the first available font from the list (user might not have all fonts installed)
 - `font-family: Helvetica, Arial, sans-serif;`
- Include web fonts in your project if you don't want to be limited to the fonts installed on your user's computer
 - Where to find web fonts: Google Fonts, Font Squirrel, Fontspring,...
 - Web Open Font Format (.woff) is supported by all modern browsers (<https://caniuse.com/?search=woff>)

WEB FONTS

- @font-face
- Declare a font family with a specific name
- Specify the source file
- Do not use links to (Google) fonts
 - Privacy/legal issues

```
@font-face{  
    font-family: myFont;  
    src: url(font.woff);  
}  
  
div{  
    font-family: myFont, ..., serif/sans-serif;  
}
```

INHERITANCE, SPECIFICITY AND CASCADING

Now it's getting weird!

- Inheritance
 - Specificity
 - Cascading
-

REUSING STYLES

```
<section>
  <h1>Headline</h1>
  <p>Some text</p>
</section>
```

```
h1 {
  color: #454545;
  font-family: Arial;
}

p {
  color: #454545;
  font-family: Arial;
}
```

Group selectors, separate them by comma

```
h1, p {
  color: #454545;
  font-family: Arial;
}
```

Elements **h1** and **p** inherit styles from their parent **section**

```
section {
  color: #454545;
  font-family: Arial;
}
```

CSS INHERITANCE

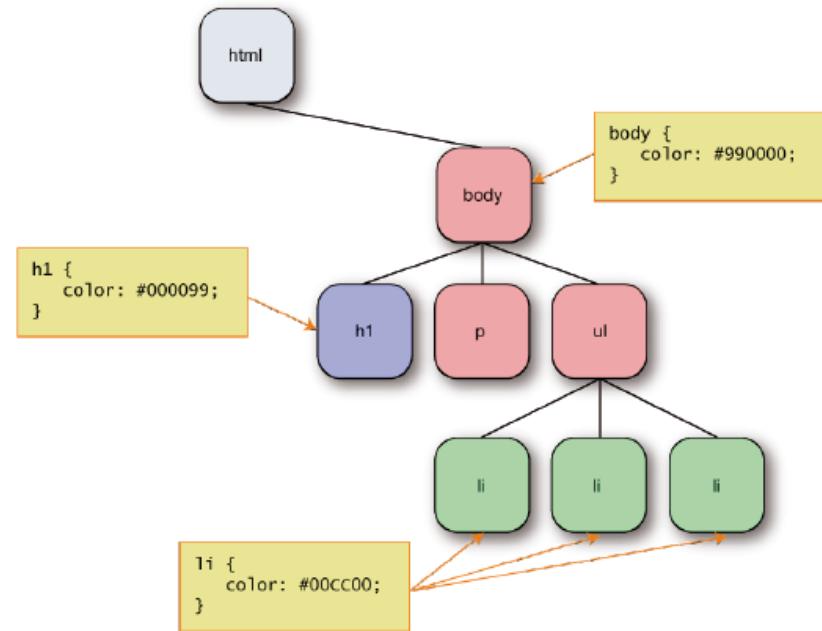
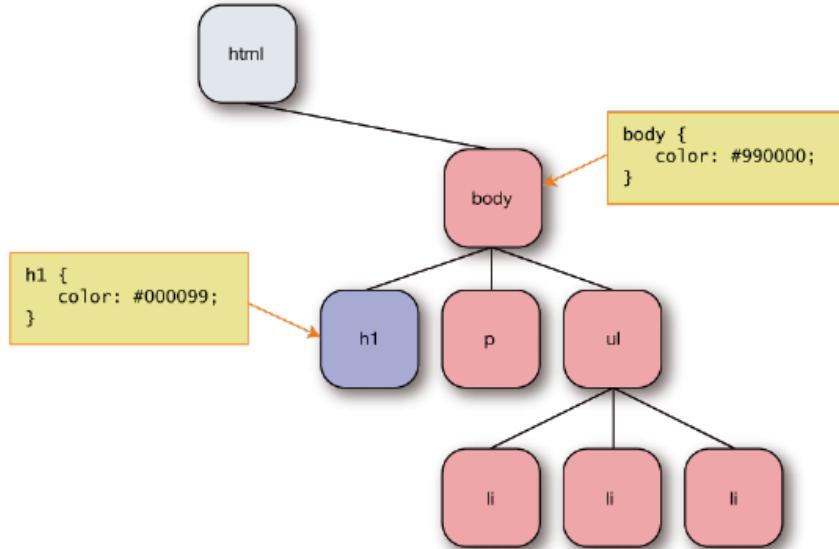
- There are two types of CSS properties:
 - inherited properties - are set to the **computed value of the parent element** - e.g. “color”
 - List of properties see <https://web.dev/learn/css/inheritance>
 - non-inherited properties - are set to the **initial value of the property** - e.g. “border”
- In the example below, the color and font-family values are inherited by the h1 and p elements from their section parent.

```
<section>
  <h1>Headline</h1>
  <p>Some text</p>
</section>
```

```
section {
  color: #454545;
  font-family: Arial;
}
```

CSS INHERITANCE

- Styles are inherited as long as specific elements do not redefine them



CSS SPECIFITY

- Sometimes it is hard to know, which CSS rule is applied
- We can determine how specific a (combined) selector is by adding the weight of the individual selectors.
- Order of specificity (from general to specific)
 - Universal selector
 - Type selector
 - Class selector
 - Attribute selector
 - Pseudoclasses
 - ID-Selector
 - Inline style

CSS SPECIFICITY

- Specificity is the algorithm used by browsers to determine which rule from competing CSS declarations gets applied to an element.
- specificity algorithm is basically a three-column value of three categories or weights
- ID column
 - For each ID in a matching selector, add 1-0-0 to the weight value.
- CLASS column
 - Includes class selectors, attribute selectors, and pseudo-classes. For each of them add 0-1-0 to the weight value.
- TYPE column
 - Includes type selectors, and pseudo-elements, and all other selectors with double-colon notation. For each of them, add 0-0-1 to the weight value

```
#introduction{ color: red; } <p class="message" id="introduction">  
.message{ color: green; }           Beispieltext  
p{ color: blue; }                  </p>
```



Beispieltext

CSS SPECIFICITY

- We can calculate specificity by using a pattern similar to semantic versioning:
 - We use the following pattern: id.class.type
 - For example: 2.3.5 > 1.4.8
 - only the first unequal number matters
- Always try to avoid overly specific selectors!

```
section.contact form [type="text"] {...} ->  
0.2.2
```

```
nav#primary a.active {...} ->  
1.1.2
```



Specific rule

CSS SPECIFICITY

- The more specific your selector is, the more likely the rules get applied to the selected element.
- But sometimes other style declarations are really !important...
 - Important “overrides” specificity
- Try to avoid the important keyword!

```
header ul#navigation a {  
    background-color: #b4e5f9;  
    text-decoration: underline;  
}  
header a {  
    text-decoration: none !important;  
}
```

CSS SPECIFICITY

```
<p>This is my first paragraph!</p>
```

```
<p class="meta">  
This is another paragraph.  
</p>
```

```
<p class="meta" id="introduction">  
This is the page introduction.  
</p>
```

```
<p style="color: black;">  
This is my last paragraph.  
</p>
```

```
.meta {  
background-color: gray;  
color: lime;  
}  
  
p {  
color: white;  
background-color: rebeccapurple;  
}  
  
#introduction {  
background-color: red;  
}
```

This is my first paragraph!

This is another paragraph.

This is the page introduction.

This is my last paragraph.

WHERE IS THE CSS CODE COMING FROM?

- User agent stylesheets
 - styles defined by the browser
- Author stylesheets
 - what we developers build
- User stylesheets
 - configured in the browser by the user of your website

The screenshot shows the browser's developer tools with the "Elements" tab selected. The left pane displays the HTML structure of the page, including sections like <head>, <body>, <main>, <section>, , and . The right pane shows the "Styles" panel, which lists CSS rules applied to the selected element. Three specific rules are highlighted with red boxes:

- A rule for elements with the selector "li {". It includes properties like "display: list-item;" and "text-align: -webkit-match-parent;".
- A rule for elements with the selector "ul {". It includes "list-style-type: disc;".
- A rule for the Pseudo-class with the selector "li::marker {". It includes properties like "unicode-bidi: isolate;" and "font-variant-numeric: tabular-nums;".

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <nav>...</nav>
    <header>...</header>
    <main>
      <section>
        <h2>The goal</h2>
        <p>...</p>
        <h3>Project steps and ToDos</h3>
        <ul>
          <li> == $0
            &:marker
              "HTML setup: gallery page (homepage), about, contact page"
          </li>
          <li>...</li>
          <li>...</li>
          <li>...</li>
          <li>...</li>
          <li>...</li>
          <li>...</li>
          <li>...</li>
          <li>...</li>
        </ul>
      </section>
      <section>...</section>
      <section>...</section>
    </main>
    <footer>...</footer>
  </body>
</html>
```

CASCADING STYLE SHEETS

huge CSS files with
thousands of rules...

different origins of
style sheets...

selectors occur more
often than once...



some styles are more
specific than others...

source order...

inheritance...

different types of
selectors...

CASCADING STYLE SHEETS

- If there's more than one property value that can be applied to an element... which one "wins"?
- Different CSS sources: browser styles, user style, external styles, page specific styles, inline styles
- CSS specificity: elements, classes, IDs, inline styles
- CSS importance: !important
- CSS inheritance
- CSS transitions and animations
- And: source order matters! Rules that come later override earlier rules.

CASCADING STYLE SHEETS - LAYER

- The **@layer** CSS rule declares a cascade layer and can also be used to define the order of precedence in case of multiple cascade layers
 - First layer is less specific

```
<p class="alert">Beware of the zombies</p>
```

Beware of the zombies

```
@layer module, state;  
  
@layer state {  
    .alert {  
        background-color: red;  
    }  
    p {  
        border: medium solid blue;  
    }  
}  
  
@layer module {  
    .alert {  
        border: medium solid violet;  
        background-color: yellow;  
        color: white;  
    }  
}
```

CASCADING STYLE SHEETS

1. Relevance: get all the rules from all available sources that apply to a given element
2. Origin and importance: sort the rules by origin and importance
3. Specificity: when the origin is the same, rules are sorted by specificity
4. Order of appearance: when the specificity is the same, rules that can be found later in the code override earlier ones.

See also

https://developer.mozilla.org/en-US/docs/Web/CSS/Cascade#cascading_order



ORGANISATION OF CSS RULES

- Define some base styles for your whole website or application using type selectors.
 - Set a font color, the font-size of text and headlines, list styles, link styles...
- Prefer class selectors over ID selectors. Why?
 - They are less specific than ID selectors or inline style, thus can be overridden easily when needed.
 - They can be combined: an element can have more than one class (but only one ID).
 - They can be reused.
- Sort rules from less to more specific.
- Try avoiding ID selectors, inline styles and !important.*
- Make use of cascade layers when overriding third-party styles.

CSS BOX MODEL

Think inside the box!



BOX MODEL

- On a website, every element is a “box”
- Block vs. inline boxes
 - Type refers to how the box behaves in terms of page flow and in relation to other boxes on the page.
 - Boxes have an **inner display type** and an **outer display type**.
- **block boxes**
 - <div>, <p>, <h1>, <article>,...
- **inline boxes**
 - , <a>, , <input>,...

Sicher | <https://www.fh-ooe.at/campus-hagenberg/studiengaenge/bachelor/soft...> ⋮

Studium für Software-Entwicklung

Vertiefungen: Web Engineering und Business Software

Software ist der „Geist in der Maschine“. Sie steckt zum Beispiel in der Kaffeemaschine, im Auto und natürlich in jedem Rechner: im Smartphone ebenso wie im Supercomputer.

Bei der Entwicklung von innovativer Qualitätsoftware ist kreatives Problemlösen durch Einsatz modernster Methoden und Werkzeuge gefragt. Somit geht das Studium Software Engineering über das reine Programmieren weit hinaus. Es umfasst den gesamten Software-Entwicklungszyklus: von der Problemanalyse über Design, Implementierung und Test bis zur Wartung und Weiterentwicklung.

Kurzprofil

Akademischer Abschluss:
Bachelor of Science in Engineering (BSc)

Studiendauer:
6 Semester (180 ECTS)

Studienplätze:
insgesamt 75 Plätze für das Vollzeit- und das berufsbegleitende Studium

Organisationsform:
Vollzeit oder berufsbegleitend

Zugangsvoraussetzungen:
Hochschulreife (Matura, Reifeprüfung, Berufsreifeprüfung, Abitur), einschlägige Studienberechtigungsprüfung oder **FH-Studienbefähigungslehrgang**

Bewerbung:
online oder schriftlich bis spätestens 30.06.
www.fh-ooe.at/bewerbung

Aufnahmeverfahren:
Bewerbungsgespräch

Anerkennung nachgewiesener Kenntnisse:
individuell für Lehrveranstaltungen möglich

Praktikum:
im 6. Semester im In- oder Ausland

Studioplan:
Lehrinhalte des Vollzeit-Studiums im Überblick
Lehrinhalte des berufsbegleitenden Studiums im Überblick

BOX MODEL – OUTER DISPLAY TYPE BLOCK

- If a box has an outer display type of **block**, then:
 - The box will break onto a **new line**.
 - The width and height properties are respected.
 - Padding, margin and border will cause other elements to be pushed away from the box.
 - If width is not specified, the box will extend in the inline direction to fill the space available in its container.
 - In most cases, the box will become as wide as its container, filling up 100% of the space available.
- Some HTML elements, such as `<h1>` and `<p>`, use block as their outer display type by default.

BOX MODEL – OUTER DISPLAY TYPE INLINE

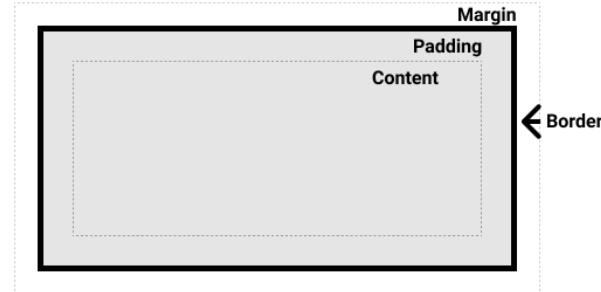
- If a box has an outer display type of **inline**, then:
 - The box will not break onto a new line.
 - The width and height properties will not apply.
 - Top and bottom padding, margins, and borders will apply but will not cause other inline boxes to move away from the box.
 - Left and right padding, margins, and borders will apply and will cause other inline boxes to move away from the box.
- Some HTML elements, such as `<a>`, ``, `` and `` use **inline** as their outer display type by default.

BOX MODEL – INNER DISPLAY TYPE

- Boxes also have an **inner display type**, which dictates how elements inside that box are laid out.
- Block and inline layout is the default way things behave on the web.
 - By default and without any other instruction, the elements inside a box are also laid out in normal flow and behave as block or inline boxes.
- **display** property sets an element's inner and outer display types (more on that when dealing with layouts)

BOX MODEL

- **Content box:** The area where your content is displayed;
 - size it using properties like inline-size and block-size or width and height.
- **Padding box:** The padding sits around the content as white space;
 - size it using padding and related properties.
- **Border box:** The border box wraps the content and any padding;
 - size it using border and related properties.
- **Margin box:** The margin is the outermost layer, wrapping the content, padding, and border as whitespace between this box and other elements;
 - size it using margin and related properties.



BOX MODEL - EXAMPLES

- Use measurements for padding and margin
 - ```
.box { margin: 12px; }
```
  - ```
.box {  
    padding-top: 32px;  
    padding-bottom: 12px;  
}
```
- Define a border width, style and color
 - ```
.box { border: 2px dashed violet; }
```

# BOX MODEL – SHORTHAND PROPERTIES

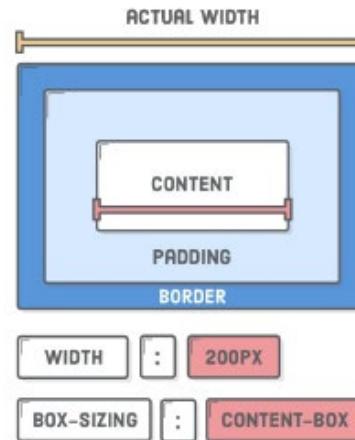
- Let you set values of multiple CSS properties simultaneously.
- Can be used for: border, margin, padding, background, font, ...

```
border-width: 2px;
border-style: solid;
border-color: red;
border: 2px solid red;
```

```
margin-top: 5px;
margin-bottom: 15px;
margin-right: 10px;
margin-left: 10px;
margin: 5px 10px 15px 10px;
```

# BOX SIZING

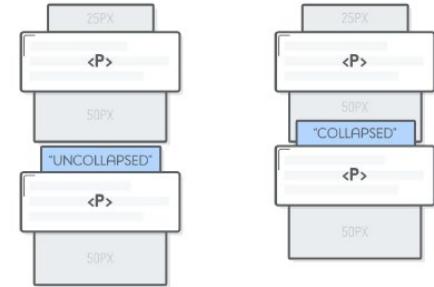
- Width and height define dimensions of a box's content.
  - Paddings and borders are additionally added.
- To define width and height of an object including paddings and borders, set the **box-sizing** property from content-box (default) to border-box.



# VERTICAL COLLAPSING OF MARGINS

- Check the following example!
  - Instead of sitting 48px apart, their 24px margins merge together, occupying the same space!
- Only **vertical** margins collapse (not horizontal)
- Only adjacent siblings collapse
  - E.g., if there would be a `<br/>` between the p's above, no collapsing would appear
- Bigger margin wins!
- For more details check <https://www.joshwcomeau.com/css/rules-of-margin-collapse/>

```
<style>
 p {
 margin-top: 24px;
 margin-bottom: 24px;
 }
</style>
<p>Paragraph One</p>
<p>Paragraph Two</p>
```



# LAYOUTS IN CSS



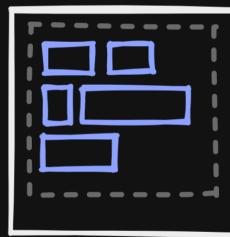
# LAYOUT ALGORITHMS IN CSS

- Flow Layout
- Table Layout
- Float Layout
- Positioned Layout
- Flexible box Layout  
(Flexbox)
- Grid Layout

## CSS Layout Modes

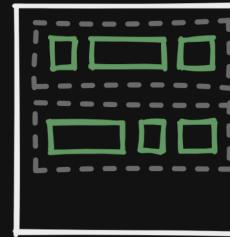
---

flow



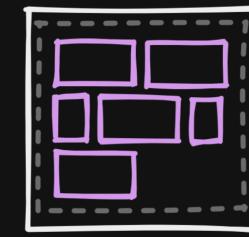
this is the default.  
elements flow like text

flexbox



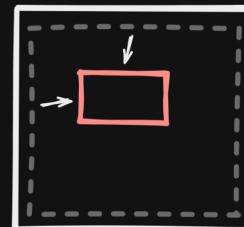
one-dimensional layout  
system

grid



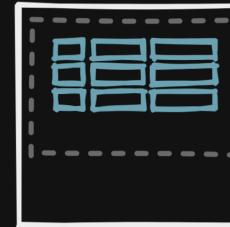
two-dimensional layout  
system

positioned



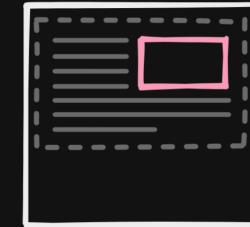
extract from normal flow  
to a precise position

table



for structured tabular  
data

float



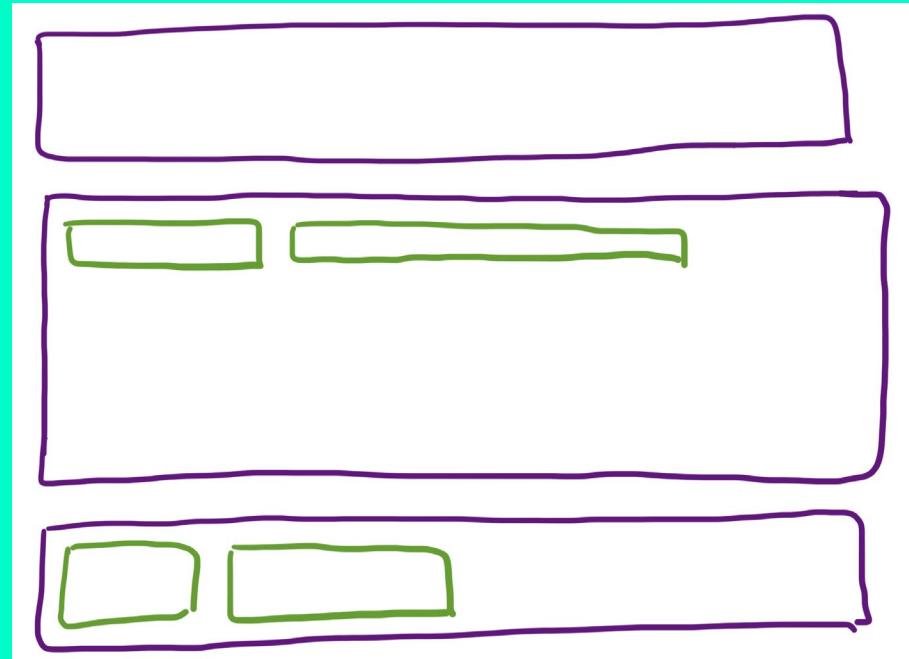
for floating images inside  
of text. occasionally used  
for layouts by masochists

# DISPLAY PROPERTY

- Sets whether an element is treated as block or inline element and the layout used for its child elements.
  - display-outside values: block, inline, inline-block, none (to hide elements)
  - display-inside values: flow, flex, grid

```
.element {
 display: block | inline | inline-block | none | flex | grid;
}
```

# FLOW LAYOUT



# FLOW LAYOUT (DEFAULT)

- **Block-level elements**

- Always create a new line
- Some default margin (top/bottom)
- Uses all of the available space (left/right) dependent on the parent element
- Examples: <div>, <h\*>, <p>, <ol>, <ul>, <table>

- **Inline elements**

- Do not start a new line (also linebreaks may happen)
- No default margin
- Use only the space that is needed to show the content
- Examples: <a>, <br> (!), <img>, <input>, <select>, <span>, <textarea>

```
<body>
 <div>Hallo</div>schöne
 neue<div>Welt</div>
</body>
```

Hallo  
schöne neue  
Welt

# inline vs block

HTML elements default to **inline** or **block**

example block elements

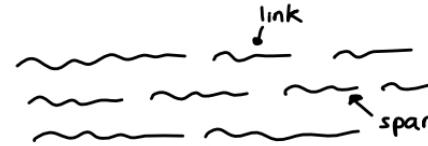
```
<p> <div>

<h1> - <h6>
<table> <form>
<article> <nav>
```

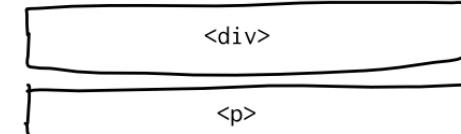
example inline elements

```
<a>
 <i>
<button> <input>
<small> <abbr>
<textarea>
```

inline elements are laid out horizontally



block elements are laid out vertically by default



inline elements ignore width & height

Setting the width is impossible, but you can use line-height to change the height

also, inline elements ignore the vertical padding of other inline elements

display can force an element to be inline or block

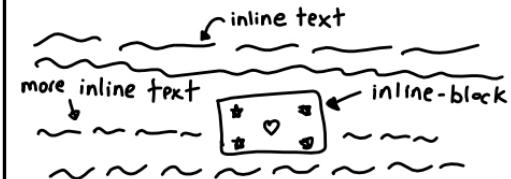
display determines 2 things:

① whether the element itself is inline, block, inline-block, etc

② how child elements are laid out (grid, flex, table, default, etc)

display: inline-block;

inline-block makes a block element that's laid out horizontally like an inline element



# FLOW LAYOUT

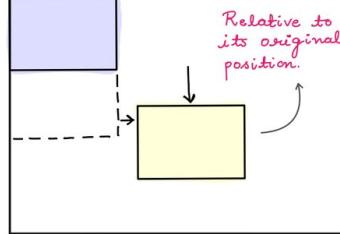
- In flow layout you can define the width of (inline-)block elements...
  - ...using an absolute or relative unit (px, %,...)
  - ...using keywords (min-content, max-content,...)
- Additionally, there are min-width/max-width properties available.
- You can do similar things with height, but be careful when setting fixed height or max-height values.
- Usually we want block elements to contain all its content and not limit the height.
- More in: [Exploring the complexities of width and height in CSS](#)

# POSITIONED LAYOUT

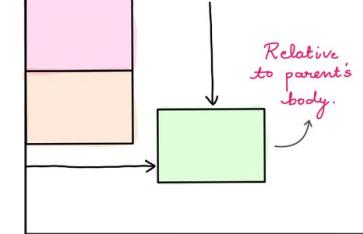
## CSS POSITIONING

@Prathikum

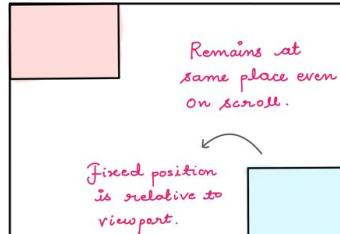
### Relative



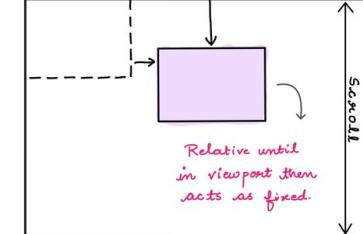
### Absolute



### Fixed



### Sticky

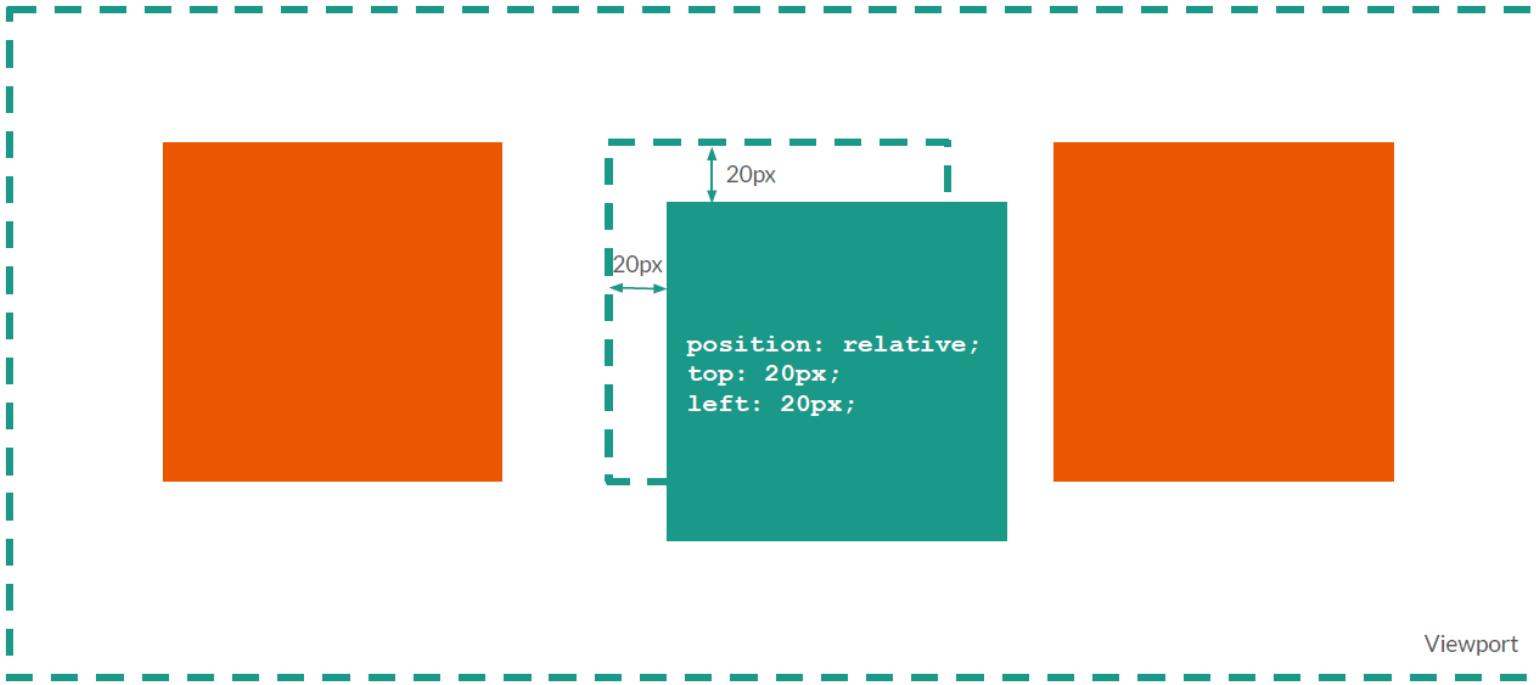


# POSITIONED LAYOUT

- By default the position of all elements on a page is static.
  - ... every element is part of the default flow layout.
  - One element after the other; next to each other or below each other.
  - Every element has its own place and pushes the other elements away when it grows.
- When we set “position” to another value than “static” we switch to positioned layout.
  - The available position values are: relative, absolute, fixed, and sticky.
- When we use positioned layout we can use CSS properties to define the element’s position:
  - The properties are: top, bottom, right, and left.
- In positioned layout, elements can overlap.

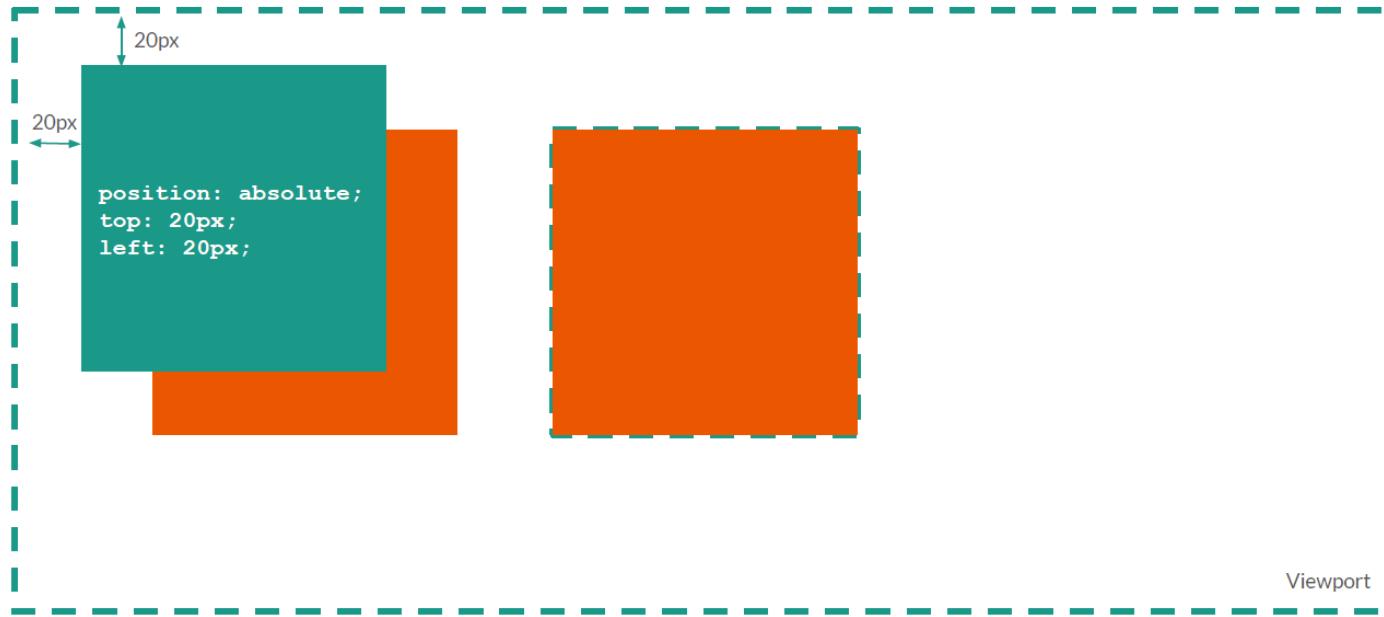
# POSITION RELATIVE

- Relative to it's original position



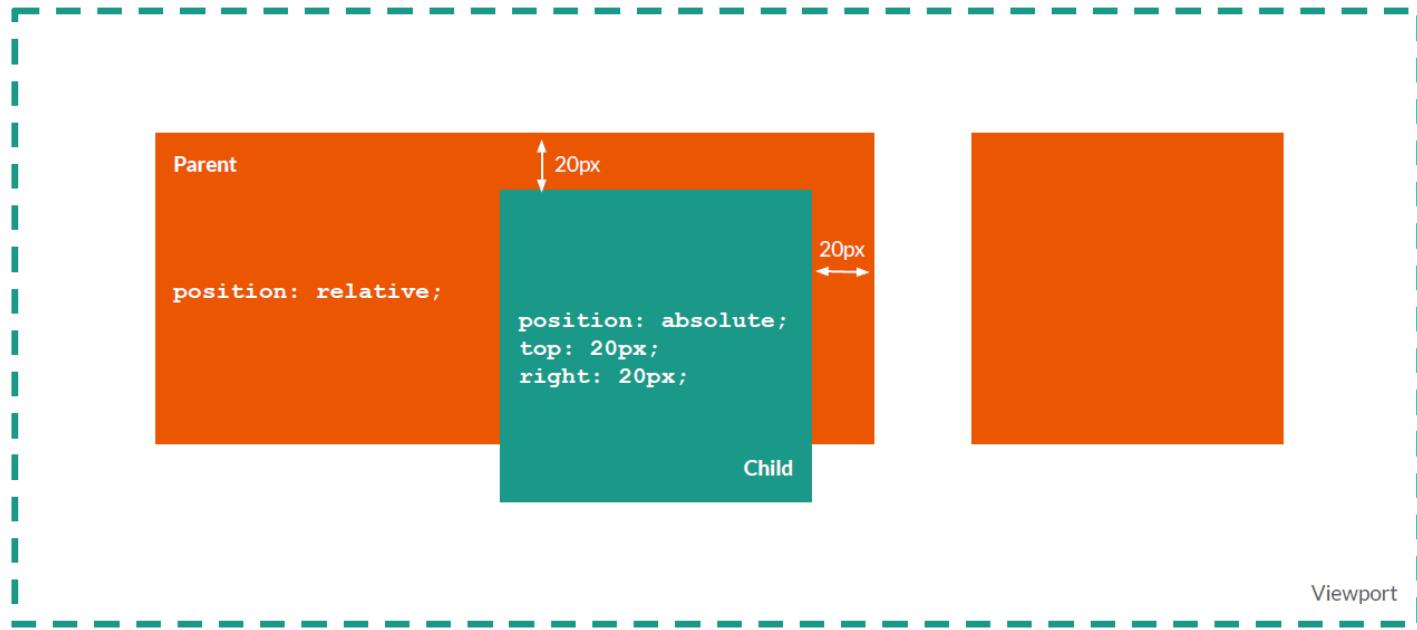
# POSITION ABSOLUTE

- If parent has position attribute static absolute refers to body element



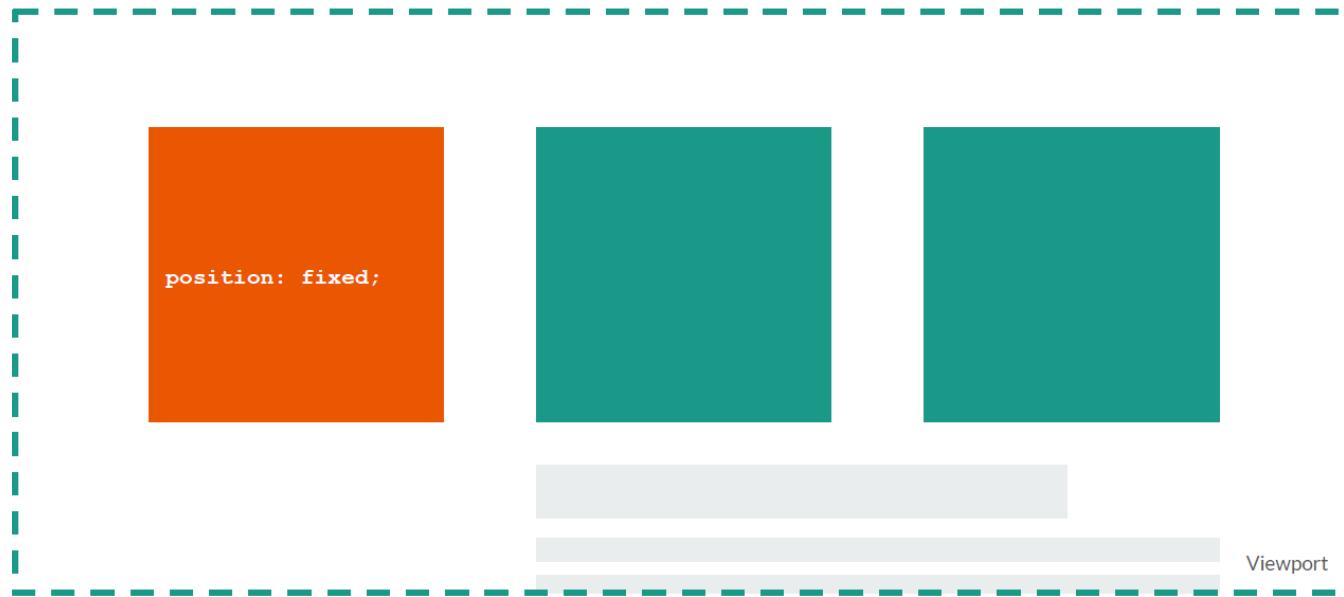
# POSITION ABSOLUTE

- If parent has position attribute relative absolute refers to the parent element



# POSITION FIXED

- Fixed position relative to viewport
- Remains on some position even on scroll

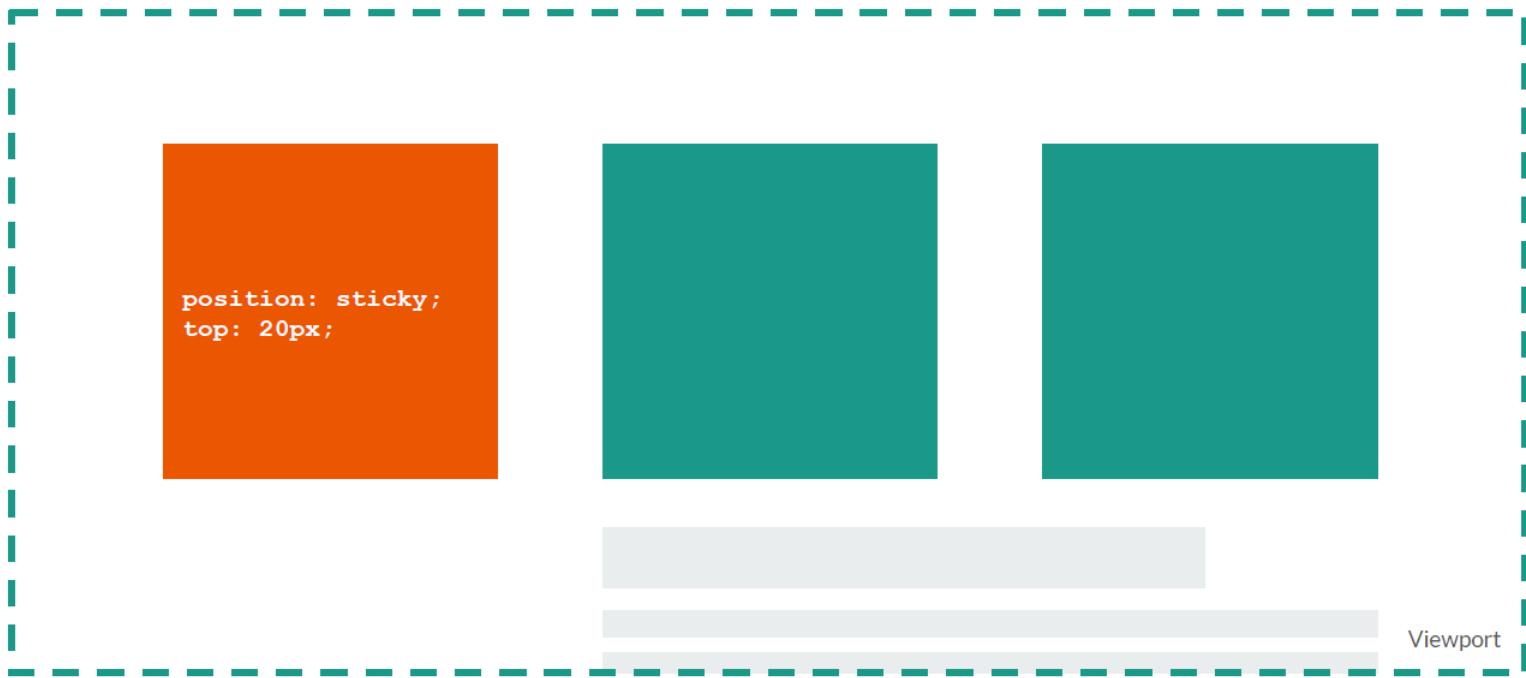


# POSITION FIXED



# STICKY POSITION

- Relative until it is in viewport than it acts as absolute

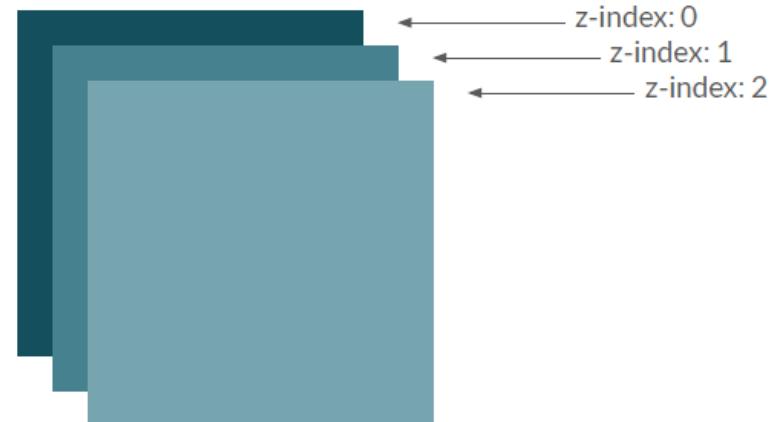


# STICKY POSITION



# z-INDEX

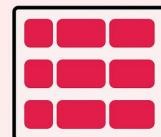
- When elements overlap, the browser needs to decide which element to render on top of the other.
  - Usually, the element that comes later in the DOM is painted on top.
- This behavior can be changed by using the z-index property.
- Works with positioned elements as well as flex and grid items (but not in flow layout).



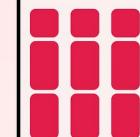
# FLEXBOX

## CSS Flexbox

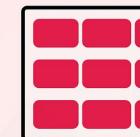
### flex-direction



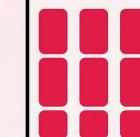
row



column

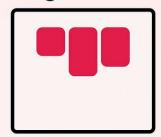


row-reverse

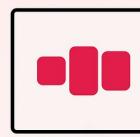


column-reverse

### align-items



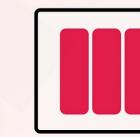
flex-start



center

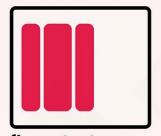


flex-end

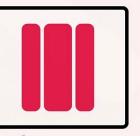


stretch

### justify-content



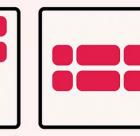
flex-start



center



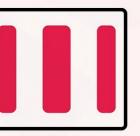
flex-end



center



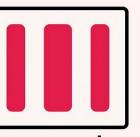
flex-end



space-between



space-around

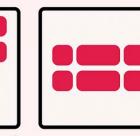


space-evenly

### align-content



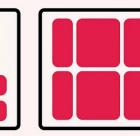
flex-start



center



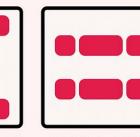
flex-end



stretch



space-between



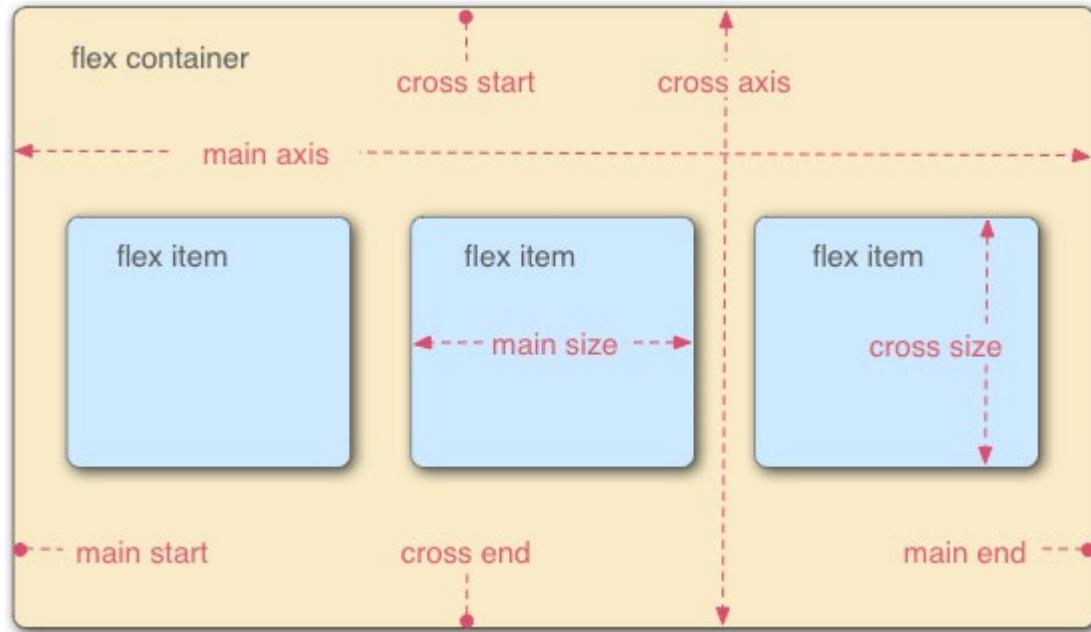
space-around

# FLEXBOX BASICS

- A layout method to arrange elements in one dimension (horizontal or vertical)
- A layout method to perfectly fill the available space in a row.
- Flexbox enables a simple, flexible layout that adapts to the conditions (size of the browser window).
- Flexbox eases the creation of responsive layouts.
- The use of Flexbox also largely eliminates the need for (absolute or relative) size specifications.
- Flex container & flex items, the (direct) children of the container
  - The flex container defines how the flex items are positioned.
  - Flex items can „ignore“ the rules set by the parent and can be manipulated individually.
- Flex items can again become flex containers.
- Guides
  - <https://css-tricks.com/snippets/css/a-guide-to-flexbox/>
  - [https://tympanus.net/codrops/css\\_reference/flexbox/](https://tympanus.net/codrops/css_reference/flexbox/)

# FLEXBOX BASICS

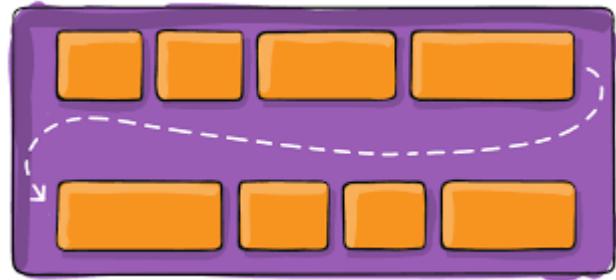
```
.container {
 display: flex;
}
```



- **flex-direction** defines main axis
  - **row** (default): left to right in ltr; right to left in rtl
  - **row-reverse**: right to left in ltr; left to right in rtl
  - **column**: same as row but top to bottom
  - **column-reverse**: same as row-reverse but bottom to top

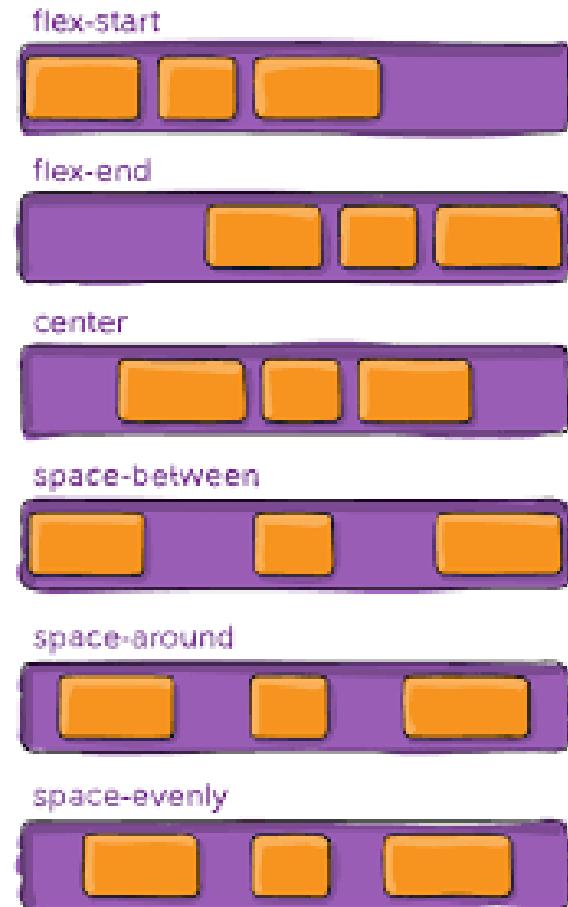
# FLEXBOX – CONTAINER PROPERTIES

- **flex-wrap:** whether flex items should stay in one line
  - **nowrap** (default): all flex items will be on one line
  - **wrap**: flex items will wrap onto multiple lines, from top to bottom.
  - **wrap-reverse**: flex items will wrap onto multiple lines from bottom to top
- **flex-flow:** shorthand for flex-direction & flex-wrap
  - E.g., column wrap



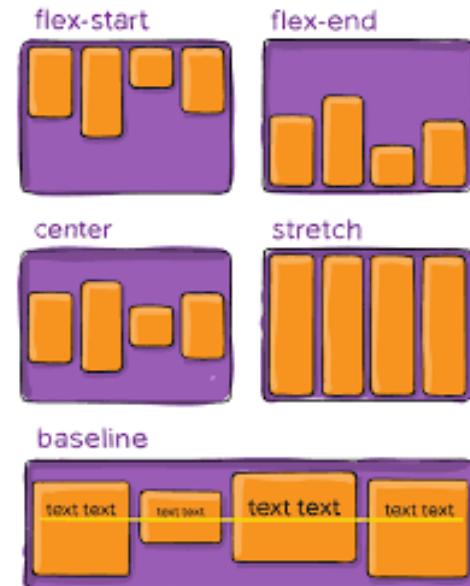
# FLEXBOX CONTAINER PROPERTIES

- **justify-content:** alignment along **main axis**
  - Be aware in case of reverse start & end change!
  - **flex-start (default):** items are packed toward the start of the flex-direction.
  - **flex-end:** items are packed toward the end of the flex-direction.
  - **center:** items are centered along the line
  - **space-between:** items are evenly distributed in the line; first item is on the start line, last item on the end line
  - **space-around:** items are evenly distributed in the line with equal space around them.
  - **space-evenly:** items are distributed so that the spacing between any two items (and the space to the edges) is equal.



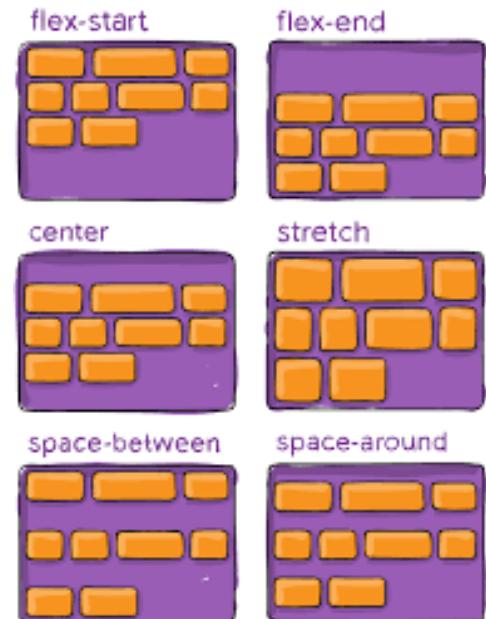
# FLEXBOX CONTAINER PROPERTIES

- **align-items:** alignment along cross axis
  - **stretch (default):** stretch to fill the container
  - **flex-start:** items are placed at the start of the cross axis
  - **flex-end:** items are placed at the end of the cross axis.
  - **center:** items are centered in the cross-axis
  - **baseline:** items are aligned such as their baselines align



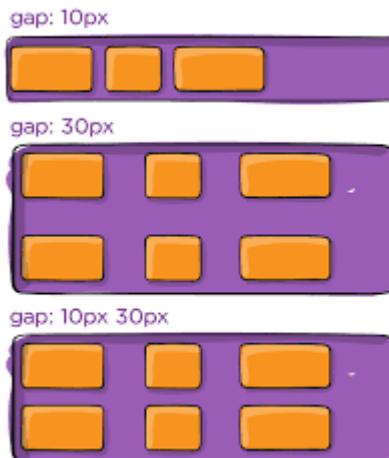
# FLEXBOX CONTAINER PROPERTY

- **align-content:** alignment of flex container's lines
  - has no effect if there is only one line of content
  - **flex-start:** items packed to the start of the container
  - **flex-end:** items packed to the end of the container
  - **center:** items centered in the container
  - **stretch:** lines stretch to take up the remaining space
  - **space-between:** items evenly distributed; the first line is at the start of the container while the last one is at the end
  - **space-around:** items evenly distributed with equal space around each line
  - **space-evenly:** items are evenly distributed with equal space around them



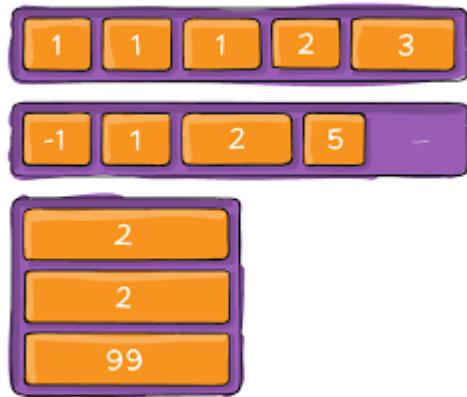
# FLEXBOX CONTAINER PROPERTY

- **gap, row-gap, column-gap:** explicitly controls the space between flex items.
  - applies spacing only between items not on the outer edges



# FLEXBOX ITEM PROPERTY

- **order:** By default, flex items are laid out in the source order. However, the order property controls the order in which they appear in the flex container.
- The initial order of all flex items is **zero (0)**
- Ordering starts with the lowest number (negative values are allowed)



# FLEXBOX ITEM PROPERTY

- **flex-grow:** ability for a flex item to grow if necessary
  - A flex grow factor is a <number> which determines how much the flex item will grow relative to the rest of the flex items
  - If all items have flex-grow set to 1, the remaining space in the container will be distributed equally to all children.
  - If one of the children has a value of 2, that child would take up twice as much of the space as either one of the others (or it will try, at least).

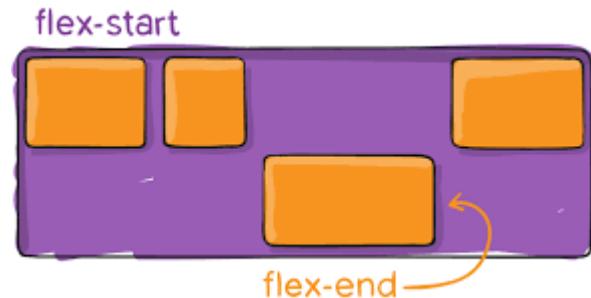


# FLEXBOX ITEM PROPERTY

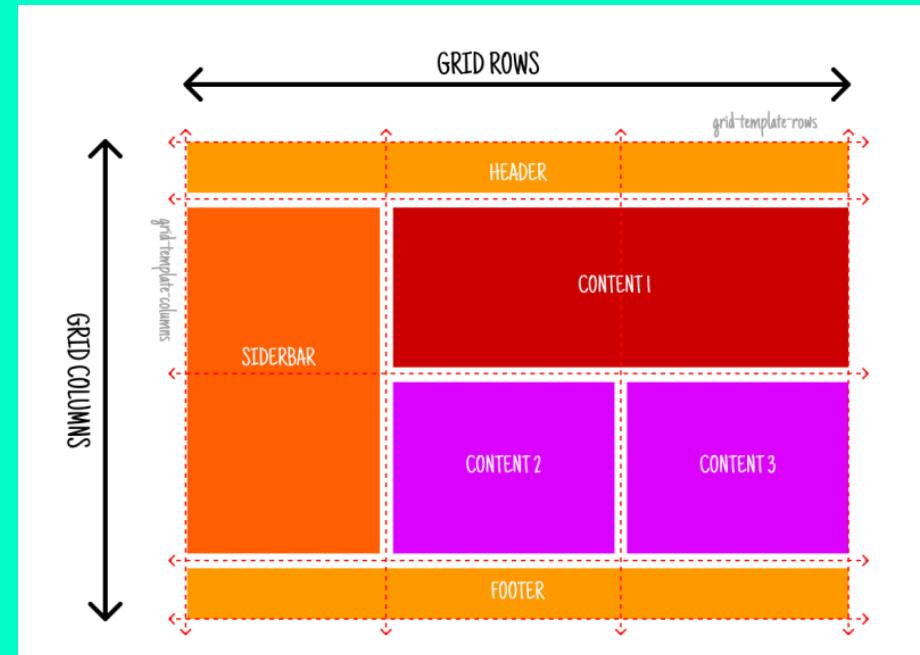
- **flex-shrink:** sets the flex shrink factor of a flex item
  - A flex shrink factor is a <number> which determines how much the flex item will shrink relative to the rest
  - “Opposite” of flex-grow
- **flex-basis:** This defines the default size of an element before the remaining space is distributed.
  - It can be a length (e.g. 20%, 5rem, etc.) or a keyword.
  - The auto keyword means “look at my width or height property”
- **flex:** shorthand for flex-grow, flex-shrink and flex-basis
  - The second and third parameters (flex-shrink and flex-basis) are optional
  - The default is 0 1 auto, but if you set it with a single number value, like `flex: 5;`, that changes the flex-basis to 0% (`flex-grow: 5; flex-shrink: 1; flex-basis: 0%;`)
  - It is **recommended** that you use this shorthand property.

# FLEXBOX ITEM PROPERTY

- **align-self:** This allows the default alignment (or the one specified by align-items) to be overridden for individual flex items.
  - Values equal to align-items
  - **center:** Alignment at the middle of the container
  - **flex-start:** Alignment at the beginning of the container (top)
  - **flex-end:** Alignment at the end of the container (bottom)
  - **stretch:** fills the container
  - **baseline:** Alignment on baseline



# CSS GRIDS



# CSS GRID LAYOUT

- A layout method to arrange elements in two dimensions.
- A layout method for complex page-layouts.
- Grid container & grid items, the (direct) children of the container.
  - The container defines the rows and columns, areas and item alignment.
  - The item can be placed precisely within the grid and override the alignment specified by the container.
- Grid items can again become grid containers.
- Playground: <https://cssgridgarden.com/#de>

```
.container {
 display: grid;
}
```

# GRID VS FLEXBOX

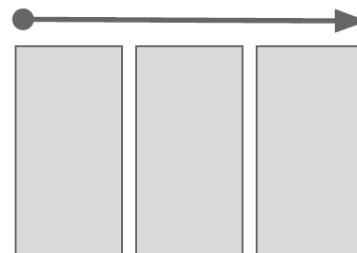
## Grid

- Two dimensions (Rows and Columns)
- complete layout with header area, two-column body and a footer
- Layout-first



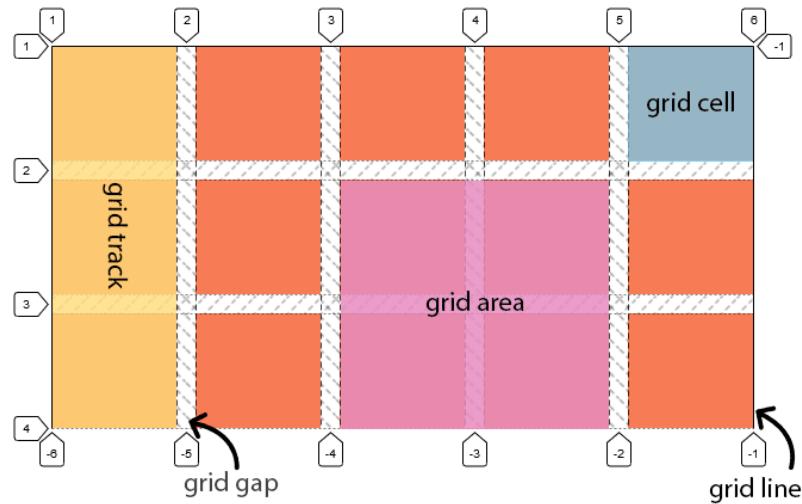
## Flexbox

- One dimension
- for stringing together elements of a line, e.g., a main menu
- Content-first



# GRID TERMINOLOGY

- **Grid container:** element, on which `display:grid` is applied
- **Grid item:** direct children of the grid container
- **Grid line:** The dividing lines that make up the structure of the grid. They can be either vertical (“column grid lines”) or horizontal (“row grid lines”) and reside on either side of a row or column.
- **Grid cell:** The space between two adjacent row and two adjacent column grid lines.
- **Grid track:** The space between two adjacent grid lines. You can think of them as the columns or rows of the grid.
- **Grid area:** The total space surrounded by four grid lines. A grid area may be composed of any number of grid cells.



# PROPERTIES FOR GRID CONTAINER – DEFINING ROWS AND COLS

- **grid-template-columns & grid-template-rows**
  - Defines the columns and rows of the grid with a space-separated list of values, i.e., track size.
  - Track size: can be a length, a percentage, or a fraction of the free space in the grid using the fr unit
    - The fr unit allows you to set the size of a track as a fraction of the free space of the grid container.
    - Auto – space is automatically calculated based on size of children
  - repeat-Fct: allows to definition of multiple rows/cols
    - grid-template-rows: repeat(4, 1fr);
  - Rows/Cols can also be named
    - grid-template-columns: [first] 40px [line2] 50px [line3] auto [col4-start] 50px [five] 40px [end];

# PROPERTIES FOR GRID CONTAINER – DEFINING ROWS AND COLS

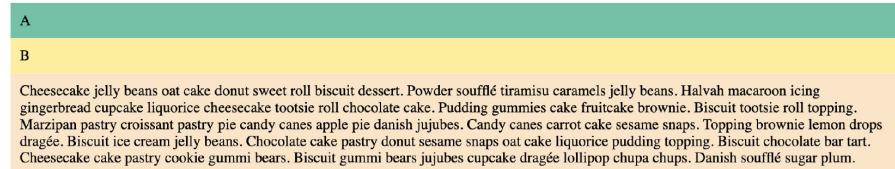
```
grid-template-columns: 1fr 1fr 1fr;
grid-template-columns: repeat(3, 1fr);
```



```
grid-template-rows: repeat(3, auto);
```

```
grid-template-columns: repeat(10, 1fr);
```

A	B	C	D	E	F	G	H	I	J
		Wafer icing tiramisu jujubes. Gummi bears chupa chups pudding marzipan jelly beans danish sweet roll dragée. Gummi bears marshmallow							

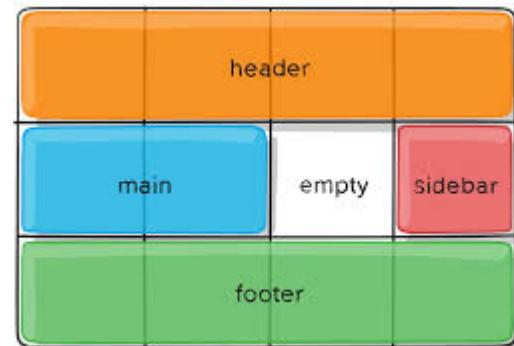


# PROPERTIES FOR GRID CONTAINER – DEFINING AREAS

- **grid-template-areas**

- Defines a grid template by referencing the names of the grid areas which are specified with the grid-area property of the grid item.
- Repeating the name of a grid area causes the content to span those cells.
- A period (.) signifies an empty cell.

```
.item-a { .container {
 grid-area: header; display: grid;
}
.
item-b { grid-template-columns: 50px 50px 50px 50px;
 grid-area: main; grid-template-rows: auto;
}
.
item-c { grid-template-areas:
 grid-area: sidebar; "header header header header"
}
.
item-d { "main main . sidebar"
 grid-area: footer; "footer footer footer footer";
}
```

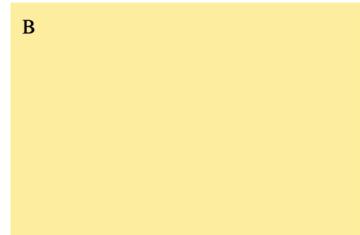


# PROPERTIES FOR GRID CONTAINER – GAP

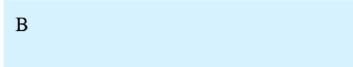
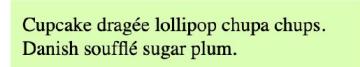
- **column-gap & row-gap**

- Specifies the size of the grid lines.
- You can think of it like setting the width of the gutters between the columns/rows.
- gap can be used as shorthand for both (row-gap column-gap)

```
grid-template-columns: repeat(3, 1fr);
grid-template-rows: repeat(2, auto);
row-gap: 1em;
column-gap: 2em;
```



Cheesecake jelly beans oat cake donut sweet roll biscuit dessert. Powder soufflé tiramisu caramels jelly beans. Pudding gummies cake fruitcake brownie. Biscuit tootsie roll topping. Cheesecake cake pastry cookie gummi bears. Biscuit gummi bears jujubes cupcake dragée lollipop chupa chups. Danish soufflé sugar plum.

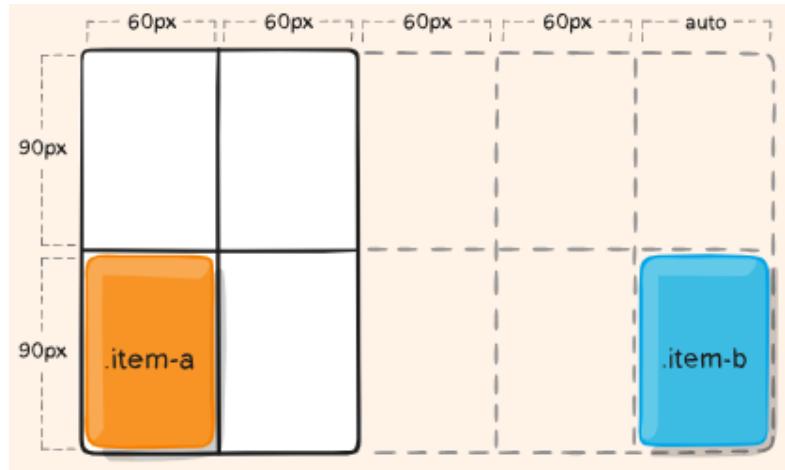


# PROPERTIES FOR GRID CONTAINER – AUTO COLS & ROWS

- **grid-auto-columns/grid-auto-rows**

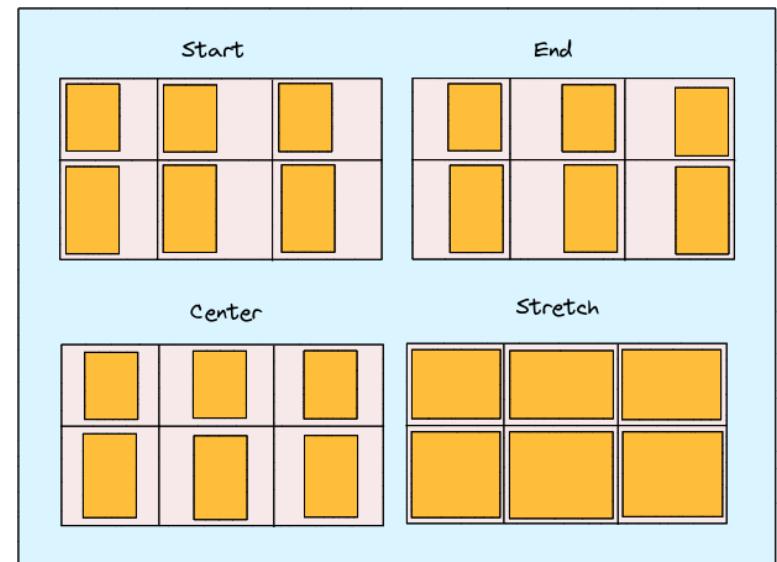
- Specifies the size of any auto-generated grid tracks (aka implicit grid tracks).
- Implicit tracks get created when there are more grid items than cells in the grid or when a grid item is placed outside of the explicit grid.

```
.container {
 grid-template-columns: 60px 60px;
 grid-template-rows: 90px 90px;
 grid-auto-columns: 60px;
}
```



# PROPERTIES FOR GRID CONTAINER

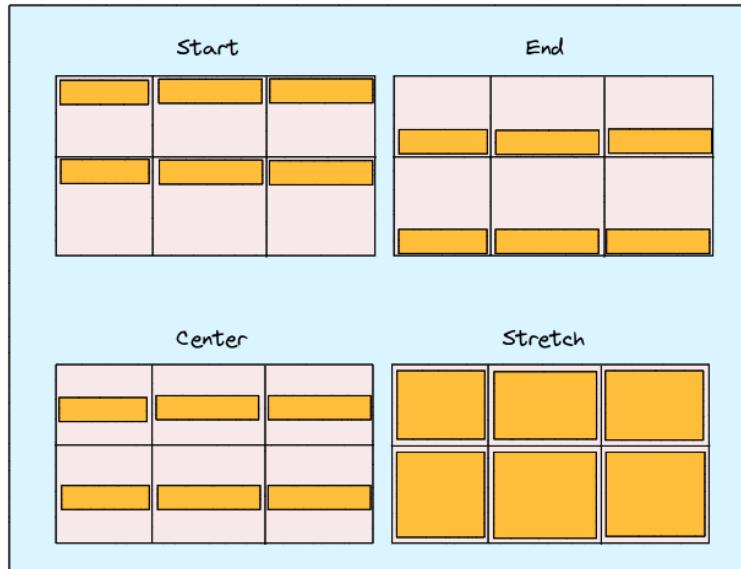
- **justify-items**
  - Alignment of items along row-axis
  - Values: start | end | center | stretch (default)
- **place-items**
  - Shorthand for  
`<align-items> <justify-items>`



# PROPERTIES FOR GRID CONTAINER

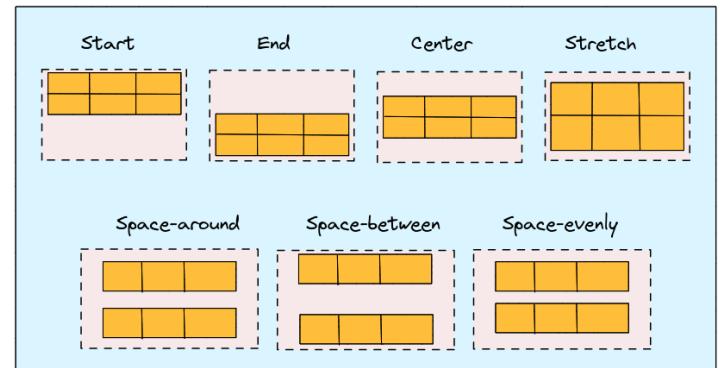
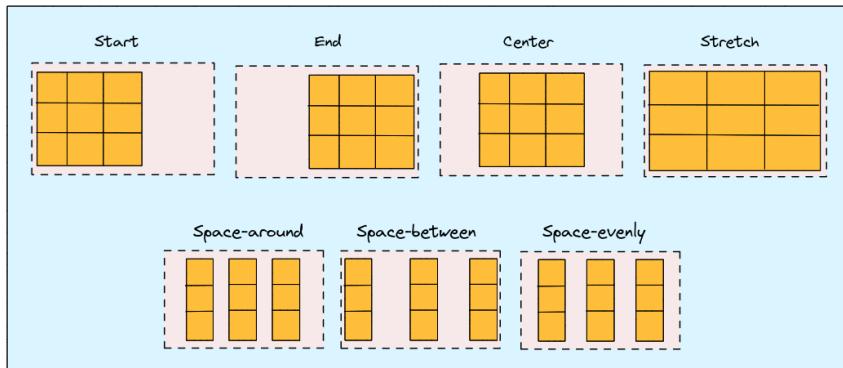
- **align-items**

- Alignment of items along column-axis
- Values: start | end | center | stretch (default)



# PROPERTIES FOR GRID CONTAINER

- **justify-content:** alignment of the grid within its container along the row axis
- **align-content:** alignment of the grid within its container along the column axis



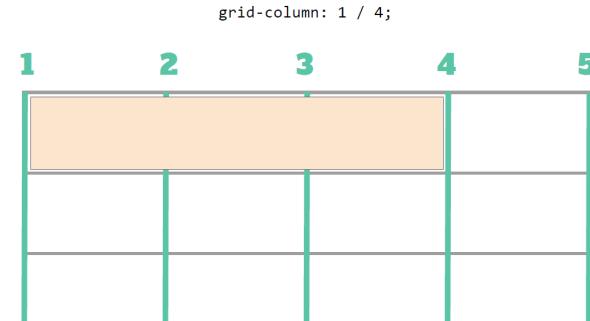
# PROPERTIES FOR GRID ITEMS

- **grid-column & grid-row**

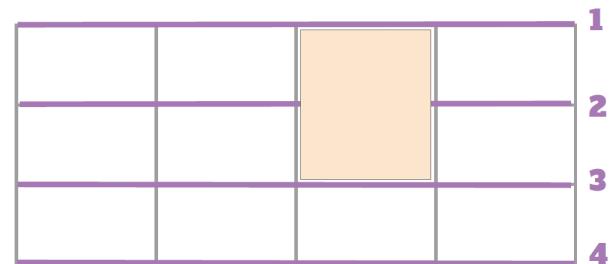
- specify a grid item's location within a grid container

```
.item {
 grid-column: <start-line> / <end-line> |
 <start-line> / span <value>;

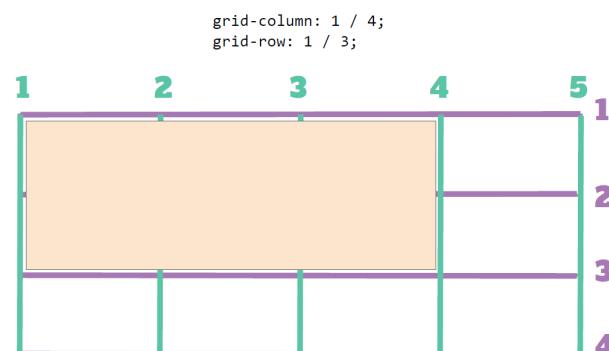
 grid-row: <start-line> / <end-line> |
 <start-line> / span <value>;
}
```



grid-column: 1 / 4;



grid-row: 1 / 3;



grid-column: 1 / 4;  
grid-row: 1 / 3;

# PROPERTIES FOR GRID ITEMS

- **grid-area**

- The grid-area property is used to assign a grid item to a named grid area, specified by the grid-template-areas property on the parent container.



```
.container {
 display: grid;
 grid-template-areas:
 "header header"
 "sidebar main"
 "footer footer";
 grid-template-rows: 80px 1fr 80px;
 grid-template-columns: 200px 1fr;
 background-color: #DAF5FF;
}

.header {
 grid-area: header;
 background-color: #89375F;
}

.sidebar {
 grid-area: sidebar;
 background-color: #E8A0BF;
}

.main {
 grid-area: main;
 grid-row-start: 2;
 grid-row-end: 4;
 background-color: #BA90C6;
}

.footer {
 grid-area: footer;
 background-color: #C0DBEA;
}
```

# PROPERTIES FOR GRID ITEMS

- **justify-self**
  - controls the **horizontal** alignment of a grid item within its cell
  - Values: start | end | center | stretch | auto
- **align-self**
  - controls the **vertical** alignment of a grid item within its cell
  - Values: start | end | center | stretch | auto.
- **place-self**
  - shorthand property
  - Values: <justify-self> <align-self>

# GRID: SPECIAL UNITS, FUNCTIONS AND KEYWORDS

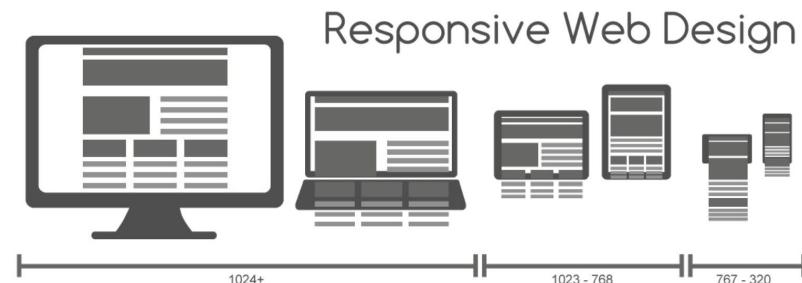
- **fr:** fraction of the free space in the grid (grid-template-columns: 1fr 2fr;)
- **minmax():** to set boundaries for flexible units
  - grid-template-columns: 1fr minmax(300px, 2fr);
  - The second column will never shrink further than 300px.
- **repeat():** repeat parts of your column/row definitions (grid-template-columns: repeat(3, 1fr);)
- **auto-fit & auto-fill:** can be used instead of a number in repeat notation
  - repeat(auto-fit, minmax(200px, 1fr)) or
  - repeat(auto-fill, minmax(200px, 1fr))
  - In case there is space left in the first grid line, auto-fit collapses empty columns to a width of 0.

# RESPONSIVE DESIGN AND MEDIA QUERIES

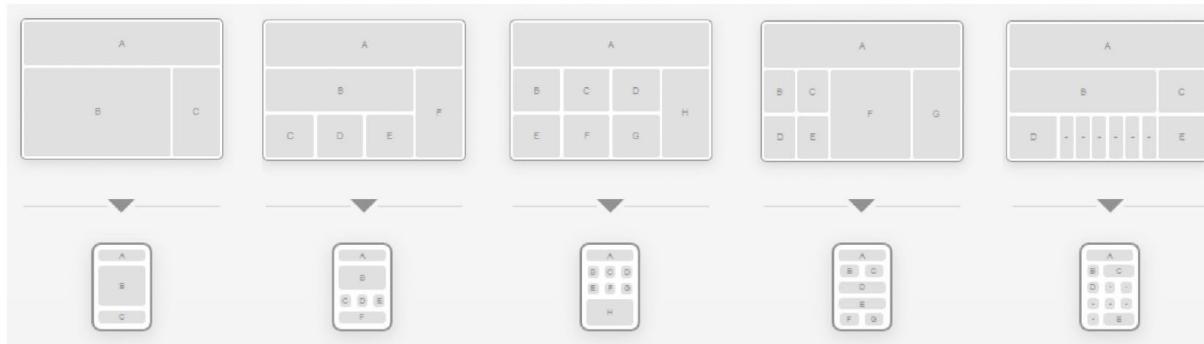
Oh, no... it looks ugly on  
the smartphone



# WHAT IS RESPONSIVE DESIGN?



- „Responsive web design“ (RWD) describes an approach to make websites look good on a variety of devices, independent of their screen sizes.
- Mobile First approach
- Fluid and adaptive design (breakpoints)
- Content Choreography



# RESPONSIVE DESIGN – HOW?

- **Media queries**
  - CSS module to target specific media types (screen, print) and features (min-width, max-width, resolution, prefers-color-scheme, prefers-reduced-motion,...)
- **Responsive images**
  - Make image resize on smaller screens and only load the image size and file format you need using the <picture> tag and the srcset and sizes attributes
- **Fluid layouts**
  - Use flex and grid layouts with their special features, viewport units and the clamp() function.
  - clamp() CSS function clamps a middle value within a range of values between a defined minimum bound and a maximum bound.
    - function takes three parameters: a minimum value, a preferred value, and a maximum allowed value.
- **Fluid typography**
  - Make the font size adapt to the viewport width using the vw unit and the clamp() function.

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

meta tag gives the browser instructions on how to control the dimensions and scaling of the page

width can be a fixed width e.g. 600px or device-width

# CSS MEDIA QUERIES

- Consist of a media type (screen, print) and one or more features that can be either true or false.
  - Features can be combined by using logical operators.
  - If the media type matches and all features evaluate to true the corresponding styles are applied.

```
@media (max-width: 720px) {
 /* tablet and smaller style go here */
}

@media (min-width: 500px) and (max-width:
600px) {
 /* layout 500-600px style go here */
}
```

```
@media print {
 /* print style go here */
}

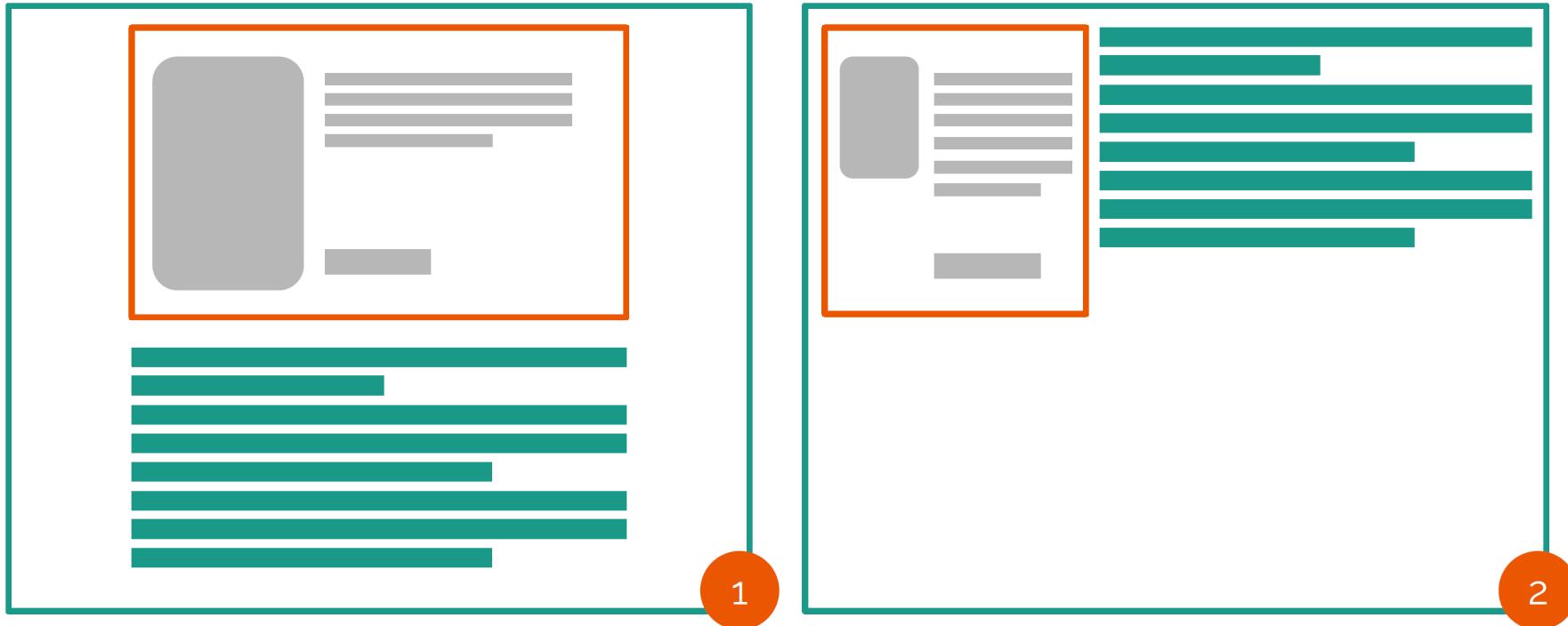
@media (orientation: landscape) {
 /* landscape style go here */
}
```

# CSS MEDIA QUERIES – BEST PRACTICES

- always in one direction (max- or min-width)
- as little as possible in media queries & as much as possible possible through normal instructions (e.g. images maximum width)
- Media queries are slowly being replaced by new CSS instructions
- Mobile-first (most end devices today are smartphones)

# WHAT MEDIA QUERIES CAN'T DO...

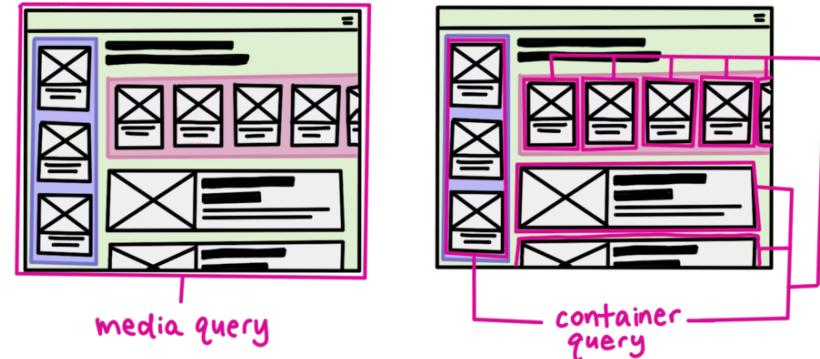
- Imagine you build a component library with many different components like buttons, form fields, cards, media elements, progress bars, and more...
- Another developer, using the components you built, can place the components anywhere. You never know where your components will end up and how much space they will have in a given layout.
- With media queries you can change the look/layout of a component only based on the total screen size, which leads to the following problem...



- The same card component (orange border) is placed in the main section (1) and in the sidebar (2). The viewport width is the same for both views, but the card component styles for this viewport size don't work well when the card doesn't have enough space...

# CONTAINER QUERIES

- Container queries solve this problem
  - New since 2023
- Instead of writing a query and checking the viewport's min-or max-width, we can provide styles based on the item's container size.
  - Two styles of container queries
    - Style queries: checking if a CSS property has a certain value
    - Size queries: ask for the container's size
      - Container needs to set the **Container type**
- You can use the same component in different places on your website/app and the layout adjusts perfectly depending on its container's size.



# CONTAINER QUERIES

- Container queries require a size container.
  - represents a reference point, or containment context, for container queries
    - For queries in the inline direction, set `container-type: inline-size;`, and for queries in both directions, set `container-type: size;`.
- The actual container query is similar to a media query and is created by the rule **@container**.

```
<div class="post">
 <div class="card">
 <h2>Card title</h2>
 <p>Card content</p>
 </div>
</div>
```

```
.post {
 container-type: inline-size;
}
```

```
/* Default heading styles*/
.card h2 {
 font-size: 1em;
}
/* If container is larger than 700px */
@container (min-width: 700px) {
 .card h2 {
 font-size: 2em;
 }
}
```

# RESPONSIVE IMAGES

- We don't want to display a large image on small screens (and load more data than necessary) or have to scale up a small image on large screens.
- We always want to see a perfectly cropped image when there's not enough space to show the full picture.
  - Use max-width property
- Problem: Browser does not know how large an image is before it loads it
  - large images are important for high resolutions (e.g. Retina displays) - but performance is poor on mobile devices
- **srcset and sizes for images**
  - give the browser a choice of different sized images so that it can select the appropriate one (either image width or resolution)

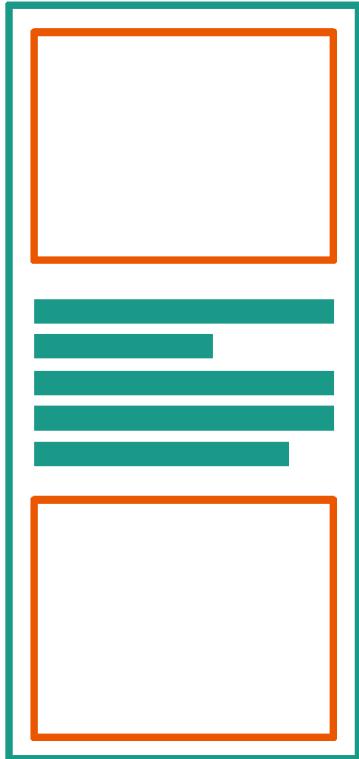
# RESPONSIVE IMAGES

```

```

- **srcset** defines the set of images we will allow the browser to choose between, and what size each image is.
- **sizes** defines a set of media conditions (e.g. screen widths) and indicates what image size would be best to choose, when certain media conditions are true

**sizes="**(max-width: 800px) 95vw, 60vw"



Small screen:  
- images' width: 95vw

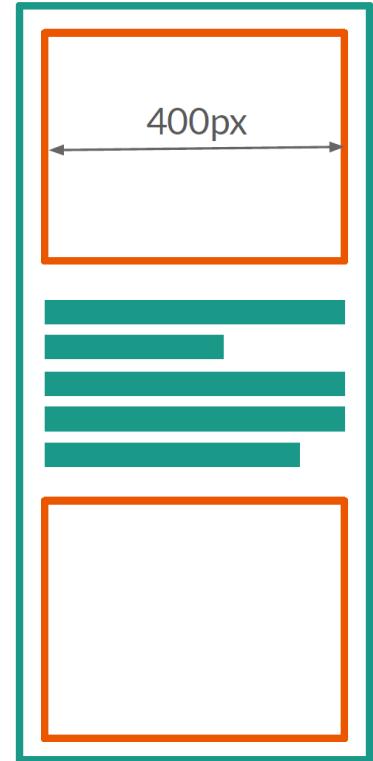


Large screen:  
- images' width: 60vw

# RESPONSIVE IMAGES

```
srcset="minerva-400.jpg 400w,
minerva-800.jpg 800w,
minerva-1200.jpg 1200w"
```

- The browser can choose from 3 images depending on the device's pixel density.
  - device pixel ratio of 1.0: minerva-400.jpg
  - device pixel ratio of 2.0: minerva-800.jpg (because an image twice the size is needed to fill the available device pixels)



# RESPONSIVE IMAGES

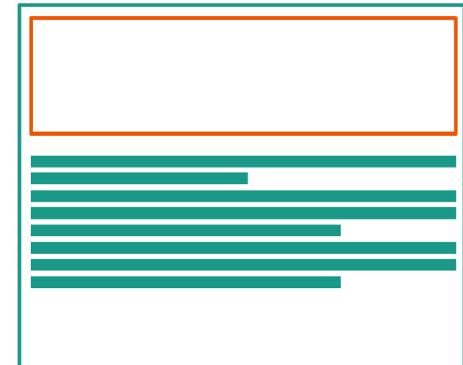
- The **<picture>** element can be used to provide...
  - different versions of the image when it comes to aspect ratio (e.g. a square image to be used on mobile screens but a 16:9 image on large screens)
  - different image file types to provide newer image formats for browser that support them and fallbacks for older browsers

# RESPONSIVE IMAGES

```
<picture>
 <source media="(max-width: 600px)" srcset="minerva-square.jpg" />
 <source media="(min-width: 601px)" srcset="minerva-wide.jpg" />

</picture>
```

- On screens up to 600px width the image minerva-square.jpg is loaded.
- On screens starting from 601px width the image minerva-wide.jpg is loaded.
- Browsers not understanding the picture element use the good old img element and loadminerva.jpg.



# RESPONSIVE IMAGES

```
<picture>
<source srcset="minerva.webp" type="image/webp" />

</picture>
```

- Browsers that support the image/webp file format load minerva.webp.
- Other browsers load minerva.jpg.

# USEFUL THINGS IN CSS



# CSS CUSTOM PROPERTIES

- Custom properties make it possible to reuse property values in a CSS file.
- Declaration: A custom property always starts with a double hyphen (--) and is case-sensitive.
  - `--main-color: #49735f;`
  - `--body-font-family: Helvetica, Arial, sans-serif;`
- Usage: Use the var() function to access the custom property's value.
  - `color: var(--main-color);`
  - `font-family: var(--body-font-family);`

# CSS CUSTOM PROPERTIES

- Custom properties can be declared in any rule block in a CSS file and are then available for the selected element and all its children; they inherit the custom property.

```
article {
 --accent-color: #a66460;
}

.article-header {
 color: var(--accent-color); /* works */
}

footer {
 color: var(--accent-color); /* does not work */
}
```

```
<article>
 <header class="article-header">
 <h1>CSS Custom properties</h1>
 </header>
 </article>
 <footer>
 <p>Some text...</p>
 </footer>
```

# CSS CUSTOM PROPERTIES

- To provide a fallback value in case the CSS custom property used is not available use the following syntax:
  - var(--accent-color, deeppink);
- To make values available throughout your CSS it's common to use the :root pseudo-class selector.
  - The :root selector matches the root element in a document tree, which is the <html> element in HTML files.

```
:root {
 --accent-color: #a66460;
 --text-color: #242424;
}
```

# IMPORT OF STYLESHEETS

- CSS Files can get rather long -> separate them into several ones
- The **@import** CSS at-rule is used to import style rules from other valid stylesheets.
- An @import rule must be defined **at the top** of the stylesheet, before any other at-rule

```
@import "custom.css";
@import url("landscape.css") screen and (orientation: landscape);
```

# INTRODUCTION TO JAVASCRIPT

Web-Design und Programmierung

# MOTIVATION – CURRENT STATUS

- HTML5 – Hypertext Markup Language
  - Markup Language for Websites
- Allows to
  - **create documents** with headings, texts, tables, listings, hyperlinks and graphics
  - Design **forms** to interact with users, e.g., for search functions, contact and ordering options
  - Insert **multimedia elements** such as videos, music and animations as well as applications directly into documents
- Based on markup

```
<html>
<head>
<title></title>
</head>
<body>
Hallo Welt!
....
</body>
</html>
```

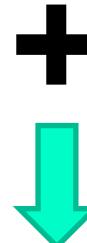
# MOTIVATION – CURRENT STATUS

- CSS – Cascading Style Sheets
  - Separation of content/structure and formatting
    - Content and structure: HTML
    - Formatting: CSS
  - Layout

## HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <title>WDP3</title>
</head>
<body>
 <p>Hallo Welt</p>
</body>
</html>
```

Content and Structure



Haloo Welt

```
p {
 color: yellow;
 background-color: green;
 font-family: "Comic Sans MS";
 font-size: 30px;
}
```

Formatting

Result

# MOTIVATION

- **Static websites**
  - Present "only" content (text and images)
  - No dynamic elements
  - Menus, slideshows, etc....
- **Hardly any user interaction possible**
  - Only forms and hyperlinks
- **Typical Use Cases for JS**
  - Dynamic manipulation of web pages (via DOM)
  - Plausibility checks of form entries
  - Suggestion of search terms
  - Asynchronous sending/receiving of data (AJAX)

Simple registration form

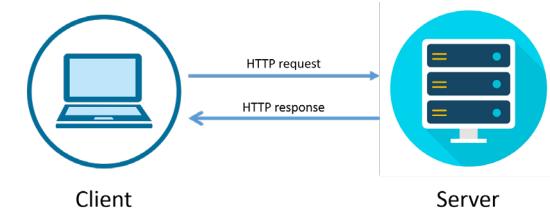
Username:	<input type="text"/>	Enter username.
Password:	<input type="password"/>	confirm must be equal
Confirm Password:	<input type="password"/>	
E-mail:	<input type="text"/>	Enter email.



# BASICS

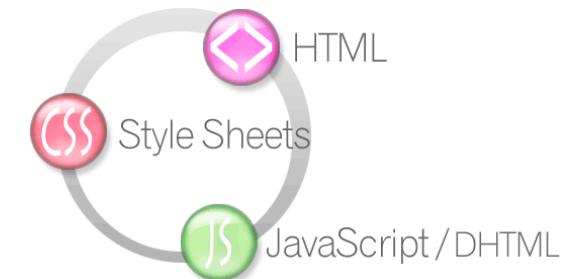


- JavaScript is a *client-side scripting language*
- What does „client-side“ mean?
  - Programmes **run** entirely in the **browser**
  - **No server requirements** necessary
  - Dependencies on browser and operating system
  - Based on HTTP request/response model
- What does „scripting language“ mean?
  - Often differences to "conventional" programming languages
    - **Implicitly declared variables**
    - **Dynamic and weak typing**
    - **Interpreted** (no translation step)
  - Often embedded in host language
    - HTML in case of JavaScript



# EMBEDDING OF JAVASCRIPT INTO HTML

- JavaScript commands can be placed in several places in an HTML file:
  - Between the `<script>` and `</script>` tags
  - **In an external file**
  - In the form of an HTML link
  - As parameters of HTML tags
- **Options for embedding**
  - Directly in the body using the `<script>` tag
  - Declare functions in the head area and call them in the respective elements (inline)
  - Swap to an **external JavaScript file** and include in the head or **end of body area.**



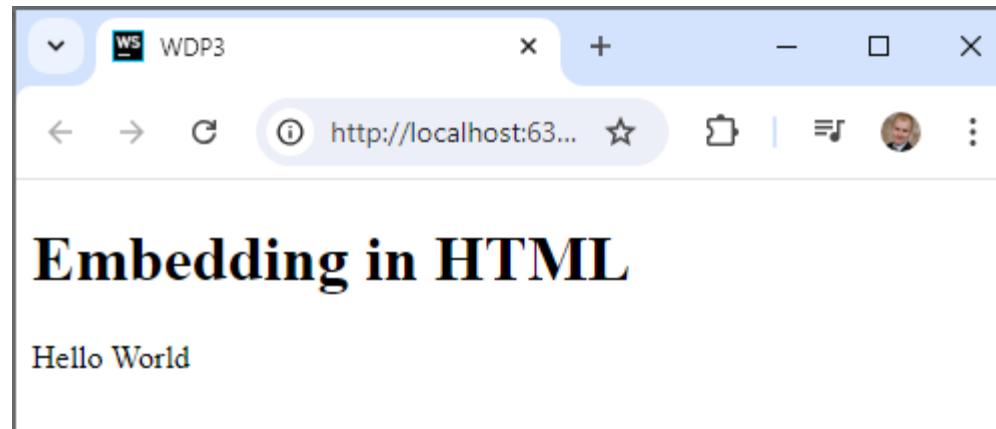
# EMBEDDING OF JAVASCRIPT INTO HTML

- **Script Tag in Body**

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <title>WDP3</title>
</head>
<body>
<h1>Embedding in HTML</h1>
<script type="text/javascript">
 document.write("Hello World");
</script>
</body>
</html>
```

**Problems:**

- **close mixing**
- **confusing**
- **does not work if JS is not supported  
(progressive enhancement)**



# EMBEDDING OF JAVASCRIPT INTO HTML

- Link with JavaScript

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <title>WDP3</title>
</head>
<body>
 <h1>Embedding in HTML</h1>
 Say hello
</body>
</html>
```



Avoid at all costs!

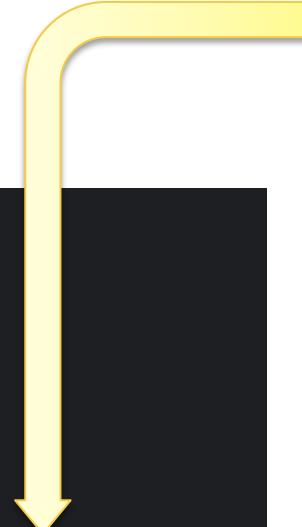
# EMBEDDING OF JAVASCRIPT INTO HTML

- External file at the end of the body tag

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <title>WDP3</title>
</head>
<body>
 <h1>Embedding in HTML</h1>
 <script type="text/javascript" src="main.js"></script>
</body>
</html>
```

Preferred variant

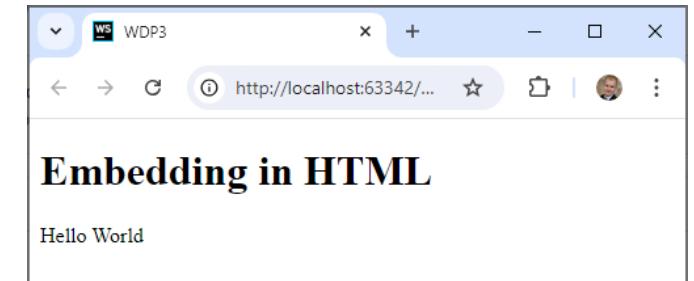


main.js

```
function hello(){
 document.write("Hello World");
}

hello()
```

The content of the file  
is processed during  
loading



- Advanced concepts

- async and defer
  - <https://www.mediaevent.de/javascript/programm-struktur.html>

# CHALLENGES WITH JAVASCRIPT

- **JavaScript is a client-side language**
  - Dependence on the respective browser + version of the user
  - Different manufacturers interpret code differently
  - Complex tests in different browsers
  - Frameworks provide a remedy
- **Effort towards standardisation:**
  - 1997 ECMA-262 (ECMAScript), which became the ISO standard ISO/IEC 16262 in 1998.
  - Last major update from ES5 to ES6
    - Also associated with new naming concept with year number (ES6=ES2015)
  - Current latest versions ES2023 (ES14)
    - Attention to browser support

CHALLENGE ACCEPTED.



# CHALLENGES WITH JAVASCRIPT

- ***Unobtrusive JavaScript***
  - Differentiation between content, formatting and behaviour
  - JavaScript as an extension of the functional scope, not as a prerequisite
- ***Progressive enhancement***
  - **Basic content** should be readable by all web browsers
  - **Basic functionality** should be able to be interpreted by any browser
  - Layout is provided by external CSS
  - **Additional functionality** is provided by unobtrusive, externally linked JavaScript
- ***Graceful degradation***
  - Error or non-support of functionalities does not lead to an overall crash, but only reduces functionality



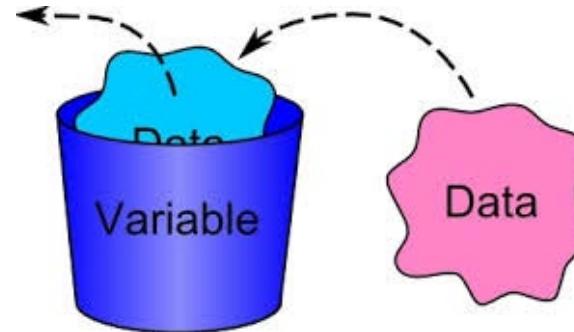
cc creative commons

JAVASCRIPT LANGUAGE

JS

# VARIABLES

- **Variable** has
  - A (unique) name  
(JavaScript is case sensitive,  
Variable x ≠ X)
  - **NO explicit data type**
    - Data type is determined at runtime
    - Data type can change during runtime
  - JavaScript is weakly, dynamically typed language
- Declaration – no type definition
  - Explicit: with keyword **let**
    - (or **var** in elder versions)
  - Implicit: without any keyword
    - **AVOID** this (error strict mode)



```
let x = 18;
```



```
y=18
```



# IMPLICIT DATA TYPES

- **Primitive**
  - **String** for sequence of characters
  - **Number** for all numerical values
    - Floating point numbers : Number and decimal part are separated with . e.g., `x = 1.75`
    - JavaScript does not provide different types for numbers (except BigInt)
  - **Boolean** for true/false values
  - **Undefined** (Value is also undefined)
  - **Symbol** (unique and immutable primitive value)
- **Complex**
  - **Object** for anything else

# IMPLICIT DATA TYPES

- Number

```
let a = 42; // positive
let b = -42; // negative
let c = 4.2; // fractional
let d = 4.2e10; // exponent
let e = Infinity; // infinite
let f = - Infinity; // - infinite
let g = NaN; // not a number
```

- Boolean

```
console.log(3 > 2);
// → true
console.log(3 < 2);
// → false
```

- String

```
"Hello World!" // normal string
'Hello World!' // normal string
`Hello World!` // template string
console.log('Hello\nWorld!');
// Output: Hello
// Output: World!
console.log("Hello" + " " + "World" + "!");
// Hello World!
console.log(`half of 100 is ${100 / 2}`);
// Output: half of 100 is 50
console.log('half of 100 is ${100 / 2}');
// Output: half of 100 is ${100 / 2}
```

- Undefined/Null

```
let a = null;
let b = undefined;
// undefined and null are different types
undefined === null // false !!!
// undefined and null can be cast to the same value
undefined == null // true
```

# CONSTANTS

- ES5 does not distinguish between variables and constants
  - Constants only by convention  
(Var name in capital letters)
  - Emulates via non-overwritable object properties
- ES6 provides its own keyword **const**
  - Attempt to set const leads to error

```
var konstanten = {}
Object.defineProperty(konstanten, 'MAX',{
 writable : false,
 enumerable: true,
 configurable: false,
 value: 50
})
```

```
const MAXIMUM = 5;
console.log(MAXIMUM);
```

# VARIABLES (SINCE ES6)

- `const` is a signal that **the identifier won't be reassigned**.
- `let`, is a signal that **the variable may be reassigned**, such as a counter in a loop, or a value swap in an algorithm. It also signals that the variable will be used **only in the block it's defined in**, which is not always the entire containing function.
- `var` is now **the weakest signal available** when you define a variable in JavaScript. The variable may or may not be reassigned, and the variable may or may not be used for an entire function, or just for the purpose of a block or loop.

# EXPRESSION - ARITHMETIC OPERATORS

Operator	Description	Example	Result
+	Addition	a = 7 + 4	11
-	Subtraction	a = 7 - 4	3
*	Multiplication	a = 7 * 4	28
/	Division	a = 7 / 4	1.75
%	Modulo	a = 7 % 4	3
-	Negation	b = 7 a = -b	-7

# EXPRESSIONS - SHORT FORMS

Operator	Description	Short form	Long form
=	Assignment	a= 7	
+=	Addition	a+= b	a = a + b
++	Increment	a++	a = a +1
--	Subtraction	a-= b	a = a - b
--	Decrement	a--	a = a - 1
*=	Multiplication	a*= b	a = a * b
/=	Division	a/= b	a = a / b
%=	Modulo	a% = b	a = a % b

# DYNAMIC TYPING

- What operations are available on other data types (except numbers)?
  - Most importantly Strings
  - Can you calculate with strings (as before)?
  - Does JavaScript understand the following instructions and what is the result?

```
console.log("Hannes" - "Schönböck"); //NaN
console.log("Hannes" + "Schönböck"); //HannesSchönböck
console.log("3" + "17"); //317
```

**Implicit data type determines the available operations**



# DYNAMIC TYPING

- Previously operations only with the same data types
  - What happens when you mix data types?

```
console.log("10" * 7); //70
console.log("10" - 7); //3
console.log("10" + 7); //107
console.log(7 + "10"); //710
```

Attention -> frequent source of errors



# DYNAMIC TYPING AND AUTOMATIC TYPE CONVERSION

- Operator +
  - Only if **both** operands are of the dynamic type **Number**, arithmetic operation is performed.
  - Otherwise, **all operands are converted to strings** and string concatenation is performed.
  - Attention: unary + operator performs type conversion
- Operators -, \*, /
  - convert (if possible) String into Number
- "" converts to 0
- null converts 0 (due to Boolean rule)
- Undefined converts to NaN

```
let numValue = (+stringValue);
/* or */
let numValue = stringValue - 0;
/* or */
let numValue = stringValue * 1;
/* or */
let numValue = stringValue / 1;
```

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → "51"
console.log("5" * 2)
// → 10
console.log("five" * 2)
// → NaN
console.log(false == 0)
// → true
```

# LOGICAL OPERATORS

Operator	Beschreibung
!	NOT
	OR
&&	AND

a	b	OR	AND	XOR
true	true	true	true	false
true	false	true	false	true
false	true	true	false	true
false	false	false	false	false

No explicit XOR Operator ->  
Simulation ( a || b ) && !( a && b )

```
let f = false;
let res = f && otherVar;
```

**JavaScript applies short term evaluation ->**  
otherVar is not evaluated, since  
result is already known after checking f

# COMPARISON OPERATORS

Operator	Description	Example	Result
<code>==</code>	Equal	<code>a = (3==4)</code> <code>a = ("Java"=="JavaScript")</code>	<code>false</code>
<code>!=</code>	not equal	<code>a = (3!=4)</code> <code>a = ("Java"!="JavaScript")</code>	<code>true</code>
<code>&gt;</code>	greater than	<code>a = (3 &gt; 4)</code>	<code>false</code>
<code>&lt;</code>	less than	<code>a = (3 &lt; 4)</code>	<code>true</code>
<code>&gt;=</code>	greater than or equal	<code>a = (3 &gt;= 4)</code>	<code>false</code>
<code>&lt;=</code>	less than or equal	<code>a = (3 &lt;= 4)</code>	<code>true</code>

```
console.log('1' > 0); -> TRUE
console.log('a' > 2); -> FALSE (ASCII Value comparison)
console.log('12' > '9'); -> FALSE (String comparison)
console.log(12 > '2'); -> TRUE
console.log(12 > '15'); -> FALSE
```

# COMPARISON OPERATOR

- JavaScript allows two variants
  - === and !==
    - If the **two operands are of the same type** and have the **same value**, then === true and !== false
  - == and !=
    - Return the same result as === and !== if operands are of the same type.
    - For unequal types, the value is used
  - Keep attention when comparing objects
    - Do not implement equals-method

```
console.log(0 == ''); // true
console.log(0 == '0'); // true
console.log(17 == '17'); //true

console.log(0 === ''); // false
console.log(0 === '0'); // false
console.log(17 === '17'); //false
```

## ● Explicit Casting

```
console.log("5" == 5);
// → true
console.log("5" === 5);
// → false
console.log(Number("5") === 5);
// → true
console.log("5" === Number(5).toString());
// → true
console.log("5" === 5.toString());
// → Unexpected token ILLEGAL
```

# DYNAMIC TYPING

- Boolean

- Implicit conversion for logical operators
- Manual conversion via !! or Boolean(x)
- All values result in TRUE except
  - 0, NaN
  - "" (empty String)
  - null
  - undefined

```
console.log (!! "Hannes"); //true
console.log (!! 3); //true
console.log (!! undefined) //false
console.log (!! null) //false
console.log (!! 0) //false
```

Data type	True	False
Boolean	true	false
String	Non empty String	"" (empty String)
Number	Remaining values	0, NaN
Object	Any object	Null
Undefined	-	undefined

# CONTROL STRUCTURES – IF/ELSE

```
if (Bedingung) {
 //Anweisungen
}
```

```
if (Bedingung) {
 //Anweisungen
} else {
 //Anweisungen
}
```

```
if (val == 3) {
 document.write("TRUE");
} else {
 document.write("FALSE");
}
```

```
switch(expression) {
 case val1:
 //statements
 break;
 case val2:
 //statements
 break;
 //further cases
 default:
 //statement
}
```

Break interrupts execution of the block and jumps to the end of the block (here after the switch statement).

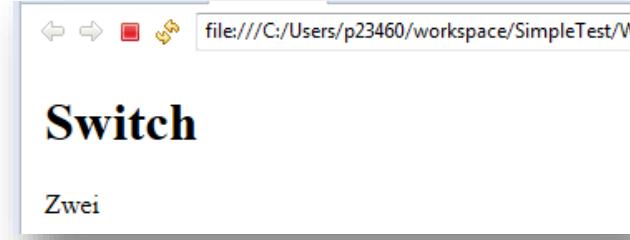
Without a break, **all statements up to the end of the switch expression** would be executed, including other case statements.

Default branch is executed if no other branch applies

# CONTROL STRUCTURES – SWITCH/CASE

```
let value = 2;

switch (value) {
 case 1 : document.write('Eins');
 break;
 case 2 : document.write('Zwei');
 break;
 case 3 : document.write('Drei');
 break;
 default : document.write('Keine Zahl');
}
```



```
let value = "1";

switch (value) {
 case 1 :
 case "1" : document.write('Eins');
 break;
 case 2 : document.write('Zwei');
 break;
 default : document.write('Keine Zahl');
}
```

The same code is to be  
executed in several cases



# CONTROL STRUCTURES - WHILE/DO-WHILE LOOPS

## While loop

(body may never be entered)

```
while (Bedingung) {
 //statements
}
```

```
let value = 1;

while (value <= 3) {
 document.write(value + "</br>");
 value++;
}
```



## Do loop

(body is always stepped on at least 1 time)

```
do {
 //statements
} while (Bedingung)
```

```
let value = 1;

do {
 document.write(value + "</br>");
 value++;
} while (value <= 3)
```



# CONTROL STRUCTURES – FOR LOOP

number of loop passes known in advance

```
for (init; condition; iteration-expression) {
 //Anweisungen
}
```

```
for (let i = 0; i <=3; i++) {
 document.write(i + "
");
}
```



**Supports** break and continue statements

# ARRAYS AND COLLECTIONS

- Array is different to other programming languages!
  - Set and Maps
-

# ARRAYS

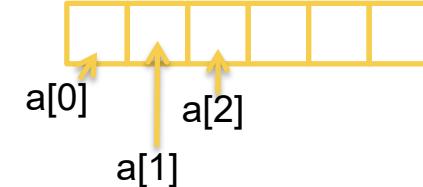
- **Array is a field with index**

- Index (often) number that starts at 0
- In strongly typed programming languages, elements must have the same data type.
- Length fixed at definition and fixed

- **JavaScript is different!**

- Since untyped language, array can contain elements with different data types
- Length does not have to be specified at definition
- Can **grow** at runtime
  - Rather vector/List semantics

`let a`



# ARRAYS - DEFINITION

- Predefined object (more on objects later)
- Container for variables of the same kind
  - Instead of a set of independent variables

- Access to array via [ ] operator and index

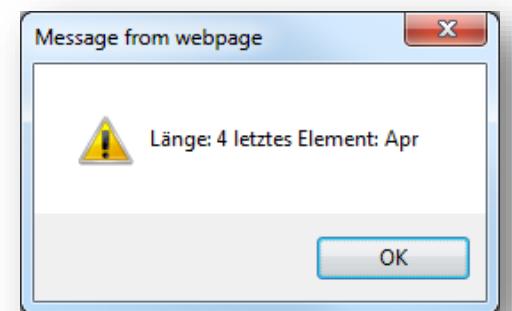
- Index starts at 0

```
let monthShort = ["Jan", "Feb", "Mar"];
alert(monthShort[0]);
```



- Adding a new element
  - Gaps allowed

```
let monthShort = ["Jan", "Feb", "Mar"];
monthShort[3] ="Apr";
alert("Länge: " + monthShort.length +
 " letztes Element: " +
 monthShort[monthShort.length -1]);
```



```
//empty Array
let a = new Array();
//empty Array
let a2 = [];
//Array with initial length of 10
let sizedArr = new Array(10);
//initialised Array
let month = new Array("Jan", "Feb", "Mar");
//short form for initialisation
let monthShort = ["Jan", "Feb", "Mar"];
```

# ARRAYS AND FOR-LOOPS

- **For-each** (since ES5)

- Suitable for Arrays

- Disadvantage: no break, continue, return allowed

```
myArray.forEach(function (value) {
 console.log(value);
});
```

- **For-in**

- Loop variable has the implicit data type string

- Not intended for use with arrays, but to iterate over properties of objects (more on this later)

```
// don't actually do this
for (let index in myArray) {
 console.log(myArray[index]);
}
```

- **For-of** (ES6)

- break, continue, return allowed

- Preferred variant to run through the entire array!

```
for (let value of myArray) {
 console.log(value);
}
```

# ASSOZIATIVE ARRAYS

- JavaScript also allows string values as index
  - Map semantics

```
let example = new Array();
example["key1"] = "Value1";
example["key2"] = "Value2";
example["key3"] = "Value3";

alert(example["key1"]);
```

# ARRAYS - STANDARD FUNCTIONS

- Common functions on arrays
  - concat() (link two Arrays)
  - join() (convert Array into String)
  - pop() (delete last element of Array)
  - push() (insert new element at the end of Array)
  - reverse() (revert order of Array)
  - shift() (remove first element of Array)
  - slice() (extract part of Array)
  - splice() (delete or add elements)
  - sort() (sort Array)

# COLLECTIONS

- In contrast to other programming languages, there were always only arrays in JS
- In ES6 there are now (at least) Map and Set available
- Maps allow to store (arbitrary) key - value pairs
- Difference to objects are own methods and arbitrary objects are allowed as keys
- Set is a list in which value may occur only once
- Check for strict equality (====)

# ITERATOREN

- Iterators simplify (iterative) access to elements stored in a particular data structure
- Iterators encapsulate knowledge how to traverse this data structure
- ES iterators provide the (only) method `next()` for this purpose
  - Returns object with properties `value` (`value`) and `done` (indicating whether iterator has reached the end or not)
  - Often in combination with new `for ... of` loop

# MAP

A screenshot of a browser's developer tools console. The tabs at the top are 'Elements' and 'Console'. The 'Console' tab is selected. Below the tabs, there are two filter icons: a magnifying glass and a funnel. The text area shows the following output:

```
Johannes Schönböck
2
1 = Johannes Schönböck
2 = Max Mustermann
1
2
Johannes Schönböck
Max Mustermann
1 = Johannes Schönböck
2 = Max Mustermann
true
false
↳
```

```
let m = new Map();
m.set(1, "Johannes Schönböck");
m.set(2, "Max Mustermann");
console.log(m.get(1));
console.log(m.size)

for (let [key, value] of m) {
 console.log(key + " = " + value);
}

for (let key of m.keys()) {
 console.log(key);
}

for (let value of m.values()) {
 console.log(value);
}

for (let [key, value] of m.entries()) {
 console.log(key + " = " + value);
}

console.log(m.has(2));
m.delete(2);
console.log(m.has(2));
```

# FUNCTIONS

Functions are a central concept in JS

# FUNCTIONS

- **Functions allow**
  - To solve complex tasks in a structured, reusable way
  - Principle of step-by-step refinement
    - Complex tasks to be broken down into smaller and smaller tasks until they can be solved easily (using their own functions)
- **Definition**
  - Interface
    - Keyword **function**
    - Unique function name
      - if there are several functions with the same name, the last one defined is called
    - List of parameters
  - Body
    - Implementation of logic
    - Return value (if needed)



```
//Definition of function without parameter
function sayHello(){
 return "Hello!";
}

<!-- Call the function -->
sayHello();
```

# FUNCTIONS

- **Java (and other typed programming languages)**
  - When function is called in Java, the
    - name and
    - number and
    - type of parameters must match **JavaScript**
- **JavaScript**
  - **ONLY** the name must match
  - Number and type of parameters are **NOT** checked
  - Parameters are **NOT** used when calling the function
    - Match is done by name only
    - Parameter list irrelevant (can be called with more or less parameters)
      - Unused parameters receive the value **undefined**
    - **No overloading** of functions supported (several functions with different parameter list)
  - **DO NOT USE THIS “FEATURE”**
    - Always call functions with corresponding parameter list!

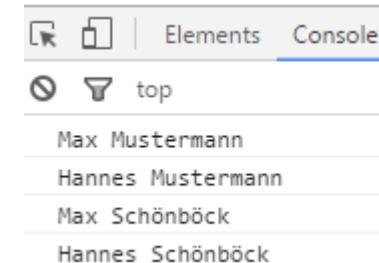


# DEFAULT VALUES FOR PARAMETERS

- Allowed since ES6
- Default parameter is used if the passed value is undefined (the value was omitted)

```
function sayHello(firstName= "Max", lastName="Mustermann") {
 console.log(firstName);
 console.log(lastName);
}

sayHello();
sayHello("Hannes");
sayHello(undefined,"Schoenboeck");
sayHello("Hannes","Schoenboeck");
```



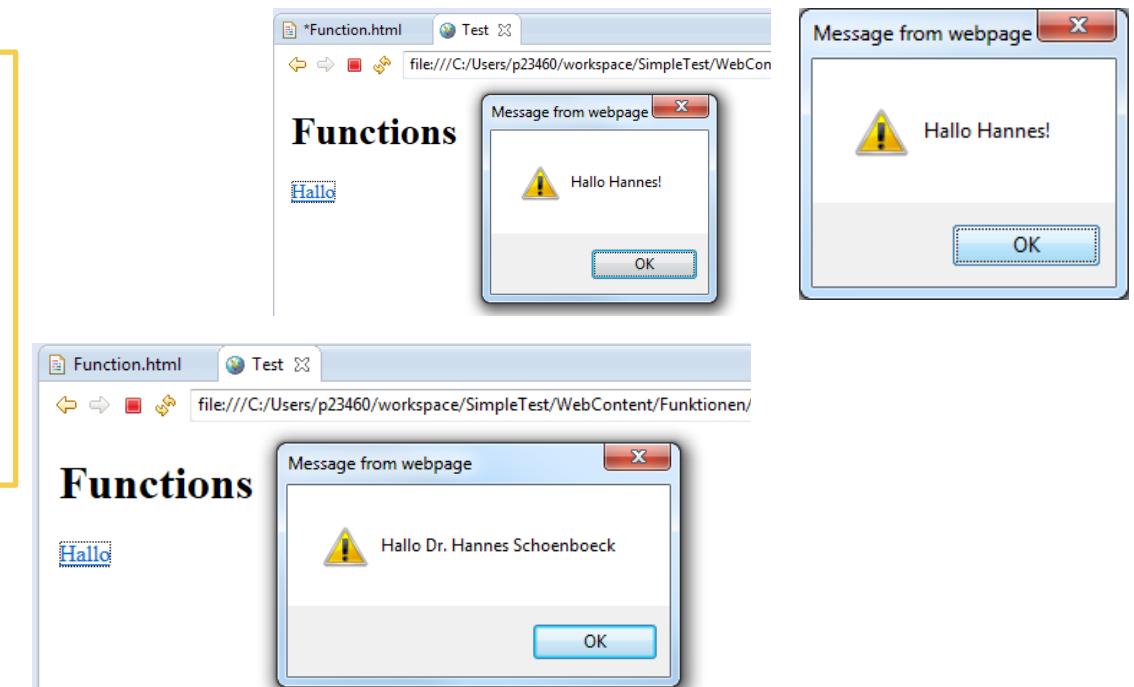
# FUNCTIONS PARAMETERS.

- How does parameter matching work in JS functions?
- Parameters are always passed as a kind of array
  - Internal name of the array is arguments

```
function sayHallo() {
 let greeting = "Hallo "
 for(let i = 0; i < arguments.length; i++) {
 greeting+= arguments[i];
 }
 alert(greeting);
}
sayHallo('Dr. ', 'Hannes ', 'Schoenboeck');
```

Check if parameter is present  
(if param != undefined)

```
//Definition with one parameter
function sayHallo(name){
 alert("Hallo " + name + "!");
 alert("Hallo " + arguments[0] + "!");
}
<!-- call of function-->
sayHallo('Hannes');
```

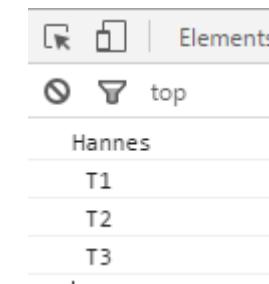


# PARAMETERS IN ES6

- Any number of function parameters
  - In ES5 via Arguments Array
- But always contains all parameters
  - ES6 introduces so-called **rest** parameters
  - Can be compared to varargs in Java

```
function addTagToArtist(name, ...tags){
 console.log(name);
 tags.forEach(function(tag){
 console.log(" " + tag);
 });
}

addTagToArtist("Hannes", "T1", "T2", "T3");
```



# ANONYMOUS FUNCTIONS

- A function can be also defined without a name.
- Anonymous function need to be assigned to a variable or used in an expression.
- The function can be executed with the () operator

```
let foo = function (text) {
 // code
}
foo('hello');
// foo is only a variable!
let bar = foo;
bar('world');
foo = 1;
foo(); //Error!! foo is not a function
```

# FUNCTIONS AND SCOPES

- Since it ES6 new keyword **let**
- Well supported in current browsers
- Block Scope
  - Variables only valid between { ... }

```
function test(x){
 console.log(y); //Reference Error: y is not defined
 if(x){
 let y = 4711;
 console.log(y); //4711
 }
 console.log(y); //Reference Error: y is not defined
}

test(true);
```

# HOISTING – „PROBLEMS“ USING VAR

- Declaration of variable is „hoisted“ at the begin of the function
- Initial value undefined

```
function hoistingTest(){
 console.log(testVar); //undefined
 var testVar = "Now I am defined";
 console.log(testVar); //Now I am defined
}
```

JS internal

```
function hoistingTest(){
 var testVar
 console.log(testVar); //undefined
 testVar = "Now I am defined";
 console.log(testVar); //Now I am defined
}
```

- Attention when covering global variables!

```
var testVar = "Global Var";

function hoistingTest(){
 console.log(testVar); //undefined
 var testVar = "Now I am defined";
 console.log(testVar); //Now I am
 defined
}
hoistingTest()
```

Declaration of testVar in function is hoisted and this is then evaluated before the global variable

- TIP: Always define variables at the beginning of the function (simulate hoisting) if you use var -> **BETTER use let**

# SCOPES OF VARIABLES

- **Global variables**
  - Variables that are defined outside of a function
  - 3 ways to create them
    - Explicitly with the keyword **let** in the global namespace
    - Implicitly without the keyword **let** in the global namespace
    - Implicitly without the keyword **let** in the "local" namespace
- Problems:
  - Pollution of the global namespace
  - Accidental name conflicts
  - No modularity
- **TIP: Avoid global variables as much as possible!**

# SCOPES OF VARIABLES

- Outside of functions -> **global**
- Within the function -> **local**
- Variables defined within a function without a keyword -> **global**
- Inner variable binds
- TIPS:
  - **Always define with let!!!**
  - **Use strict mode**

```
<script type="text/javascript">
 let expGlobalVar = "An explizit global Variable";
 impGlobalVar = "An implicit global Variable";

 function simpleFunc(){
 let y = "I am an explicit local Variable";
 x = "I am NO implicit local Variable - I am global as well";
 }

 console.log(expGlobalVar);
 console.log(impGlobalVar);
 //both calls would lead to an error, since var is not declared now
 //console.log("Check x before calling function " + x);
 //console.log("Check y before calling function " + y);
 simpleFunc();
 console.log("Check x after calling function " + x);
 // would still lead to an error since var y is local to function
 //console.log("Check y after calling function " + y);
</script>
```

An explizit global Variable  
An implicit global Variable  
Check x after calling function I am NO implicit local Variable - I am global

# ARROW FUNCTIONS (ES6)

- Up to ES5, the same keyword for functions (subroutines) as well as object methods (`function`).
  - If a function is defined within an object, it is automatically an object method.
- ES6 now offers different syntactic constructs
  - so called arrow-functions for subroutines
  - **DO NOT** use for Object methods
- Well suited for so-called callbacks (more on this in DOM)

```
//ES6
let quadrat = x => x*x;

console.log(quadrat(2));

//ES5
let quadrat = function(x) {
 return x*x;
}

console.log(quadrat(2));
```

# ARROW FUNCTIONS

- Left side function parameter, right side function body, in between =>

Definition	Description
① () => {...}	Function without parameters
x => {...}	Function with single parameter
② (x,y) => {...}	Function with several parameters
<Fkt-Param> => {return x*x}	Functions body as block
③ <Fkt-Param> => x*x	Function body as Statement (no return needed)

① `let foo = ()=>{  
 console.log("An arrow fct without params")  
}  
foo();`

② `let quadrat = x => x*x;  
console.log(quadrat(2));`

③ `let add=(x,y)=>{  
 return x+y  
}  
  
console.log(add(3,4));`

# JAVASCRIPT STRICT MODE

- Strict mode leads to changes in JavaScript execution semantics
  - Avoidance of "silent" errors
  - Sometimes runs better than normal code
  - Prohibits keywords that will (most likely) come in newer versions

```
// File level strict mode syntax
'use strict';

...
function strict() {
 // Function-level strict mode syntax
 'use strict';
 return "Hi! I'm a strict mode function!" ;
}
```

# JAVASCRIPT STRICT MODE

- “Accidents” throw error; variables must be defined

```
"use strict";
// Assuming a global variable mistypedVariable exists
mistypedVaraible = 17; // this line throws a ReferenceError due to the misspelling of variable
```

- Errors related to objects

```
"use strict";

// Assignment to a non-writable property
let obj1 = {};
Object.defineProperty(obj1, "x", { value: 42, writable: false });
obj1.x = 9; // throws a TypeError
```

```
// Assignment to a getter-only property
let obj2 = { get x() { return 17; } };
obj2.x = 5; // throws a TypeError
```

```
// Assignment to a new property on
//a non-extensible object
let fixed = {};
Object.preventExtensions(fixed);
fixed.newProp = "ohai"; // throws a TypeError
```

# DEBUGGING

- JavaScript is a loosely typed and interpreted language
  - Many errors only occur at runtime
- Debugging tool extremely important
- Google Chrome already has many tools included
  - Google Chrome Developer Tools
  - Activation via F12
  - Offers possibilities to examine HTML, CSS and JS more closely
  - Console for error messages
- **console.log is your friend!!!**

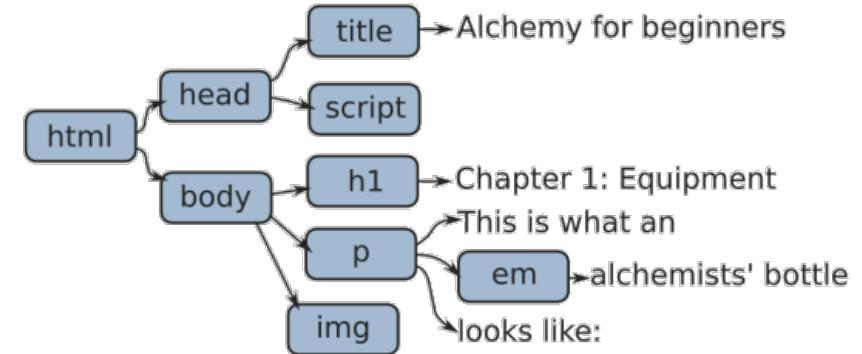


# DOCUMENT OBJECT MODEL (DOM)

Web Design und Programmierung

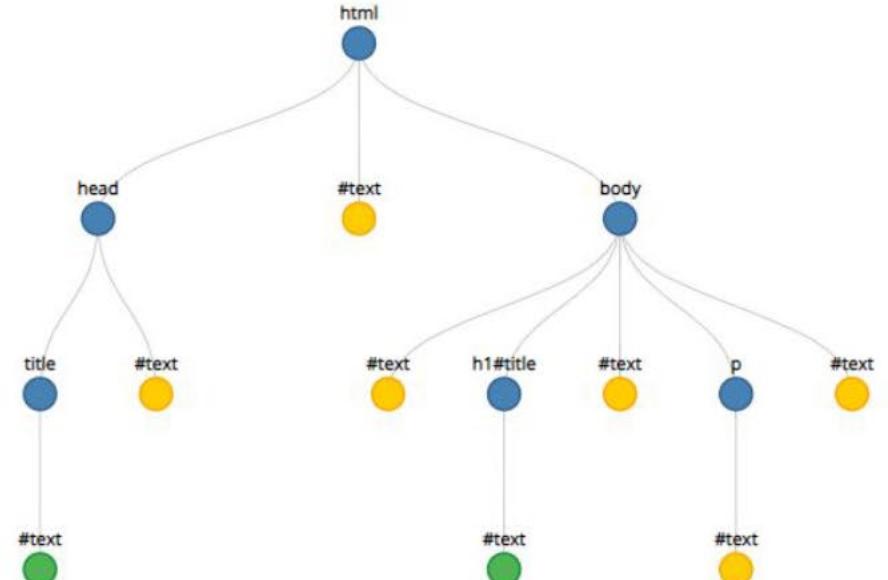
# BASICS

- HTML document should be editable via JavaScript
- Development of the **Document Object Model (DOM)**
  - Specification of an interface for accessing HTML documents
  - Presentation of elements as a tree structure
  - **Element nodes and relationships**
    - Root node
    - children, parent, siblings
    - HTML elements and attributes are mapped as objects (nodes) with corresponding attributes in JavaScript.
  - In particular, DOM allows
    - searching for and navigate between the individual nodes of a document
    - creating, moving and deleting nodes
    - reading, changing and deleting text contents
    - event handling

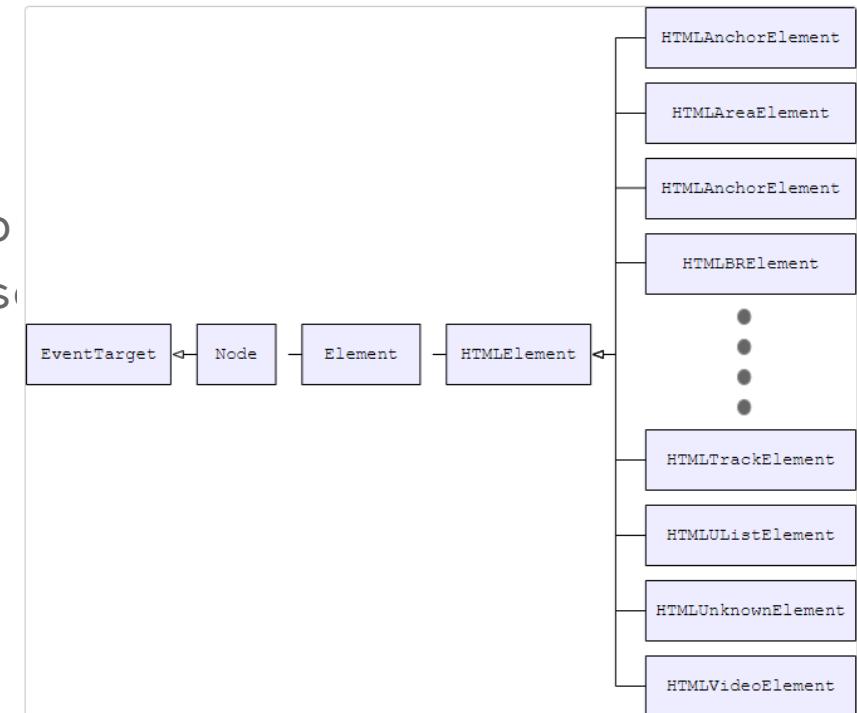


# BASICS

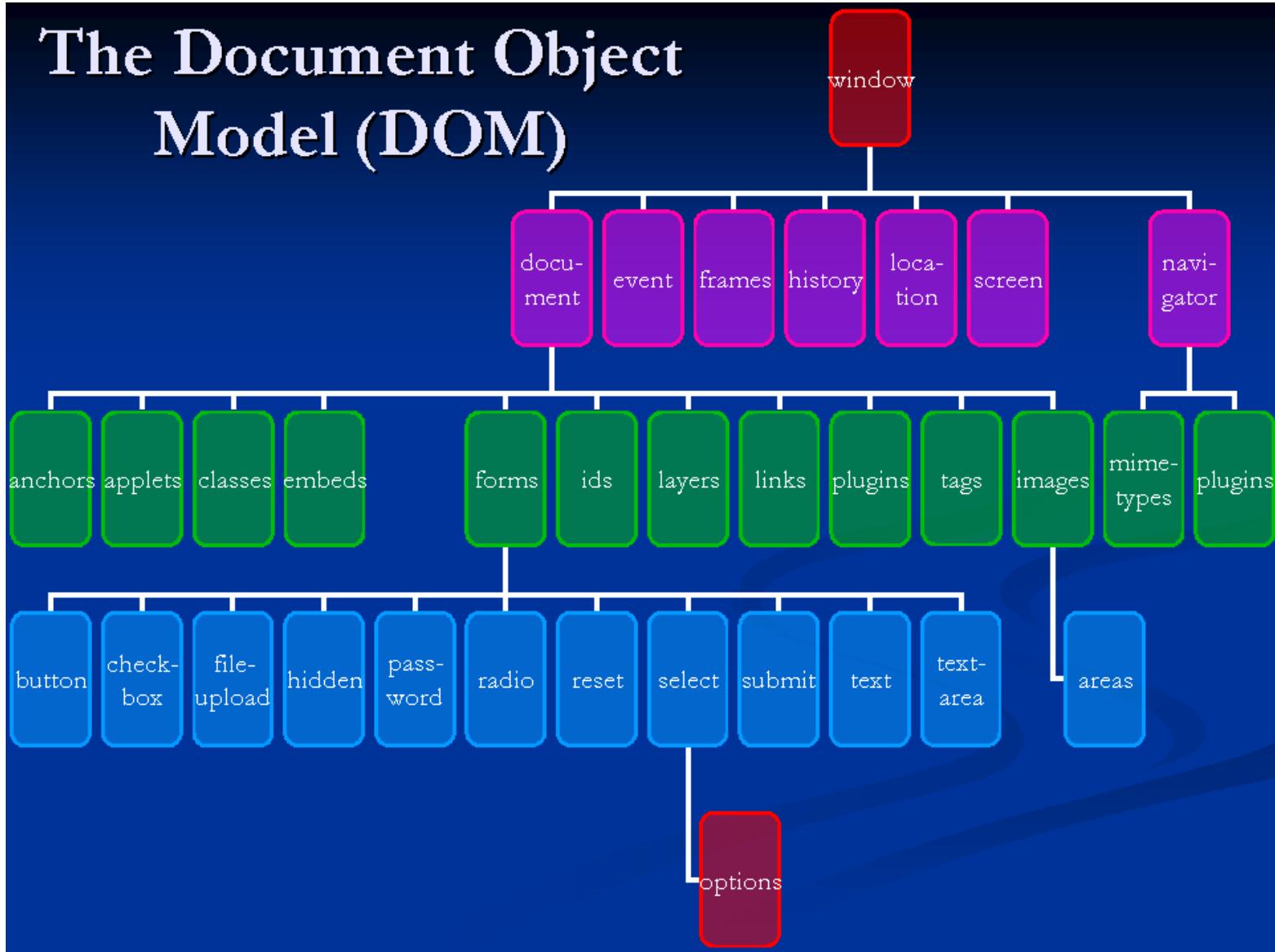
```
<html>
 <head>
 <title>WDP 3</title>
 </head>
 <body>
 <h1 id="title">Hello WDP3</h1>
 <p>rem ipsum dolor sit amet,
 consectetur adipiscing elit.</p>
 </body>
</html>
```



- Basic Data Types
  - Node, Document, Element, NodeList, Attr
- Every object located within a document is a node or document, an object can be an element node but also a node.

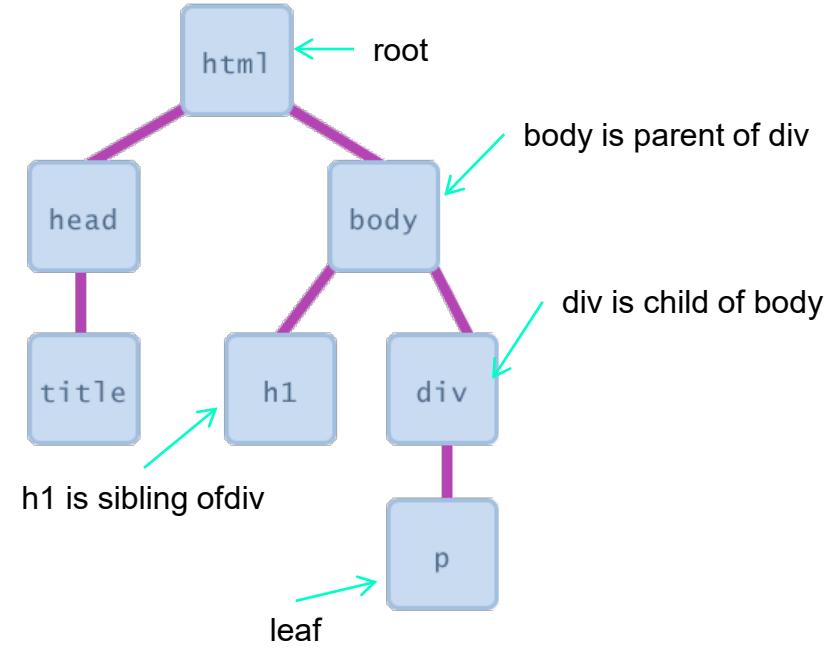


# BASICS



# BASICS

- Objects are created from HTML document
  - window, document, elements
- API methods can be used to access content, structure and presentation.
- Most important node types
  - **Document node** represents the entire tree structure
  - **Element node** corresponds to an exact element in HTML
  - An **attribute node** corresponds exactly to an attribute in HTML
  - **Text node** represents the textual content of an element or attribute
- <http://de.selfhtml.org/javascript/objekte/htmllemente.htm>

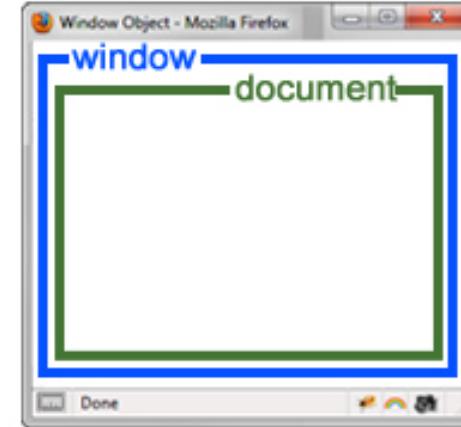


# BASICS

- Methods for retrieving, modifying, ... elements (DOM manipulation)
  - **Searching/selecting** elements
    - Selection by ID, by type of tag
    - Nested operations possible (cf. CSS)
  - Set and read **properties** of elements
    - E.g., changes of CSS properties
  - **Create** and attach nodes
    - Dynamic creation of HTML elements to change DOM
  - **Move** or **delete** nodes
    - Deleting elements from HTML document

# IMPORTANT OBJECTS OF DOM - WINDOW-OBJECT

- **window** is the uppermost object of the object family
- Global object (always available)
  - We have already used it implicitly
  - global **this** is always window object
- Properties
  - name, innerHeight, innerWidth, status, toolbar, ...
- Known functions
  - **alert()** - Outputs text in a dialogue box
  - **confirm()** - Displays a dialogue window with two buttons for "OK" and "Cancel"
  - **open()** - Opens a new window, requires at least 2 parameters
    - URI = target address of a file to be loaded into the new window
    - window name = target
  - **prompt()** , blur(), close(), ...



# IMPORTANT OBJECTS OF DOM - DOCUMENT-OBJECT

- **document** Object is the root element of the HTML document
  - Content displayed in a browser window
  - Child of window object
- HTML elements are direct or indirect children of document object
- Access to nodes
  - `document.documentElement` points to `<html>`.
  - **document.body** points to `<body>`
- Access to arrays of page elements
  - `document.forms`: access to all form elements (forms)
  - `document.images` allows access to all `img` elements (graphics)
  - `document.links` allows access to all `a` elements that have an `href` attribute (hyperlinks)
  - `document.anchors` allows access to `a`-elements that have a `name` attribute (link anchors)
- Navigation via named objects possible
  - Problem: often no names available or quickly very deep nesting

# SELECTING ELEMENTS

**Attention** -> EXCEPT getElementById and querySelector all others deliver a list (array)! Loop necessary for processing

Method	Meaning	Example
getElementById(id)	selects HTML element with certain id attribute	document.getElementById(„alles“);
getElementsByName(name)	selects HTML elements with certain name attribute	document.getElementsByName(„Ball“);
getElementsByTagName(tag)	selects HTML elements with certain tag type	document.getElementsByTagName(„img“);
getElementsByClassName(CSS-Klasse)	selects HTML elements with certain CSS class	document.getElementsByClassName(„cl“)
querySelector(CSS-Selector)	First element that matches selektor	document.querySelector(„.myclass“)
querySelectorAll(CSS-Selector)	All Elements that match the selector	document.querySelectorAll(„.myclass“)

## getElementById:

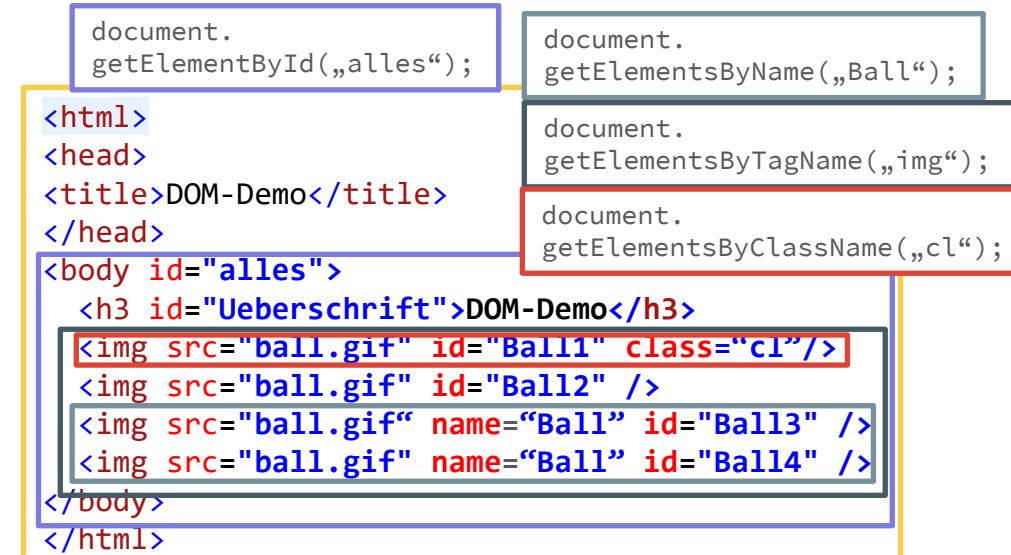
Reference to element whose identifier is id  
 Null if does not exist  
 Also returns hidden element and dynamically created element

## getElementsByName, getElementsByTagName, getElementByClassName

Provide HTMLCollection with elements that have a specific name attribute, are of a specific element type, or have a specific specific CSS class

## querySelectorAll

Returns NodeList that matches selector



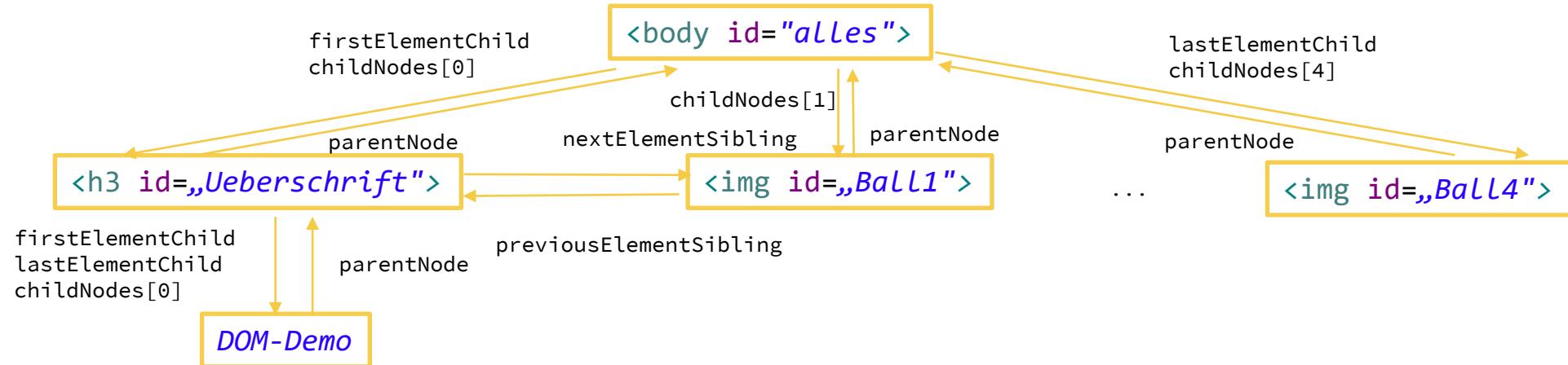
# SELECTING ELEMENTS

- `querySelector` allows use of CSS-selectors
  - Allows very flexible selection
  - Supports CSS 3.0
  - `document.querySelector` searches entire document
  - `element.querySelector` restricts search to elements that are children of `element`

# NAVIGATION IN DOM TREE

- Navigation along two axes
  - Father/child (parent/child):** reproduces HTML nesting
  - Sibling:** HTML elements on the same nesting level

Method	Description
firstElementChild	First child element (first element in the children collection)
lastElementChild	Last child element (last element in the children collection)
nextElementSibling	The next child element of the parent node
previousElementSibling	The previous child element of the parent node
parentNode	Father node
childNodes	Access to collection with <b>child nodes</b> (elements and text nodes)
children	Access to the collection with <b>child elements</b>



# NODES VS ELEMENTS

- **HTMLCollection**
  - List of **HTML elements (tags)**
  - Standard case, which is mostly desired
- **NodeList**
  - List of **HTML nodes** (e.g., also text nodes, returns, comments, ...)
- Array-like behaviour (e.g., length property)
- **Live vs Non-Live Objects**
  - **HTMLCollection** normally **LIVE**
    - Changes in the DOM tree affect the collection
    - getElementsBy... - methods return live object
    - Caution with loops
    - Property children
  - **NodeList** normally **NOT LIVE**
    - querySelectorAll - NodeCollection
    - Exception: Method childNodes returns LIVE NodeList
- Video: <https://www.youtube.com/watch?v=ubNP6fbT2Ac>

NODES	ELEMENTS
firstChild	firstElementChild
lastChild	lastElementChild
nextSibling	nextElementSibling
previousSibling	previousElementSibling

# CREATE NEW ELEMENTS

Methods	Syntax	Description
createElement	document.createElement(html-Tag)	Creates a node from the HTML element passed as parameter
createTextNode	document.createTextNode(„Text“)	Creates a new text node in the tree with the text given as parameter
createAttribute (attName)	document.createAttribute („html-Attribut“)	Creates a corresponding attribute; must be assigned to a node via setAttributeNode(node)

```
let div = document.createElement("div");
```

# INSERT/DELETE ELEMENTS

- **node.appendChild(child) or  
node.append(childOrDOMString)**
  - Inserts the node child as the last child element of the node node
- **node.prepend(childOrDOMString)**
  - Inserts the node child as the first child element of the node node
- **node.insertBefore(newChild, child)**
  - Inserts a child node (newChild) before another child node (child) within a node.
- **node.after(childOrDOMString)**
  - Inserts node child after node as sibling node
- **node.before(childOrDOMString)**
  - Inserts node child before node as sibling node
- **node.removeChild(child)**
  - Deletes the child node from the node node
- **node.replaceChild(newChild, oldChild)**
  - Replaces within node the child node oldChild with newChild

```
let a = document.createElement("a");
a.href = "www.orf.at";
let text = document.createTextNode(
"Link zu ORF");
a.appendChild(text);
parent.appendChild(a);
```

```
let div =
document.createElement("div");
div.append("Some text");
div.prepend("Headline: ");
console.log(div.textContent);
// "Headline: Some text"
```

# INSERT ELEMENTS

- Example for insertBefore:

- Within elements with the class col, a link with the text "Click here" to the page test.html in the parent folder is to be inserted after each H3.

```
<body>
<div id="wrapper">
 <div class="content">
 <ul id="menu">
 Home
 Products

 </div>
 <div id="main">
 <h3>Vivamus nec</h3>
 <div class="col">
 <h3>Vivamus nec</h3>
 <p>Lorem ipsum ... </p>
 <p>Lorem ipsum ... </p>
 </div>
 <div class="col">
 <h3>Vivamus nec</h3>
 <p>Lorem ipsum ...</p>
 <p>Lorem ipsum ...</p>
 </div>
 </div>
 <div id="footer"><p>Copyright</p></div>
</div>
</body>
```

```
let cols = document.querySelectorAll(".col");
for(let c of cols){
 let afterH3 =
 c.querySelector("h3").nextElementSibling;
 let l = document.createElement("a");
 let t = document.createTextNode("Hier Klicken");
 l.appendChild(t);
 l.href="../test.html";
 c.insertBefore(l,afterH3);
}
```

# INSERT ELEMENTS

- element.insertAdjacentElement (pos, element) or  
element.insertAdjacentHTML (pos, htmlstring) or  
element.insertAdjacentText (pos, text)
- **Position-Parameter:**
  - 'beforebegin': insert before element
  - 'afterbegin': insert as first child
  - 'beforeend': insert as last child
  - 'afterend': insert after element

Hallo

Erster Absatz

Some Text

Another Text

Letzter Absatz

Nach dem Container

```
<!DOCTYPE html>
<html lang="en">
 ><head>...</head>
 ><body data-new-gr-c-s-check-1>
 ><h1>Hallo</h1>
 ><div id="container">
 ><p>Erster Absatz</p>
 ><p>Some Text</p>
 ><p>Another Text</p>
 ><p>Letzter Absatz</p>
 </div>
 ><p>Nach dem Container</p>
 ><script>...</script>
 ><script>...</script>
```

```
<!-- beforebegin -->
<p>
 <!-- afterbegin -->
 foo
 <!-- beforeend -->
</p>
<!-- afterend -->
```

```
<div id="container">
 <p>Some Text</p>
 <p>Another Text</p>
</div>
<script>
 let div =
document.querySelector("#container");
 let e1 = document.createElement("h1");
 e1.append("Hallo");
 div.insertAdjacentElement('beforebegin',e1);
 div.insertAdjacentHTML
 ("afterbegin","<p>Erster Absatz</p>");
 div.insertAdjacentHTML
 ("beforeend", "<p>Letzter Absatz</p>");
 div.insertAdjacentHTML
 ("afterend","<p>Nach dem Container</p>");
</script>
```

# ATTRIBUTES

- `object.getAttribute(attribute)` or `object.name_of_attribute`
  - Reads out the attribute with the corresponding name
- `object.setAttribute(attribute, value)` or `object.name_of_attribute = value`
  - Adds a new attribute with a specific value to an object or overwrites the value if the attribute already exists.

The first heading of type H1 is to be displayed right-justified.

Easier

```
let ausrichtung = document.createAttribute("align");
ausrichtung.nodeValue = "right";
let element = document.querySelector("h1");
element.setAttributeNode(ausrichtung);

element.align = "right"

element.setAttribute("align","right");
```

# ATTRIBUTES VS. PROPERTIES

- Normally for HTML attribute corresponding property on DOM element.
- Special cases: CSS classes, innerHTML, innerText,... where no direct HTML attribute is available.
- **Special case CSS classes**
  - Property is called `className` (not `class`)
  - Adding a CSS class
    - `elem.className += " additional_class"`
  - Deletion problematic (as CSS classes are only strings)
  - Attribute `classList` (preferred variant)
    - Adding a CSS class
      - `elem.classList.add("additional_class")`
    - Deleting a CSS class
      - `elem.classList.remove("remove_class")`
- CSS can be changed via the `style` attribute (inline CSS)
  - Availability of all CSS attributes

# DOM - SUMMARY

- Hierarchical HTML data model is represented in DOM as a tree

```
<body id="alles">
 <h3 id="Ueberschrift">DOM-Demo</h3>
 <h3>Hello</h3>

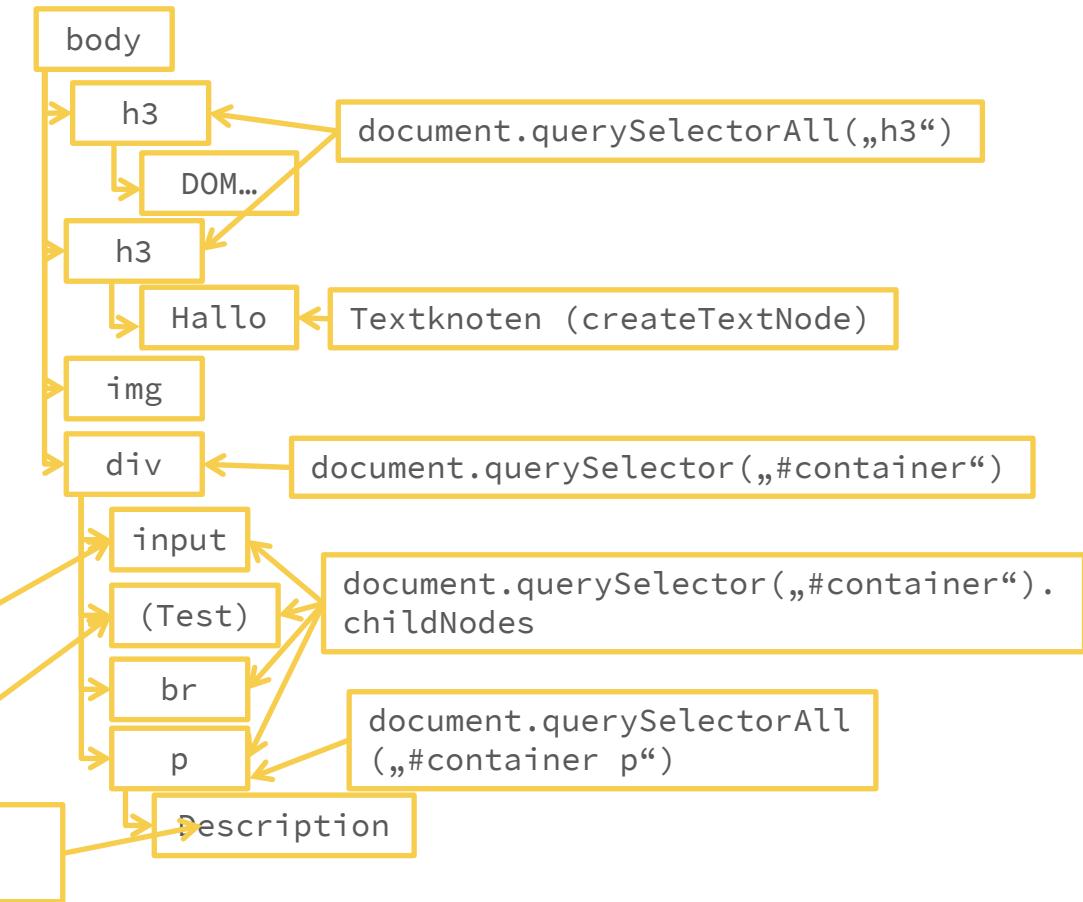
 <div id="container">
 <input
 type="checkbox"></checkbox>
 (Test)

 <p>Description</p>
 </div>
</body>
```

```
document.querySelector("#container").
firstElementChild
```

```
document.querySelector("#container").
childNodes[1]
```

```
document.getElementById("#container").
firstElementChild.nodeValue //get text
```

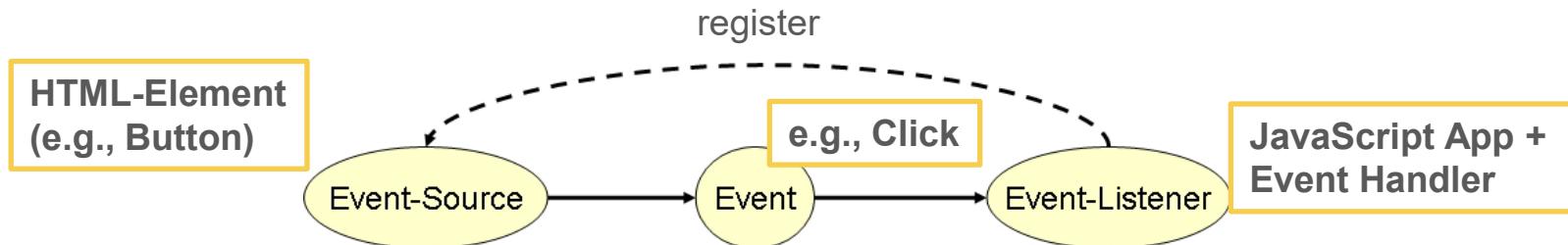


# EVENTS

---

# EVENT HANDLING

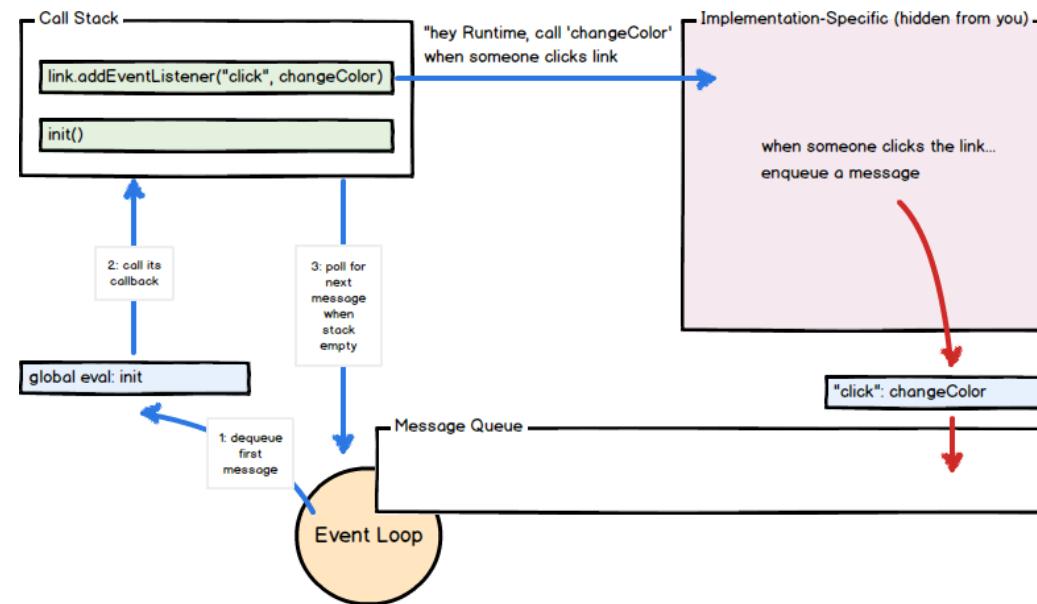
- Reactions to user interactions



- Event-Handler
  - Typically function to be called to react accordingly to event
- Different kinds of events
  - Mouse-Events: click, dblclick, mousedown, mousemove, mouseout, mouseover, mouseup
  - Keyboard-Events: keydown, keyup, keypress
  - Form-Events: change, focus, reset, select, submit
  - General events: load, unload, resize, scroll...

# EVENT HANDLING - EVENT-LOOP

- Events are asynchronous – event handler is called by runtime environment
- JavaScript runtime environment contains message queue
  - Contains events that are to be processed and the corresponding callback function.
  - Without callback function "nothing" would happen after event
- Endless loop always queries for new events
- In case of a new event, callback is called automatically



# EVENT HANDLING - EVENT-HANDLER

- Event handler
  - JavaScript functions that are to be called when a certain event occurs.
  - Register via "onevent", e.g. onclick, onmouseover, ...
- Code that is to be executed as a result of user interaction
  - Any code possible
  - Dynamic registration of event handlers

## Javascript Event Listener



# EVENT HANDLING

- **Inline Event-Handler**
    - Event-handler is added to HTML-Element (avoid)  

```
Hello2
```
  - **External Event-Handler**
  - Setting event handlers in script blocks
    - Event shows the finished loading of the page (incl. all resources such as images, CSS, ...!)
  - Function as event handler via DOM
- Preferred variant  
(graceful degradation)!
- ```
let link =  
document.querySelector("#link3");  
link.addEventListener("click", ()=>{  
    alert("Hallo 3 !");  
});
```
- ```
//function delegate
window.onload = addEventListener;
//anonymous function
window.onload = () => {
 alert("Finished loading DOM");
 addEventListener();
}
```
- ```
function addEventListener(){  
    let link = document.querySelector("#link3");  
    link.addEventListener("click", ()=>{  
        alert("Hallo 3 !");  
    });  
}
```

EVENT HANDLING - MOUSE-EVENTS

```
<html>
<head>
<title>DOM-Demo</title>
<script type="text/javascript">
//bei nur einer Funktion
window.onload = function(){
    window.status = "";
    addEventListener();
}

function addEventListener(){
    let btn = document.querySelector("#btn1");
    btn.addEventListener("click", ()=>{
        window.status+=[click];
    });
    btn.addEventListener("mousedown", ()=>{
        window.status+=[mousedown];
    });
    btn.addEventListener("mouseup", ()=>{
        window.status+=[mouseup];
    });
    btn.addEventListener("dblclick", ()=>{
        window.status+=[doubleClick];
    });
}
</script>
```

```
</head>
<body id="body" >
<h1>Mausereignisse</h1>
<form>
<input type="button" value="Click me!"
       id="btn1">
</form>
</body>
</html>
```



EVENT HANDLING - KEY-EVENTS

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Demo Key Events</title>
</head>
<body>
    <h1>Keyboard events</h1>
    <form>
        <input type="text" size=10 id="txt1">
    </form>

    <script type="text/javascript">
        window.onload = function (){
            let txt = document.querySelector("#txt1");
            txt.addEventListener("keypress", (e)=>{
                console.log(e);
                console.info("Key pressed: " + e.key);
            });
        }
    </script>
</body>
</html>
```

Parameter e provides details of event

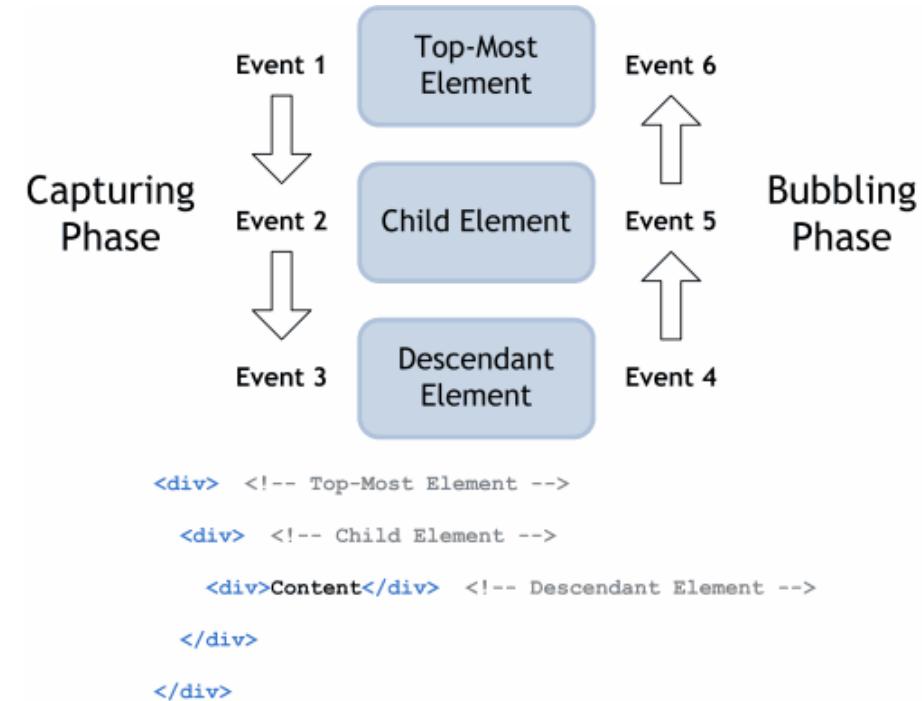
EVENT HANDLING – BEST PRACTICE

- Outsource JavaScript code to external js files
 - Add event handlers dynamically after DOM tree is fully loaded
- No inline event handlers to connect events and event handlers
- Elements that require JavaScript are added dynamically
- Functional page even without JavaScript



EVENT HANDLING - CAPTURE VS BUBBLING PHASE

- When event occurs, it starts at the DOM root.
- From there, the event is forwarded to the place where the event occurred (target).
- At each node that is passed, the event can be handled (**capture phase**)
- The event then travels back through all instances (**bubbling phase**)
- The event can be handled in both phases



EVENT HANDLING - CAPTURE VS BUBBLING PHASE

- Programmer may decide
 - Boolean flag in addEventListener Method
- Forwarding events can be stopped
 - stopPropagation – Method on event
- Prohibit the default behaviour
 - E.g., in case of a link, it would automatically open the according page
 - preventDefault() method prohibits default and allows programmer to handle event with custom code

```
document.body.addEventListener("click", () => {
  alert('Klick am Body in Capture Phase')
},true);

document.body.addEventListener("click", () => {
  alert('Klick am Body in Bubbling Phase')
},false); //Default
```

```
document.body.addEventListener("click", (e) => {
  alert('Klick am Body in Capture Phase');
  e.stopPropagation();
  e.preventDefault();
},true);
```

EVENT HANDLING AND DYNAMIC ELEMENTS

```
<div id="main">
  <h2>Some header</h2>
  <p>First static paragraph</p>
  <p>Second static paragraph</p>
  <button id="btn">Insert</button>
</div>

<script>
  window.onload = ()=>{
    //add event handler to all existing paragraphs
    for(let p of document.querySelectorAll("p")){
      p.onclick = (e)=>{
        console.log("Current target ",e.currentTarget);
        console.log("Target ",e.target);
      }
    }

    document.querySelector("#btn").onclick = (e)=>{
      let p = document.createElement("p");
      p.append("First dynamic paragraph");
      document.querySelector("#btn").before(p);
    }
  }
</script>
```

Some header

First static paragraph

Second static paragraph

After button click

Some header

First static paragraph

Second static paragraph

First dynamic paragraph

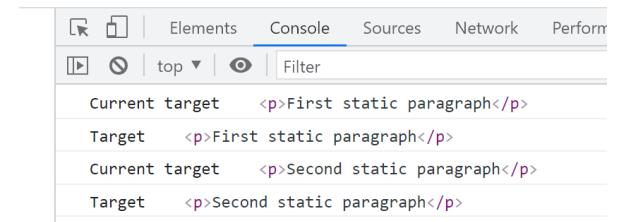
Click on all three paragraphs

Some header

First static paragraph

Second static paragraph

First dynamic paragraph



Dynamic elements do not receive event handlers here!

EVENT HANDLING - DYNAMIC ELEMENTS

```
<div id="main">
  <h2>Some header</h2>
  <p>First static paragraph</p>
  <p>Second static paragraph</p>
  <button id="btn">Insert</button>
</div>

<script>
  window.onload = ()=>{
    //add event handler to all existing paragraphs
    //by using it's parent and target property
    let parent = document.querySelector("#main");
    parent.onclick = (e)=>{
      if(e.target instanceof HTMLElement){
        console.log("Current target
",e.currentTarget);
        console.log("Target ",e.target);
      }
    }
    document.querySelector("#btn").onclick = (e)=>{
      let p = document.createElement("p");
      p.append("First dynamic paragraph");
      document.querySelector("#btn").before(p);
    }
  }
</script>
```

Some header

First static paragraph

Second static paragraph

After button click

Some header

First static paragraph

Second static paragraph

First dynamic paragraph

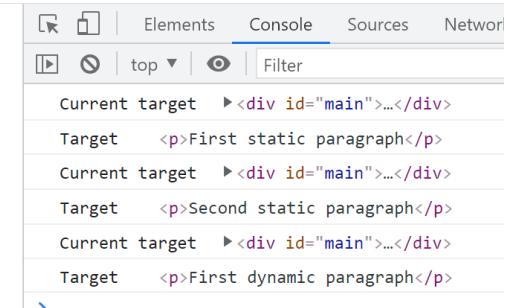
Click on all three paragraphs

Some header

First static paragraph

Second static paragraph

First dynamic paragraph



Dynamic Elements receive Event Handler!

EVENT HANDLING - OVERVIEW ON EVENT HANDLERS

abort (on abort)

blur (on exit of a form element)

change (on completion of the change)

click (on clicking an element)

dblclick (on double clicking an element)

error (in case of an error)

focus (when activating a form field)

keydown (with key pressed)

keypress (with the button held down)

keyup (with the key released)

load (on loading a file)

mousedown (with the mouse button pressed)

mousemove (with the mouse moved on)

mouseout (when leaving the element with the mouse)

mouseover (when moving the mouse over the element)

mouseup (with the mouse button released)

reset (when resetting the form)

select (when selecting text)

submit (when sending the form)

HTML DATA-* ATTRIBUTES

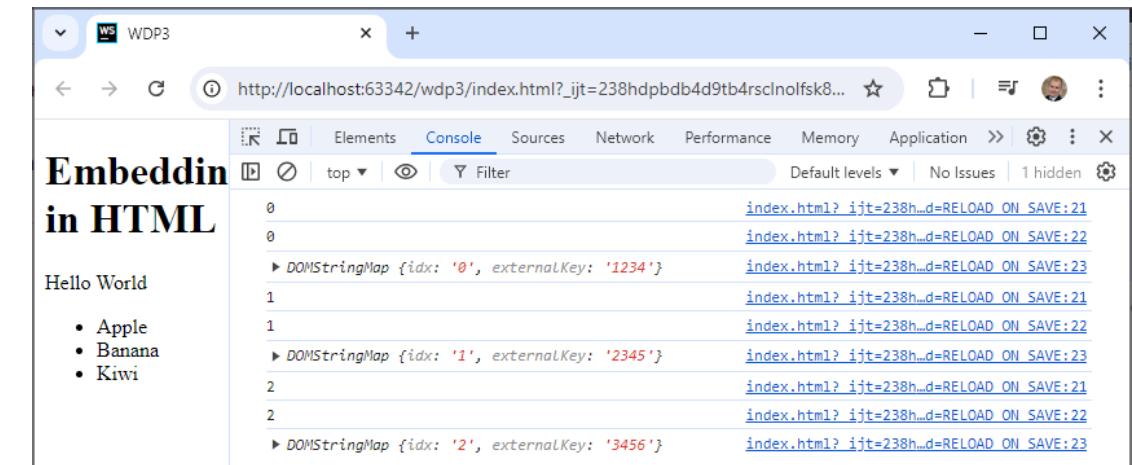


HTML DATA-* ATTRIBUTES

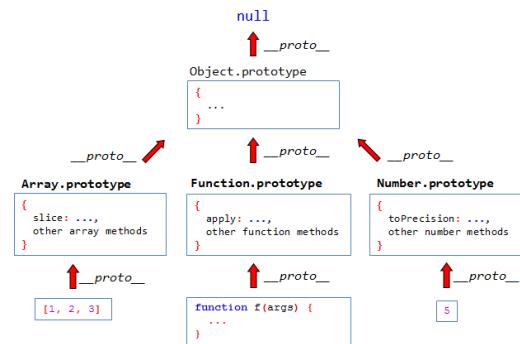
```
<ul>
  <li data-idx="0" data-external-key="1234">Apple</li>
  <li data-idx="1" data-external-key="2345">Banana</li>
  <li data-idx="2" data-external-key="3456">Kiwi</li>
</ul>
```

- HTML5 allows to define custom data attributes
- data attributes allows to store additional information on elements
- data attributes are easy accessible via JavaScript

```
let list = document.querySelectorAll('li');
list.forEach(function(item) {
  console.log(item.getAttribute('data-idx'));
  console.log(item.dataset.idx);
  console.log(item.dataset);
});
```



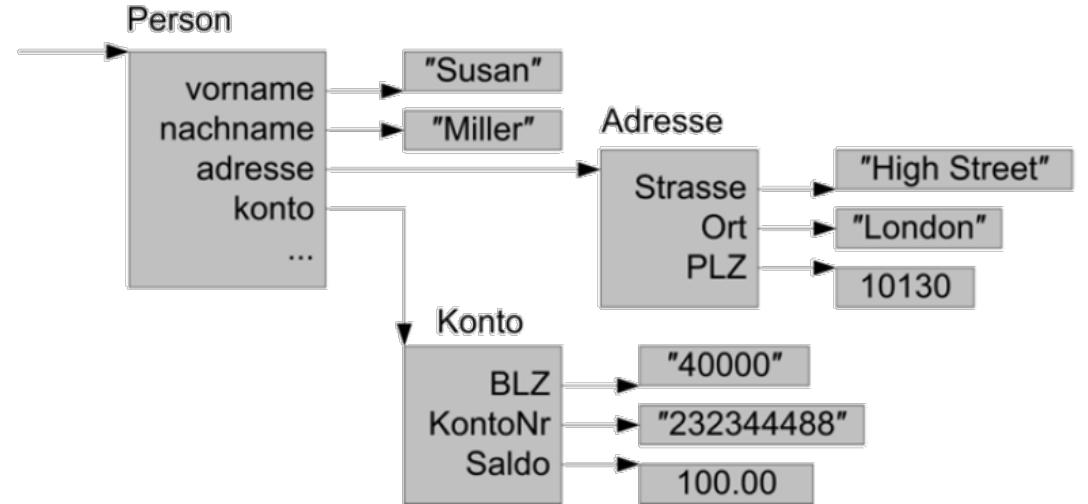
OBJECTS IN JAVASCRIPT



Web Design und Programmierung

OBJECTS IN JAVASCRIPT

- Objects enable structures to be represented as realistically as possible in programs.
- JS is a classless, object-oriented programming language
 - **prototype-based programming**
 - Based on objects (in JS almost everything is an object)
 - Objects are created by cloning already existing objects (prototype); all properties and methods are copied in the process.
 - Object as general prototype from which all other objects are derived
- Classes from ES6 only Syntactic Sugar
- Predefined Objects
 - Objects of the DOM (e.g., `document`, `window`, ...)
 - `Number`, `String`, `Array`, `Math`, `Date`, `Boolean`...



OBJECTS IN JAVASCRIPT

- Objects are named containers and contain

- **Properties (attributes)**

- **Key-value pairs**

- Value either of primitive data type or again objects)

- **Methods**

- To modify the data or to interact with the environment

- Creating variables of the type Object

- Object is predefined and always available

- new Object()

- Keyword **new** → call of so-called Constructor method

```
let myObject = new Object(); //Objekt per Object-Constructor
```

- Erzeugung über Objektliteral (JSON Notation)

```
let myObject = {key1 : „value“, key2 : 10}; //Objekt per Literal
```

- Object.create

```
let object= Object.create(Object.prototype);
```

- Objekte kopieren

```
let object= Object.assign(target,source);
```

OBJECTS PER LITERAL (JSON)

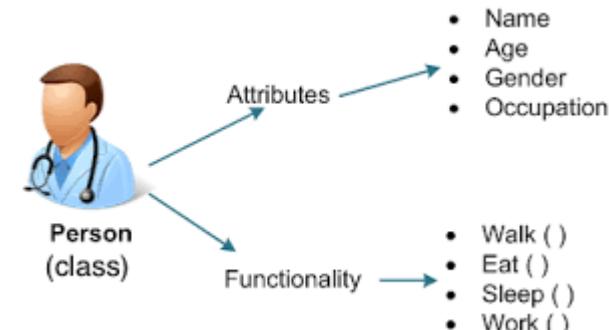
- Properties and methods are created inside
 - Key/value pairs
 - Separation by ":"
- Often used in parameter lists
- All properties are per default public
- Properties can be added/removed during runtime

```
let myObject1 = {  
    //properties are created internally  
    value : 1,  
    myMethod : function(){  
        alert("My attribute value: " + this.value);  
    }  
};  
//call of method  
myObject1.myMethod();
```

Disadvantage: only exactly one instance possible

PROPERTIES AND METHODS

- Objects can be assigned **properties**
 - Dot notation: objectVar.attribute
 - [...] Notation
 - objectVar["value"]
 - "Not allowed" identifiers with dot notation, can be used with bracket notation
 - objectVar["first name"]
 - Undefined, if property not yet defined (read access before write access)
- Objects can define **methods**
 - Bind function to object
 - Keyword **this** denotes current object
 - Access to properties with this.attribute
 - Call with objectVar.method([Params])
- Remove attributes or methods with `delete`



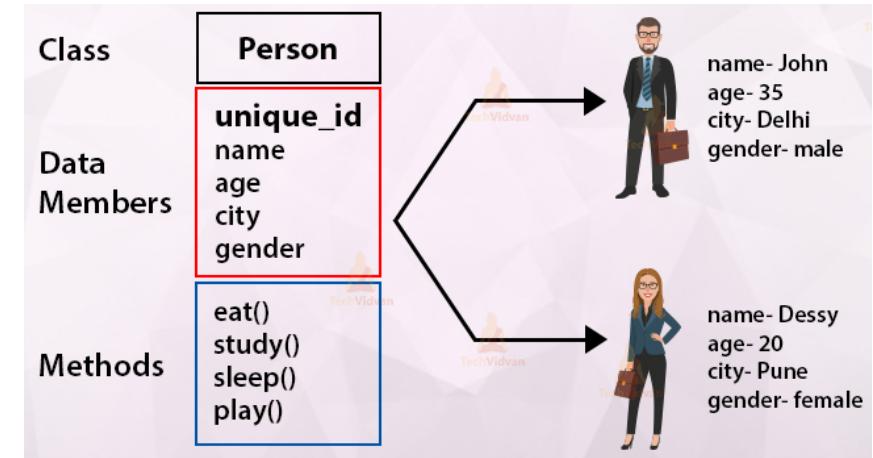
```
let myObject = new Object();
myObject.value = 1;
alert(myObject.value);
```

```
let myObject = new Object();
myObject.value = 1;
myObject.myMethod = function() {
    alert("My attribute value: "
        + this.value);
}
//call method
myObject.myMethod();
//remove Attribute
delete myObject.value;
```

Disadvantage: only exactly one instance possible

CLASSES

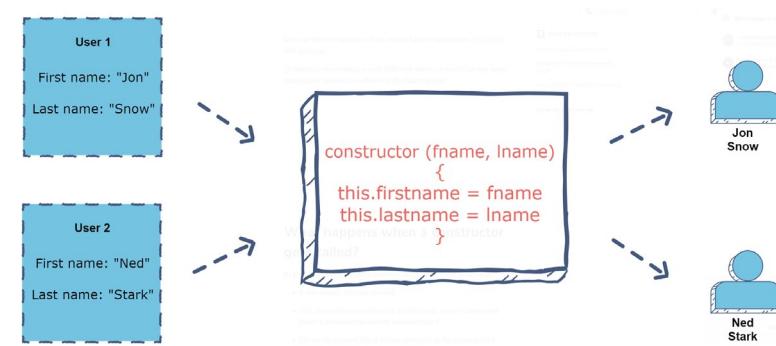
- Only **one instance** can be created via literal or extension of object.
- You need concepts to create "analogue" object instances
 - **Same structure (properties, methods)**
 - **Property values can differ**
- ES 6 introduces class syntax
(but still based on prototypes)
 - (More useful and clearer)
Syntactic Sugar
 - Syntax extension **does not** introduce a new OOP model into the language
- Object orientation based on prototypes
- Classes represent templates for objects
- Any number of instances (objects) possible for a class



CLASSES AND CONSTRUCTOR

- **Class definitions**

- Definition of classes via keyword **class**
- **Constructor:** definition of a function used to create and initialize an object
 - Allocates memory for new object
 - Newly created object can be addressed using **this**
- Access to object properties and methods using **this**
- Constructor functions called when invoking `new` operator
- Returns the object that gets created within it by default
 - No return statement needed
- If you don't specify any constructor, a default constructor is automatically created which has no parameters



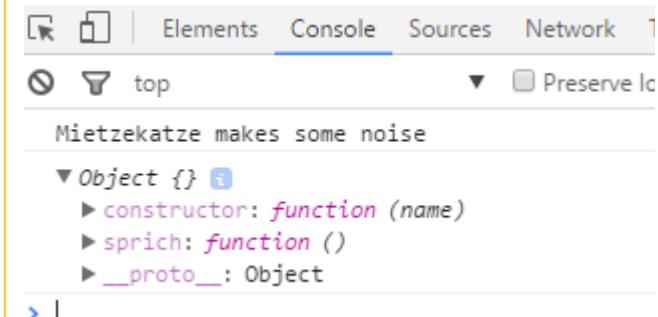
CLASSES AND CONSTRUCTOR

- The constructor method is a special method to create and initialize objects.
 - A class can only have one special method called constructor
 - If there are several constructor methods in a class, a SyntaxError is thrown.
- Shortened notation for methods (no more keyword function necessary)
 - Methods are swapped out into a prototype object

```
class Animal{
  constructor(name) {
    this.name = name;
  }

  talk() {
    console.log(this.name + ' makes some noise');
  }
}

let a1 = new Animal("Mietzekatze");
a1.talk();
console.log(Object.getPrototypeOf(a1));
```



METHODS

- “Normal” methods do not need function keyword
- Keyword **static** defines static methods
 - Static methods are called on a class and not on an object.
 - Cannot be called via a created instance

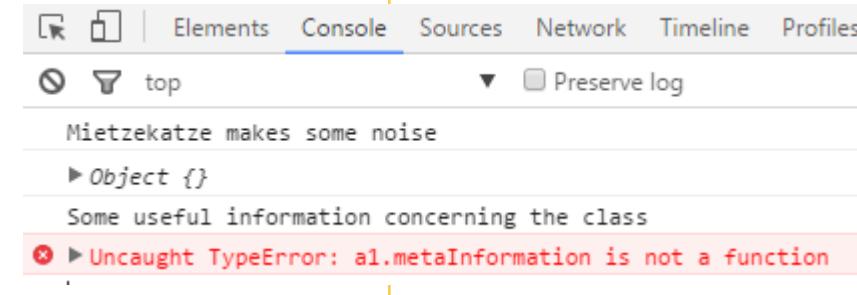
```
class Animal{
    constructor(name) {
        this.name = name;
    }

    talk() {
        console.log(this.name + ' makes some noise');
    }

    static metaInfoInformation() {
        console.log("Some useful information concerning the class");
    }
}

let a1 = new Animal("Mietzekatze");
a1.talk();
console.log(Object.getPrototypeOf(a1));

Animal.metaInfoInformation();
a1.metaInformation();
```



PRIVATE FIELDS AND METHODS

- Visibility modifiers are missing
 - (no private, public, ...)
 - Properties and methods are public
- Private properties simulated
 - Methods created inside a constructor function on the this reference are called privileged public
 - Public: can be called from outside
 - Privileged: have access to private properties
 - Disadvantage: methods exist per object instance

```
class Person {  
    constructor(name) {  
        let _name = name;  
        this.setName = function(name) {  
            _name = name;  
        }  
  
        this.getName = function() {  
            return _name; }  
    }  
  
    let instance = new Person('Hannes');  
    console.log(instance.getName()); //=> 'Hannes'  
    console.log(instance._name); //=> undefined
```

PRIVATE FIELDS AND METHODS

- **Private fields** and methods supported by #

- Can only be accessed in class itself
- Provide getter/setter to gain public access

```
class Person{  
    //private fields  
    #firstName;  
    #lastName;  
  
    constructor(firstName,lastName)  
{  
        this.#firstName = firstName;  
        this.#lastName = lastName  
    }  
  
    //public method  
    print(){  
        this.#internal();  
    }  
}
```

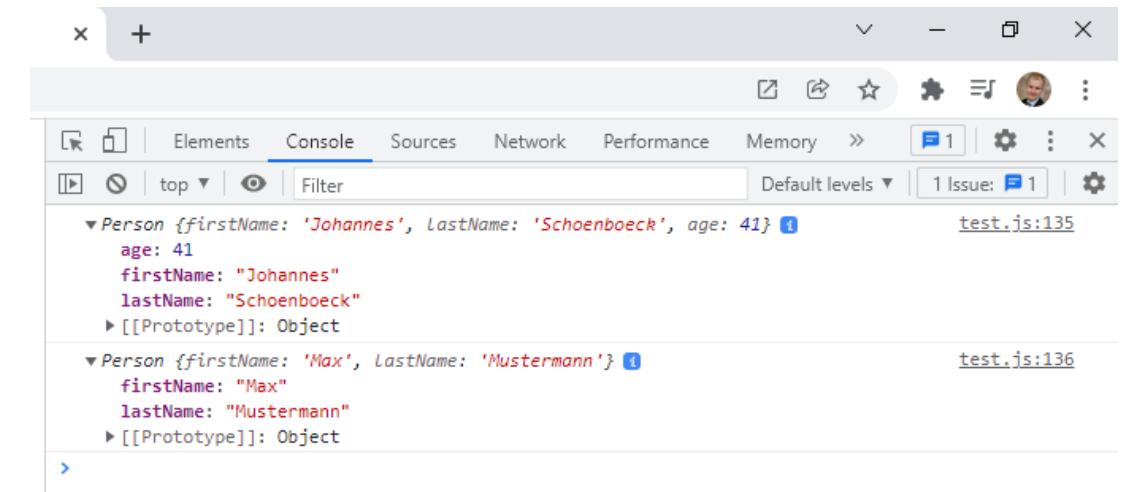
```
get firstname(){  
    return this.#firstName;  
}  
  
set firstname(val){  
    this.#firstName = val;  
}  
  
//private method  
#internal(){  
    console.info("First name: ",this.#firstName);  
    console.info("Last name: ",this.#lastName);  
}
```

```
let p1 = new Person("Johannes","Schoenboeck");  
//testing private fields/methods  
console.log(p1.lastName); //undefined  
//console.log(p1.#lastName); //error  
//p1.internal(); //error  
//public methods  
p1.firstname = "Hannes";  
p1.print();
```

EXTEND OBJECTS

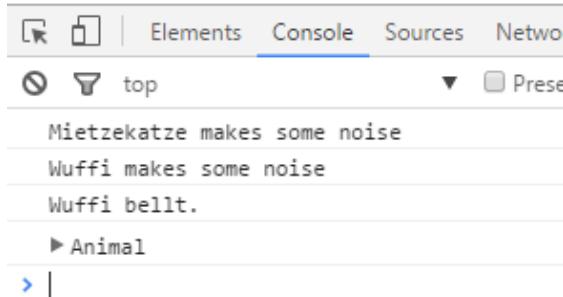
- Objects can be extended
 - Specific to instance
- Extension of the constructor function for attributes
- Extension with new methods via prototyping
- Extension via inheritance

```
class Person{  
    constructor(firstName,lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName  
    }  
  
    let p1 = new Person("Johannes","Schoenboeck");  
    let p2 = new Person("Max","Mustermann");  
    //create additional property - only on p1  
    p1.age = 41;  
    console.log(p1);  
    console.log(p2);
```



INHERITANCE

- Inheritance via key-word **extends**
- Goal: **Reuse**
- Subclass (Dog) extends base class (Animal)
 - Only methods/properties can be added but not taken away
- Methods of the base class can be called with super
- Inheritance only between classes that are semantically related
 - Do not inherit e.g., Person from Animal, although both have a name property



```
class Animal{
    constructor(name) {
        this.name = name;
    }

    talk() {
        console.log(this.name + ' makes some noise');
    }

    static metaInfoInformation() {
        console.log("Some useful information concerning the class");
    }
}

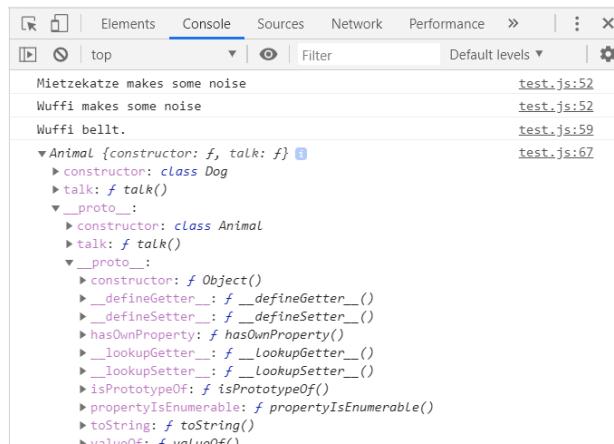
class Dog extends Animal{
    talk() {
        super.talk();
        console.log(this.name + ' bellt.');
    }
}

let a1 = new Animal("Mietzekatze");
a1.talk();

let d1 = new Dog("Wuffi");
d1.talk();
console.log(Object.getPrototypeOf(d1));
```

OVERWRITE AND DYNAMIC BINDING

- JS supports override
 - same method name in classes in an inheritance relationship
- JS supports concept of **dynamic binding**
 - due to dynamic typing default behaviour
- Compared to Java, **lacks**
 - Interfaces
 - Abstract classes
 - Visibility scope „protected“



```
class Animal{
...
  talk() {
    console.log(this.name + ' makes some noise');
  }
}

class Dog extends Animal{
  talk() {
    super.talk();
    console.log(this.name + ' bellt.');
  }
}

let a1 = new Animal("Mietzekatze");
a1.talk();

let d1 = new Dog("Wuffi");
d1.talk();
console.log(Object.getPrototypeOf(d1));
```

CLASSES - HOW THEY WORK AT RUNTIME

- Constructor Function for Objects (ES5)
- constructor only a "special function" -> constructor function in ES 5
- Convention for Constructor Function
 - Start with capital letters
- Calling the constructor function
 - NOT a normal function call
 - Otherwise context global window object
 - No return value, variable would remain empty
 - Call only with new
 - Creates new, empty object
 - Sets this as context object
 - Object x is an instance of the reference type MyObject

```
function MyObject(val){  
    this.value = val;  
}
```

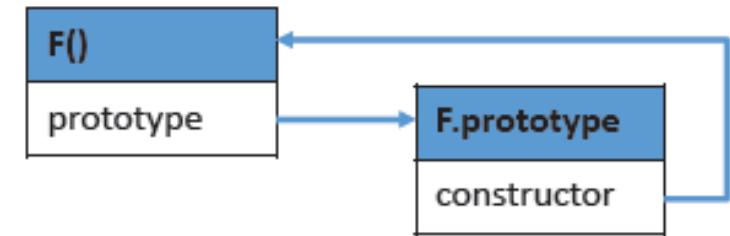
```
function MyObject(val){  
    this.value = val;  
}  
//x wäre leer  
let x = MyObject(17);
```

```
function MyObject(val){  
    this.value = val;  
}  
//erzeugt neues Objekt  
let x = new MyObject(17);
```

PROTOTYPE REFERENCE

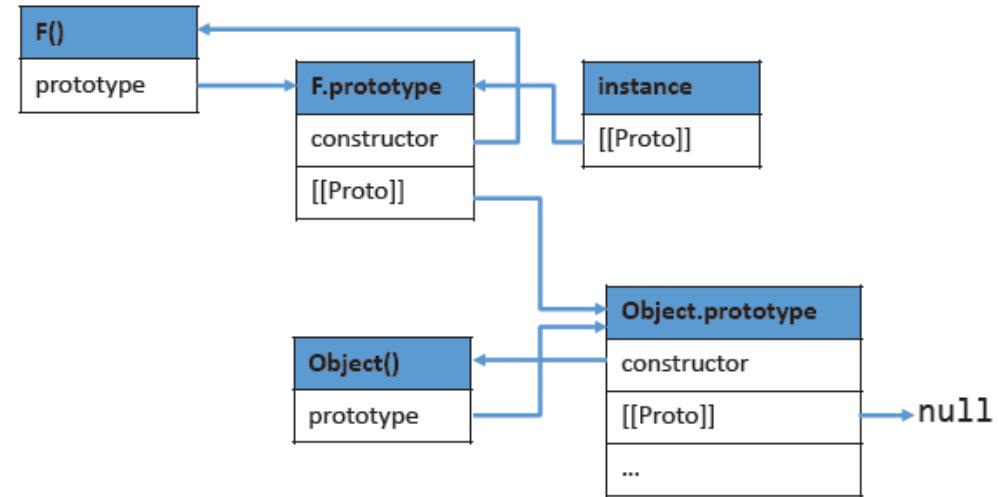
- Each object automatically has a property called prototype
- Points to an object with common
 - **Properties** (rather not useful)
 - **Methods**
- Available to all instances (objects) of a reference type (class)
- **Methods** (and attributes) of the prototype **exist only once for all instances** (of a particular reference type)
- When a constructor function F is created, it receives a property called prototype
- By default, this is an (almost) empty object.
 - Has a property called constructor
 - Points to the function object F

```
function F() { }
```



PROTOTYPE

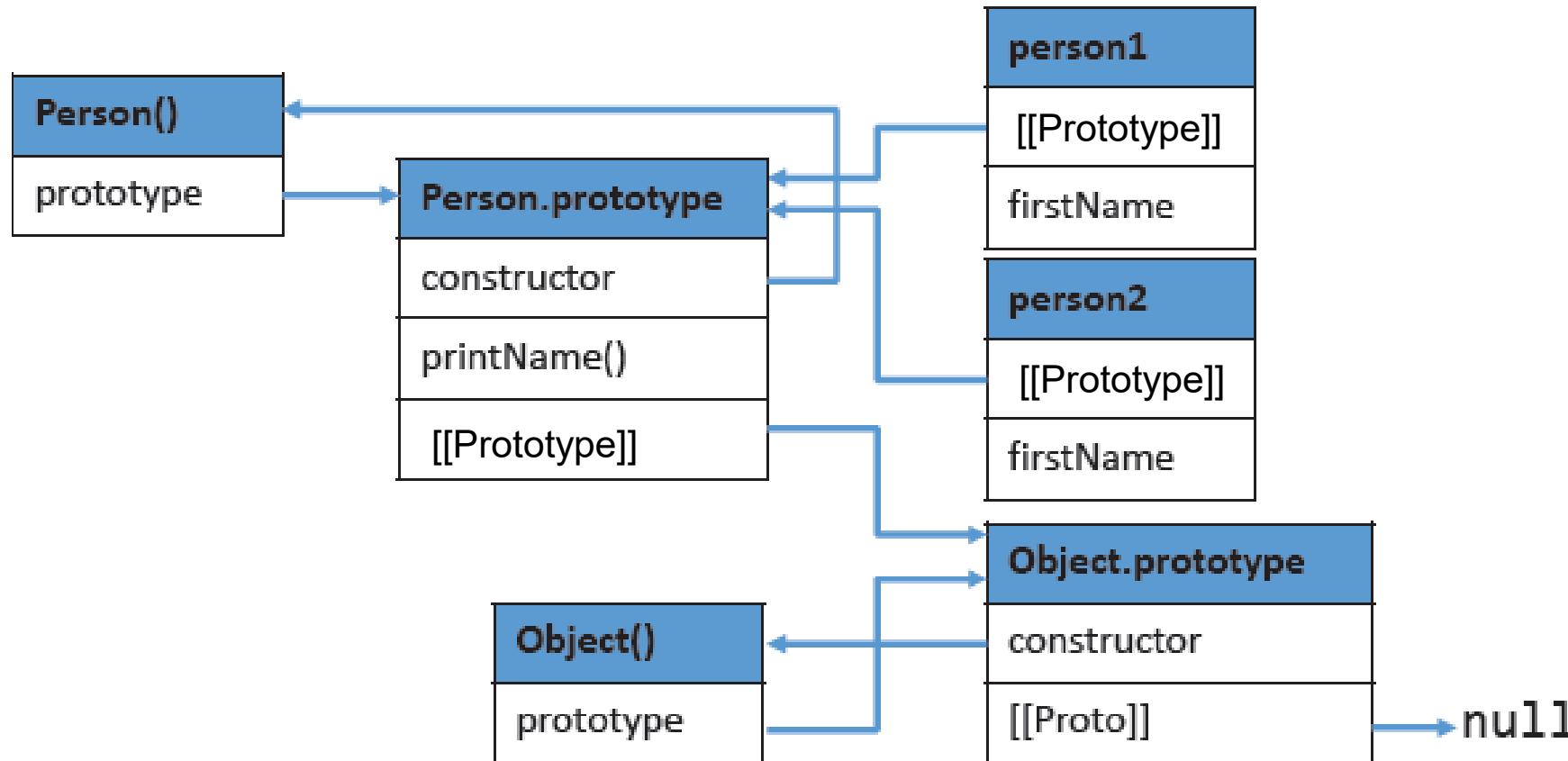
- No copy of the old object-> inefficient
 - only the newly added properties are saved
 - reference from which object the new one has emerged
 - old object is the prototype of the new one
 - Only prototype contains methods – more memory-efficient
- If property of an object is called up
 - the property is first searched for in the object itself
 - or along the prototype chain
- If an object in such a chain is manipulated (e.g. new property), all objects derived from it automatically receive this property
- *Object* as general prototype from which all other objects are derived
 - Provides basic functionalities
 - Property prototype allows the setting of prototypes of an object
 - Important methods: `toString()`, `hasOwnProperty(prop)`, `isPrototypeOf(object)`, ...



PROTOTYPE CHAIN

- Numerous instances

```
function Person(firstName) {  
    this.firstName = firstName;  
}  
Person.prototype.printName = function () {  
    console.log(this.firstName);  
};  
let person1 = new Person("Hugo");  
let person2 = new Person("Tim");
```

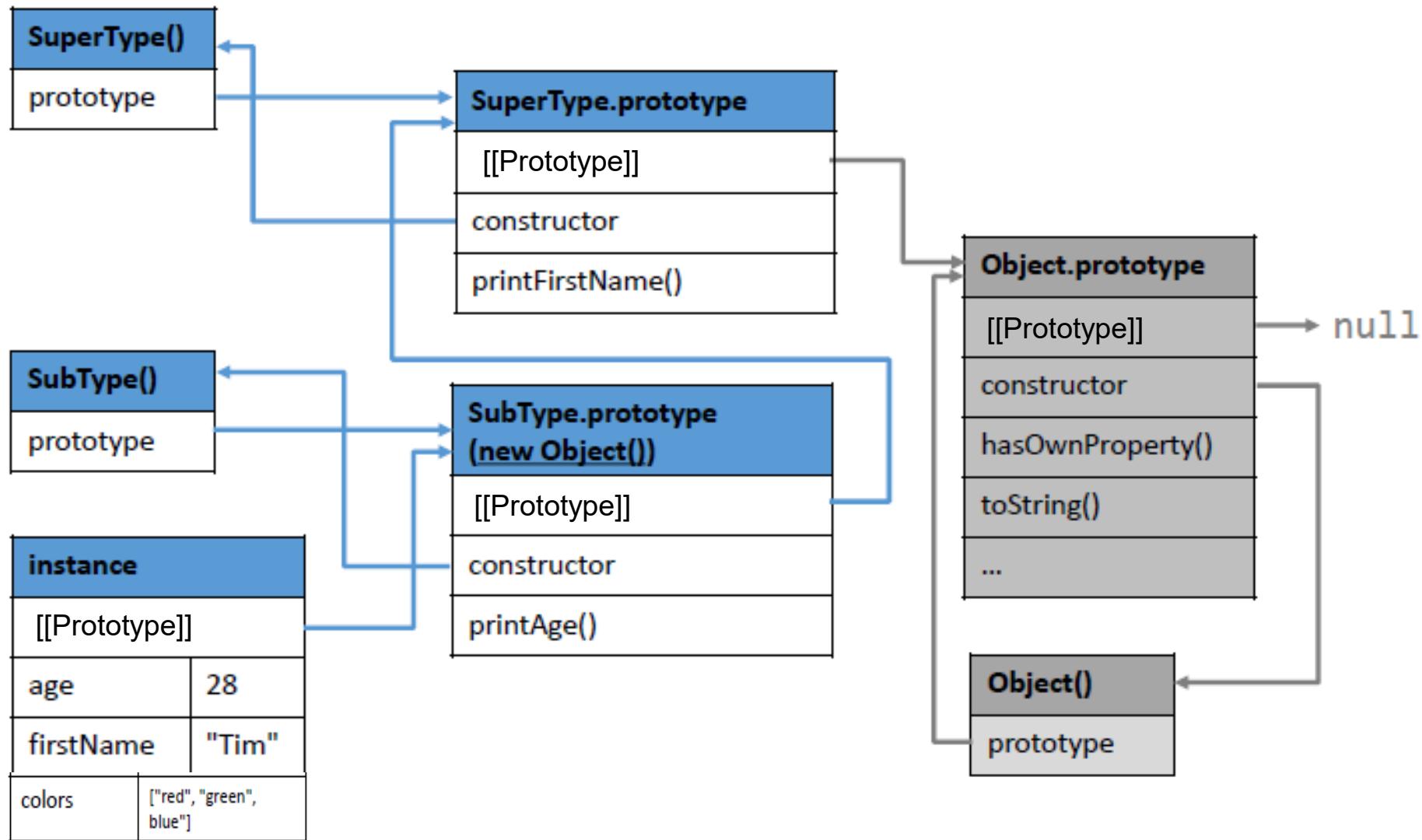


INHERITANCE AND PROTOTYPES

- *Parasitic Combination*
Inheritance –
Internal implementation

```
function SuperType(firstName) {
    this.firstName = firstName;
    this.colors = ["red", "green"];
}
SuperType.prototype.printFirstName = function () {
    console.log(this.firstName);
};
function SubType(firstName, age) {
    SuperType.call(this, firstName);
    this.age = age;
}
SubType.prototype = Object.create(SuperType.prototype);
SubType.prototype.constructor = SubType; // constructor-property
SubType.prototype.printAge = function () {
    console.log(this.age);
};
```

INHERITANCE AND PROTOTYPES - PARASITIC COMBINATION INHERITANCE



OBJECT DESTRUCTURING

- The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.
 - Assignment of attributes to variables - Variable has to have the same name as the property
 - Shortened notation and simpler use

```
//Object destructuring
let student = new Student("Hannes", "Schönböck", 123456);

let {firstName, lastName, matNr} = student;
console.log(firstName);
console.log(lastName);
console.log(matNr);
```

SPREAD OPERATOR

- The spread (...) syntax allows an iterable, such as an array or string, to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected.
- In an object literal, the spread syntax enumerates the properties of an object and adds the key-value pairs to the object being created.
 - ... operator (spread operator) allows to copy objects easily

```
function sum(x, y, z) {  
    return x + y + z;  
}  
  
const numbers = [1, 2, 3];  
  
console.log(sum(...numbers));
```

```
let person = {  
    name:"Hannes",  
    age:41  
}  
  
let person2 = person;  
person.name = "Johannes";  
console.log(person2);  
let person3 = {...person};  
person.name = "Karl";  
console.log(person3);  
console.log(person2)
```



A screenshot of a browser's developer tools console window. The title bar includes icons for file, edit, top, zoom, and filter. Below the title bar, there are three log entries, each preceded by a blue right-pointing arrow:

- ▶ {name: 'Johannes', age: 41}
- ▶ {name: 'Johannes', age: 41}
- ▶ {name: 'Karl', age: 41}

The third entry is partially cut off at the bottom.

NULLISH COALESING OPERATOR - ??

- It is a logical operator, which returns the right-hand side of the operator if the left side is null or undefined.

```
const x = null;
const value = x ?? 'default value';
console.log(value); // Output: 'default value'
```

OPTIONAL CHAINING - ?

- Allow to access object values without null/undefined check and void throwing an error.
- If one part of the access chain is null or undefined, the expression returns undefined

```
const obj = {  
  a: {  
    b: null  
  }  
}  
console.log(obj.a.b.c); // Uncaught TypeError: Cannot read property 'c' of null  
console.log(obj.a.b?.c); // undefined
```

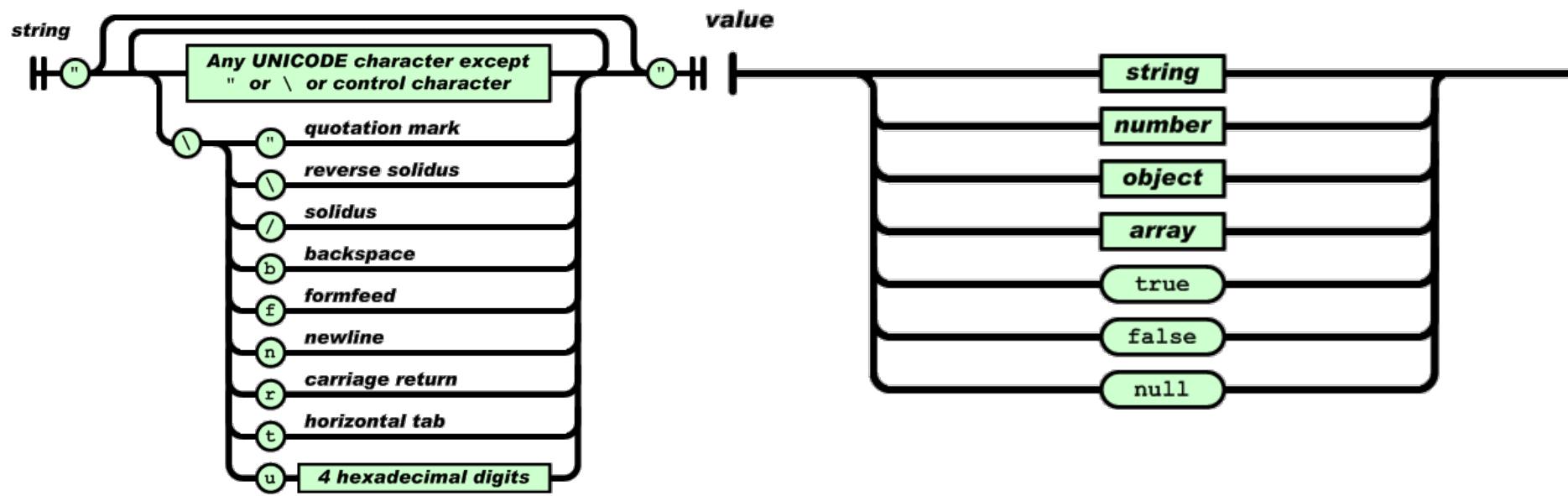
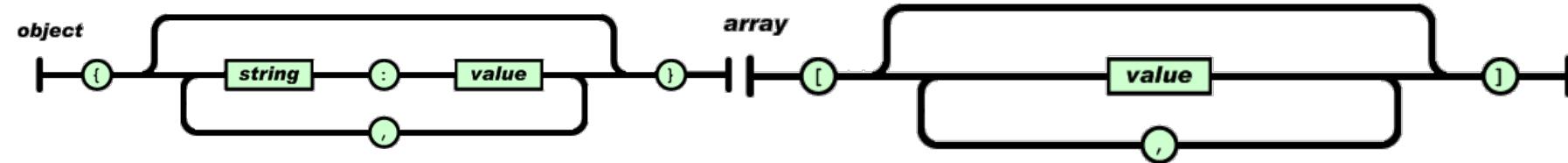
JSON – JAVASCRIPT OBJECT NOTATION



BASICS

- **JSON** (JavaScript Object Notation) is a lightweight data exchange format
- Easy to read, understand and create for both human and machine
- JSON is based on simple structures
 - Objects
 - Key-value pairs
 - Array
- Language independent
 - Supported in many programming languages
 - Heavily used for data exchange (AJAX, REST)
- JSON is a method to serialize JavaScript object structures (objects, arrays, numbers, strings, ...) into a string and vice versa.
- JSON is also an object in JavaScript which provide the capability to serialize (`stringify(val)`) and parse (`parse(val)`).
- Each JSON text is valid JavaScript but not each JS is a valid JSON text!

JSON SYNTAX



JSON SYNTAX - EXAMPLES

Objekt:

```
{"firstName": "John", "lastName": "Doe"}
```

Array mit Objekten:

```
"employees": [  
    {"firstName": "John", "lastName": "Doe"},  
    {"firstName": "Anna", "lastName": "Smith"},  
    {"firstName": "Peter", "lastName": "Jones"}  
]
```

JSON kann mit JS verarbeitet werden:

```
var employees = [  
    {"firstName": "John", "lastName": "Doe"},  
    {"firstName": "Anna", "lastName": "Smith"},  
    {"firstName": "Peter", "lastName": "Jones"}  
];  
  
// returns John Doe  
employees[0]["firstName"] + " " +  
employees[0]["lastName"];
```

```
{  
    "firstName": "John",  
    "lastName": "Smith",  
    "isAlive": true,  
    "age": 25,  
    "height_cm": 167.6,  
    "address": {  
        "streetAddress": "21 2nd Street",  
        "city": "New York",  
        "state": "NY",  
        "postalCode": "10021-3100"  
    },  
    "phoneNumbers": [  
        {  
            "type": "home",  
            "number": "212 555-1234"  
        },  
        {  
            "type": "office",  
            "number": "646 555-4567"  
        }  
    ],  
    "children": [],  
    "spouse": null  
}
```

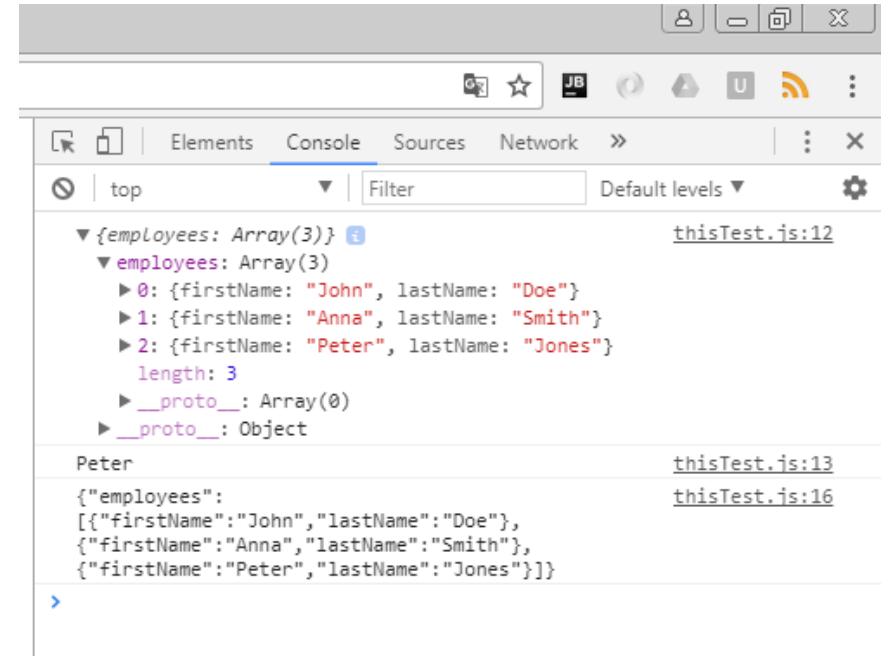
JSON IN JAVASCRIPT

- **JSON.parse()**
 - Converts JSON String into an according object
- **JSON.stringify()**
 - Converts JSON object into an according string

```
let text = '{ "employees" : [ ' +
    '{ "firstName":"John" , "lastName":"Doe" },' +
    '{ "firstName":"Anna" , "lastName":"Smith" },' +
    '{ "firstName":"Peter" , "lastName":"Jones" }
] }';

let obj = JSON.parse(text);
console.log(obj);
console.log(obj.employees[2].firstName);

let jsonString = JSON.stringify(obj);
console.log(jsonString);
```



READ JSON FILE

- Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline
- Loading a file is similar to a HTTP request
 - AJAX call to asynchronously load data
 - Data contains JSON object
- **More on AJAX later**

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

FROM SCOPES TO MODULES

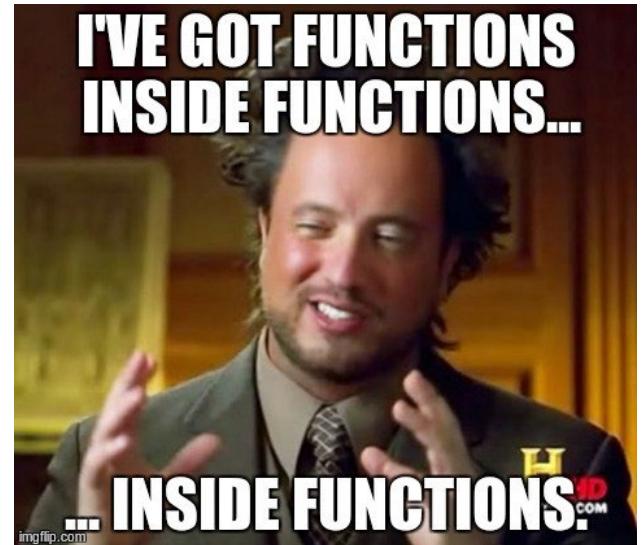
Web Design und Programmierung

SCOPES OF VARIABLES

- Scopes (validity areas) essential in programming languages
- Scopes are determined in JavaScript up to ES6 only by functions
- **Global variables**
 - Variables that are defined outside of a function
 - 3 ways to create them
 - Explicitly with the keyword **let** in the global namespace
 - Implicitly without the keyword **let** in the global namespace
 - Implicitly without the keyword **let** in the "local" namespace
- Problems:
 - Pollution of the global namespace
 - Accidental name conflicts
 - No modularity
- **TIP: Avoid global variables as much as possible!**

INNER FUNCTIONS AND CLOSURES

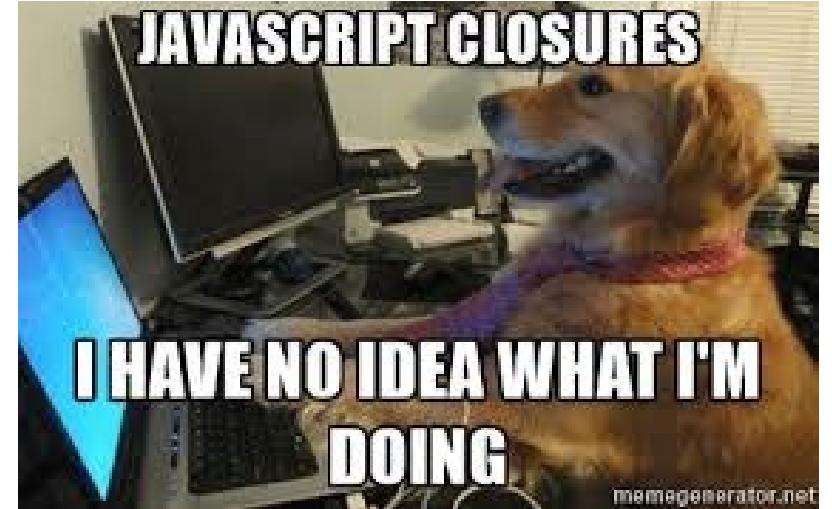
- JavaScript allows functions inside functions (inner functions)
 - What happens here?
 - Return value of create is inner function display
 - This function is then executed
- Surrounding function of `dipslay (create)` was finished before `display()` was called
- Inner function has access to variables from outer function
- Closures: Functions that remember the "environment" in which they were created.



```
function create() {  
  let name = "WDP";  
  
  function display() {  
    console.log(name);  
  }  
  
  return display;  
}  
  
let myFunc = create();  
myFunc();
```

EXECUTION CONTEXT

- JavaScript Engine runs in two-pass process
 - 1st pass (creation stage) - before the actual code execution
 - 1 Declaring function parameters
 - 2 Declaration of local variables and function objects
(without initialisation)
 - 3 Declaration and initialisation of functions
 - 2nd pass (Execution Stage)
 - Code execution



```
function outer(arg1, arg2){  
    let local_var = 'foo';  
    let a_function_obj = function(){  
        console.log("a function");  
    }  
    function inner(){  
        console.log("inner");  
    }  
    outer(1,2)
```

The code illustrates the three phases of the creation stage:

- 1 Declaring function parameters
- 2 Declaration of local variables and function objects (without initialisation)
- 3 Declaration and initialisation of functions

EXECUTION CONTEXT

- With each function call a so-called "Context Execution Object"(CEO) is created, in which the variables are stored in the creation stage
- Is stored in the "execution stack"
- Sequence:
 - For each formal parameter (if function - not for global code)
 - Store property with name and value in the CEO
 - For each function declaration
 - Create function object and store property with name and value ("pointer to function") in the CEO.
 - If a property with the same name already exists in the CEO then replace it (Not for function expressions!)
 - For each variable declaration
 - If a property with the same name already exists in the CEO do nothing
 - otherwise: store property with name (and value = undefined) in CEO

EXECUTION CONTEXT

- When are function objects created?
 - **Function declarations**
 - During Creation Stage of the "enclosing" function
 - = before execution of the "enclosing" function
 - **Function expressions**
 - During Code Execution Stage
 - = during the execution of the code (of the "enclosing" function)
- Function objects "remember" the CEO of the enclosing function when they are created.
 - = the CEO that was "current" when the function object was created -> Lexical Scope

```
function test() {  
    console.log("hello");  
}
```

```
...  
let myFunc = function () {  
    console.log("hello");  
};  
...
```

EXECUTION CONTEXT

- The scope chain during the execution of a function f consists of:
 - CEO of call of f
 - by calling the function f (in Creation Stage)
 - the CEO that "remembered" $f =$ the CEO that was "current" when f was created = CEO of f Parent (Closure)
 - the CEO that "remembered" f Parent
(function that created f)
 - CEO that was current when function f Parent was created (Closure of the Parent function)
 - etc. up to the Global CEO
- – Note: Global Scope == Global Object == Global CEO == window (in the browser)
- If a variable is accessed (read/write), the scope chain is searched.
 - Starting with "local scope" (CEO) up to global scope (CEO)
- If variable is not found in scope chain, then
 - Reading: ReferenceError
 - Writing: created in global scope
 - If Strict-Mode -> Error

EXECUTION CONTEXT

```
let welcome = "KWM";
function showMessage(msg) {
    2 let temp = msg;
    alert(temp);
}
1 showMessage(welcome);
```

Global Scope
Local scope

	Call Stack	Creation Stage	Execution Stage
1	Global Scope	{ Global Object: window this: window showMessage: Ref to Fkt-Obj welcome:undefined }	{ Global Object: window this: window showMessage: Ref to Fkt-Obj welcome: "KWM" }
2	showMessage Global Scope	{ ... showMessage: Ref to Fkt-Obj welcome: "KWM" { msg: „KWM“ temp: undefined } }	{ ... showMessage: Ref to Fkt-Obj welcome: "KWM" { msg: „KWM“ temp: „KWM“ } }

EXECUTION CONTEXT

Global Scope
Local scope

```
function firstFunction() {
  let x = 10;
  secondFunction();
  console.log(x);
}

function secondFunction(){
  let y=5;
  console.log(y);
}
firstFunction();
```

	Call Stack	CEO 1st pass	CEO 2nd pass
	Global Scope	{ ... firstFunction:Ref auf Fkt-Obj secondFunction: Ref auf Fkt-Obj }	{ ... firstFunction:Ref auf Fkt-Obj secondFunction: Ref auf Fkt-Obj }
1	firstFunction Global Scope	{ ... firstFunction:Ref auf firstFunction secondFunction: Ref auf secondFkt { x: undefined } }	{ ... firstFunction:Ref auf firstFunction secondFunction: Ref auf secondFkt { x: 10 } }
2	secondFunction firstFunction Global Scope	{ firstFunction:Ref auf firstFunction secondFunction: Ref auf secondFkt { x: 10 } { y: undefined } }	{ firstFunction:Ref auf firstFunction secondFunction: Ref auf secondFkt { x: 10 } { y: 5 } }

Note: lexical Scope
(x in second function would be undefined)

EXECUTION CONTEXT

Global Scope
Local scope
Closure

```
function firstFunction() {
  var x = 10;
  secondFunction();
  console.log(x); ②
}

function secondFunction(){
  var y=5;
  console.log(y);
  console.log(x); ③
}

firstFunction(); ①
```

	Call Stack	CEO 1. Pass	CEO 2. Pass
	Global Scope	{ " firstFunction:Ref auf Fkt-Obj }	{ " firstFunction:Ref auf Fkt-Obj }
①	firstFunction Global Scope	{ " firstFunction:Ref auf firstFunction { x: undefined secondFunction: Ref auf Fkt-Obj } }	{ " firstFunction:Ref auf firstFunction { x: 10 secondFunction: Ref auf Fkt-Obj } }
②	secondFunction firstFunction Global Scope	{ firstFunction:Ref auf firstFunction { x: 10 secondFunction: Ref auf Fkt-Obj { y= undefined } } }	{ firstFunction:Ref auf firstFunction { x: 10 secondFunction: Ref auf Fkt-Obj { y= 5 } } }

EXECUTION CONTEXT

Global Scope
Local scope
Closure

```
function firstFunction() {
  let x = 10;
  console.log(x); 2

  let secondFunction = function () {
    let y=5;
    console.log(y);
    console.log(x);
  }

  return secondFunction;
}

let f = firstFunction(); 1
f();
```

	Call Stack	CEO 1st pass	CEO 2nd pass
1	Global Scope	{ ... firstFunction:Ref auf Fkt-Obj f: undefined }	{ ... firstFunction:Ref auf Fkt-Obj f: Ref auf Fkt Obj secondFct }
2	firstFunction Global Scope	{ firstFunction:Ref auf firstFunction { x: undefined secondFunction: undefined } }	{ firstFunction:Ref auf firstFunction { x: 10 secondFunction: Ref auf Fkt-Obj } }
3	secondFunction firstFunction Global Scope	{ firstFunction:Ref auf firstFunction { x: 10 secondFunction: Ref auf Fkt-Obj { y= undefined } }	{ firstFunction:Ref auf firstFunction { x: 10 secondFunction: Ref auf Fkt-Obj { y= 5 } }

ENCAPSULATION UNTIL ES5 - IIFE

- IIFE: Immediately Invoked Function Expression
 - Usually anonymous function which is called directly as soon as it is declared
 - Function enclosed in parentheses before it is called
 - Not evaluated as declaration but as expression
- Similar to the following code, but IIFE is only executed exactly once

```
(function() {  
  console.log("IIFE");  
})();
```

```
let iife = (function() {  
  console.log("IIFE");  
});  
iife();
```

IIFE (ES5)

- Isolation from external, global scope
- Basis for module patterns and namespaces
 - used in various frameworks to prevent pollution of the global scope
- IIFE can also return values
 - Often an object is returned as return value
 - Attributes and methods of the return object can be accessed externally
 - Rest is private within the module

```
(function () {
    let private_variable = "private";
})();
```

```
let person = (function () {
    let firstName = 'Tim', a = 20;
    return {
        name: firstName,
        age: a
    };
})();
// this is undefined, no firstName for you.
console.log(person.firstName);
// this outputs 'Tim'
console.log(person.name);
// this outputs '20'
console.log(person.age);
```

MODULES

Make it reusable!

MODULES - BASICS

- Currently code divided into
 - global code
 - Code within functions
- Finer encapsulation needed for reusability and maintainability
- Namespaces
- Data encapsulation through modules
 - Public interface (API)
 - Can be used from outside
 - Services for third parties
 - Private implementation
 - Private variables and implementation
 - No access by third parties from outside

MODULES IN ES6

- ES6 introduces native support
- ES6 modules are stored in files
 - 1 module per file and 1 file per module
- Modules contain (typically) declarations of functions and variables
 - Are local to the module
 - Can be explicitly exported and imported from other modules

	Scripts	Modules
HTML Element	<script>	<script type="module">
Default mode	non-strict	strict
Top-level variables are	global	local to module
Value of this at top level	window	undefined
Executed	synchronously	asynchronously
Declarative imports (import statement)	no	Yes
File extension	.js	.js

MODULES IN ES6

- Each module is executed exactly once at load time
 - Variable declaration, functions, etc...
- Declarations are local to the module
 - Public code must be exported explicitly
 - Named or default exports
 - Public code can be imported from other modules
 - Import/Export must always be specified at the beginning
- Modules are singletons
 - Only 1 instance exists (even if module is imported multiple times)

MODULES IN ES6

- Named exports
 - Several per file possible
 - Keyword export
 - Only exported declarations can be used in other modules

```
//---- lib.js ----  
const sqrt = Math.sqrt;  
  
export function square(x) {  
    return x * x;  
}  
  
export function diag(x, y) {  
    return sqrt(square(x) + square(y));  
}
```

```
//---- main.js ----  
import { square, diag } from './lib.js';  
  
console.log(square(11)); // 121  
console.log(diag(4, 3)); // 5
```

```
<!--index.html-->  
<!DOCTYPE html>  
<html lang="en">  
<head><title>Modultest</title></head>  
<body>  
<script type="module" src="main.js"></script>  
</body>  
</html>
```

```
//---- main.js ----  
//Import whole module  
import * as lib from './lib.js';  
  
console.log(lib.square(11)); // 121  
console.log(lib.diag(4, 3)); // 5
```

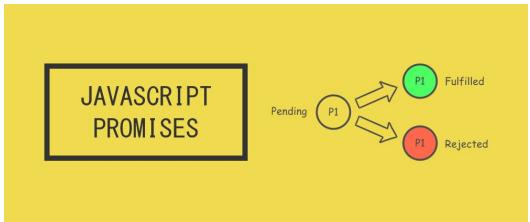
MODULES IN ES6

- Default exports
 - Only 1 per file
 - Useful if only 1 "value" is to be exported, e.g., entire class

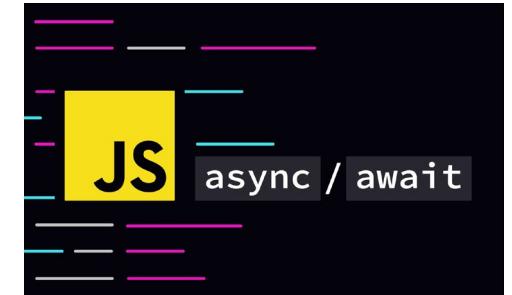
```
//---Person.js---  
export default class {  
    constructor(firstName, lastName){  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    print() {  
        console.log(`Hallo ${this.firstName}  
${this.lastName}`);  
    }  
}
```

```
<!--index.html-->  
<!DOCTYPE html>  
<html lang="en">  
<head><title>Modultest</title></head>  
<body>  
<script type="module" src="main.js"></script>  
</body>  
</html>
```

```
----- main.js -----  
import Person from './Person.js';  
let p1 = new Person("Hannes",  
"Schönböck");  
p1.print();
```



ADVANCED CONCEPTS



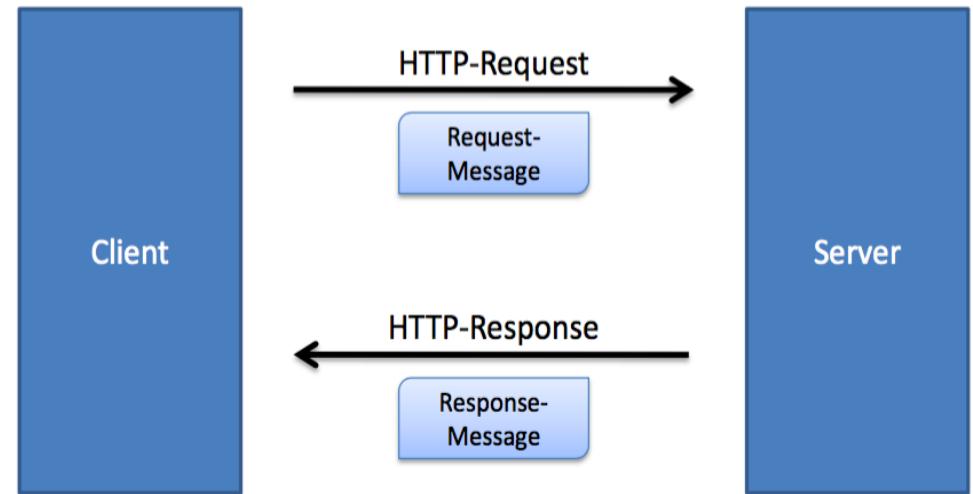
Web Design und Programmierung

AJAX



HYPertext Transfer Protocol

- Data transfer between server and clients based on HTTP Protocol
- De facto standard communication protocol on the web
 - Text-based (= non-binary) Request/response concept
 - Client sends request to server
 - Server sends response to client
- Stateless
 - Server handles all client requests completely independently of each other
- Connectionless
 - No permanent connection between client and server necessary

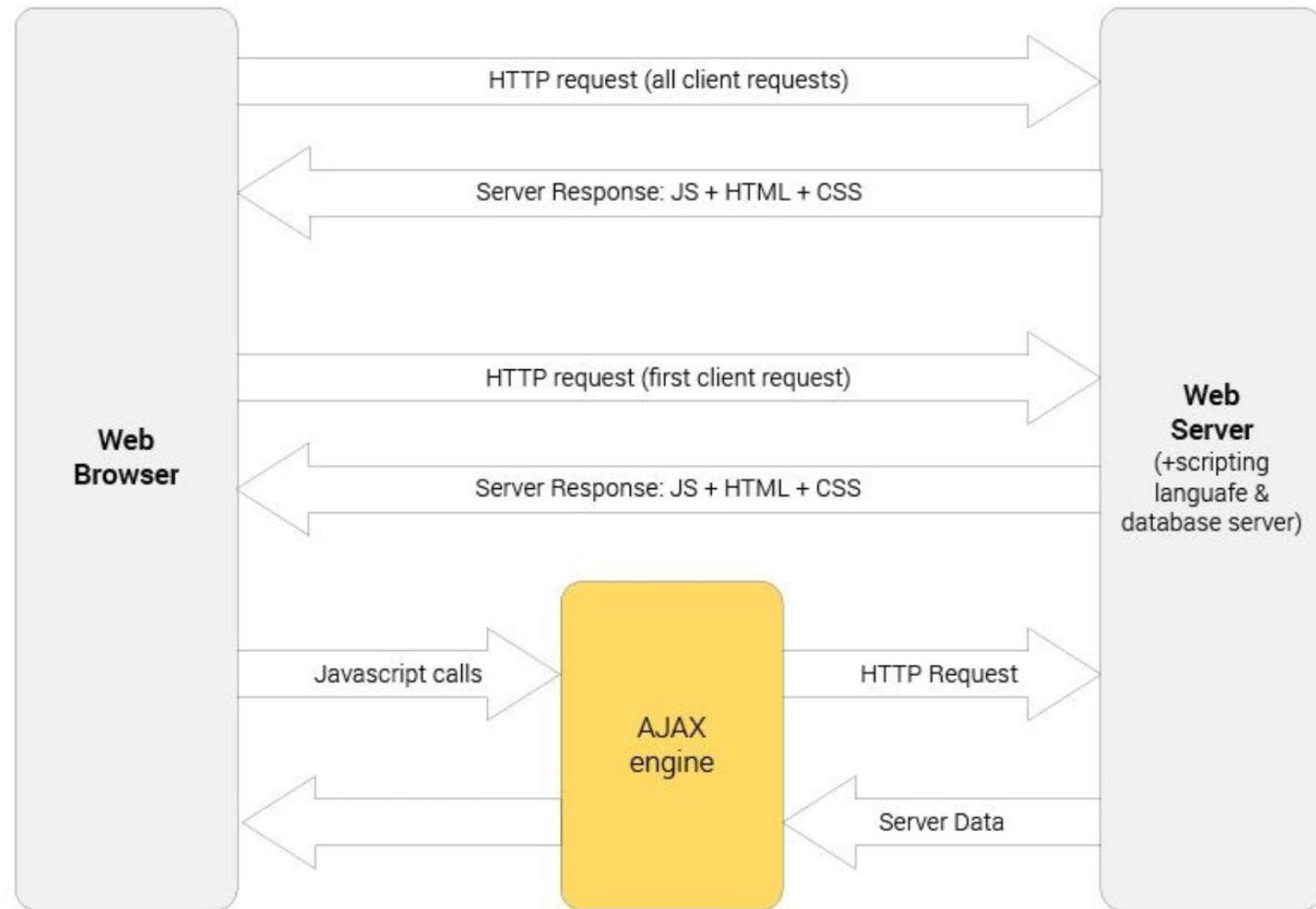


REQUEST/RESPONSE - VERBS

- Describes the action to be performed for the resource identified via the URL
- HTTP recognizes eight methods (“verbs”)
 - **OPTIONS**: Request information for communication path to a resource
 - **GET**: Request resource
 - **HEAD**: Request resource, header only
 - **POST**: Create new resource
 - **PUT**: Change resource
 - **DELETE**: Delete resource from server
 - **TRACE**: For test purposes, request message is sent back in the body of the response
 - **CONNECT**: Tunneling with HTTP proxies
- Browser sends requests to server
 - HTTP GET and POST are used
- Browser receives and interprets responses
 - Response to HTTP status code Automatic forwarding (3xx)
 - Display of errors (4xx, 5xx)
 - HTML / CSS is interpreted and displayed by the rendering engine
 - Engine for executing Javascript
 - Other resources (images, documents, etc.) are either embedded in the display or saved as files

AJAX (ASYNCHRONOUS JAVASCRIPT AND XML)

- The idea behind Ajax was to make websites more dynamic. Before, only full page reloads were possible.
- Allows to fetch additional data at any time from the web server after loading the site.
- It is used nowadays mainly to load JSON data, but any type of data can be fetched.



AJAX AND FETCH

- Fetch API allows to send (any) HTTP Request from a JavaScript program
- `fetch()` only supports asynchronous usage and support Promises

```
let url ="https://someurl.com/api/data";
fetch(url)
  .then(res=>res.json())
  .then(data=>{
    //handle the data
  })
  .catch(err=>console.log(err));
```

```
fetch("http://localhost:3000/shopping/"+id,{
  method: "DELETE"
}).then(response => response.json())
.then(data => {
  ...
}).catch(err =>{
  console.warn("Something went wrong...",err)
});
```

```
fetch("http://localhost:3000/shopping/"+id,{
  method: "PUT",
  headers: {'Content-Type': 'application/json' },
  body: JSON.stringify(obj), //Data to send to server
}).then(response => response.json())
.then(data => {
  .. //do sth with changed data returned from server
});
```

PROMISES AND ASYNC/AWAIT



ASYNCNCHRONOUS EXECUTION

- Code, that gets executed after a certain “event” occurred
 - E.g., User interaction like click or if HTTP Request returned desired data (in case of AJAX)
- We currently know callbacks
- Callback is a function that is called by function to return a result
 - Asyncron
 - Functions are “first-class objects”
 - Objects as parameters of functions
- Callbacks necessary because JS event controlled
- Callbacks allow certain code to be executed only after certain code has been executed

```
function doHomework(subject, callback) {  
    alert(`Starting my ${subject} homework. `);  
    callback();  
}  
  
function alertFinished(){  
    alert('Finished my homework');  
}  
  
doHomework('math', alertFinished);
```

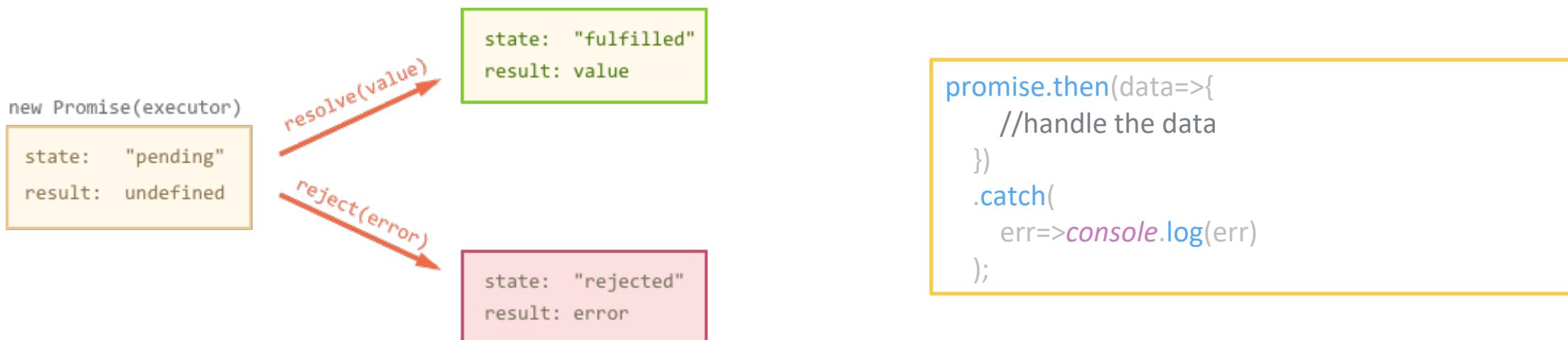
CALLBACK HELL

- Calls from multiple nested callbacks quickly become confusing
- Solutions
 - Avoid anonymous callbacks
 - Use of Promises (since ES6)
- Promise encapsulates the result of an asynchronous operation
 - Placeholder for result or error
- Promises simplify execution and nesting of asynchronous calls
- Promise is JS object combines “producing code” and “consuming code”
 - “producing code”: operation that takes time and is therefore executed asynchronously (e.g. access to external API, AJAX calls, ...)
 - “consuming code”: waits for the result of the “producing code”
- Promise publishes the result

```
a(function (resultsFromA) {  
  b(resultsFromA, function (resultsFromB) {  
    c(resultsFromB, function (resultsFromC) {  
      d(resultsFromC, function (resultsFromD) {  
        e(resultsFromD, function (resultsFromE) {  
          f(resultsFromE, function (resultsFromF) {  
            console.log(resultsFromF);  
          })  
        })  
      })  
    })  
  })  
});
```

PROMISE

- Promise has 3 states
 - Pending: Result of Promise not yet known, asynchronous operation not yet completed
 - Fulfilled: Asynchronous operation successfully completed
 - Rejected: Asynchronous operation failed
 - Cause of error can be queried
- Promise takes so-called executor function when creating
 - Producing code, starts automatically
 - Executor calls 1 of 2 functions
 - Resolve(value) - Operation successful, returns result, state set to fulfilled
 - Reject(error) - error case, state changes to rejected



PROMISES - EXAMPLE

- “When you graduate from BA, your parents will give you a smartphone”
 - Promise – you don't yet know for sure whether the event will happen and whether your parents will keep their promise
- Promise → pending: You don't know for sure whether you will get the smartphone until you graduate
- Promise → resolved: BA completed and cell phone received
- Promise → rejected: BA not completed or parents do not buy a smartphone

```
let buyPhone = true;

// Promise
let willIGetNewPhone = new Promise((resolve, reject) =>{
    if (buyPhone) {
        let phone = {brand: 'Samsung', color: 'black'};
        resolve(phone); // fulfilled
    } else {
        let reason = new Error('Parents are sad');
        reject(reason); // reject
    }
});
```

```
// call our promise
let askParents = function () {
    willIGetNewPhone
        .then(fulfilled => {
            // yay, you got a new phone
            console.log(fulfilled);
            // output: { brand: 'Samsung', color: 'black' }
        })
        .catch(error) =>{
            // oops, parents don't buy it
            console.log(error.message);
            // output: ,parents are sad'
        });
};
```

ASYNC/AWAIT

- Async/Await has been included in ES 2017
 - Good support in current browsers
- `async` functions
 - Function always returns a Promise (automatic wrapping)

```
async function f() {  
    return 1;  
}  
  
f().then(alert); // 1
```



```
async function f() {  
    return Promise.resolve(1);  
}  
  
f().then(alert); // 1
```

- `await` enables the expectation of a Promise
 - Only possible within `async` function

```
async function f() {  
    let promise = new Promise((resolve, reject) => {  
        setTimeout(() => resolve("done!"), 1000)  
    });  
    let result = await promise; // wait till the promise resolves (*)  
    console.log(result); // "done!"  
}  
f();
```

CUSTOM EVENTS AND OBSERVER PATTERN AND MVC PATTERN



CUSTOM EVENTS

- Events occur when the user interacts with the browser
- Programmer can react to these events
- Events already known from DOM
 - onclick, onsubmit, onload, ...
- In addition to predefined events, you can also create your own
 - **Specificity:** They signal unique actions within your application. Imagine an "itemAdded" event for a shopping cart or a "levelCompleted" event in a game.
 - **Decoupling:** Code that creates events (like a form submission) is separate from code that listens for them (like updating a progress bar). This improves code maintainability.
 - **Data Transfer:** You can attach custom data to events using the detail property, allowing informative messages to be passed along.

CUSTOM EVENTS

- **Event Initialization:** initialize a new custom event using **CustomEvent**
 - Parameter: event type and an optional object containing any additional data to be passed along with the event.
- **Dispatching the Event:** Once the custom event is initialized, we can dispatch it on a specific DOM element using the `dispatchEvent()` method.
- **Subscribing to Custom Events:** To respond to custom events, we need to add event listeners to the target elements. Event listeners "listen" for a specific event type and execute a function when that event occurs.

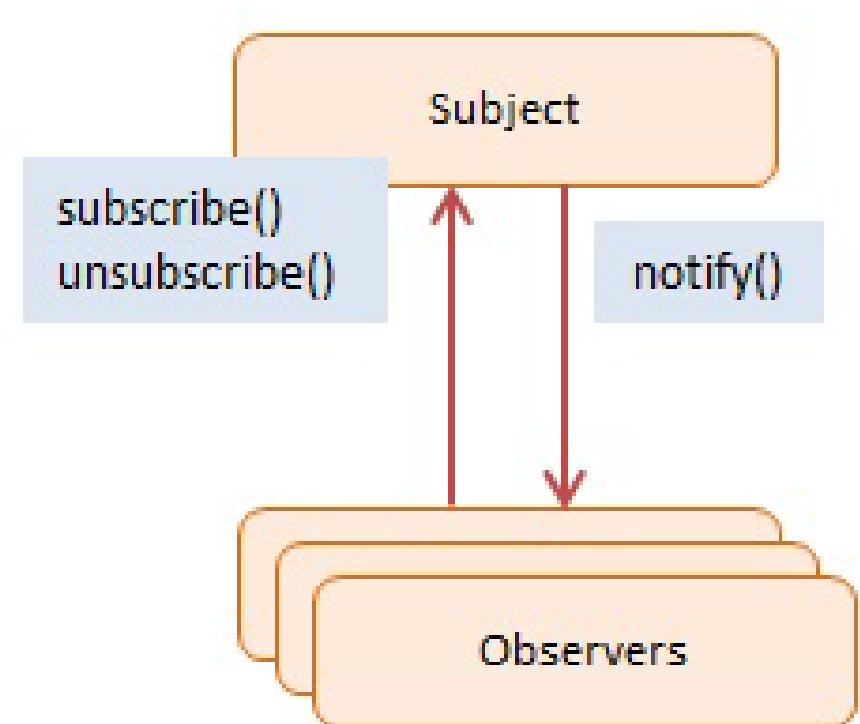
```
// Create a new custom event
const customEvent = new CustomEvent('customEventType', {
  detail: { key: 'value' } // Optional additional data
});
```

```
// Dispatch the custom event on a DOM element
document.dispatchEvent(customEvent);
```

```
// Add an event listener for the custom event
document.addEventListener('customEventType',
  (event) => {
    // Event handling logic here
    console.log('Custom event triggered with data:', event.detail);
});
```

OBSERVER PATTERN

- Enables 1:n relationship between objects so that when one object changes, all dependent objects are notified
- Common pattern in OO languages
 - Enables loose coupling
- Event handlers are based on observer patterns
- **Subject**
 - Holds list of observers
 - Provides methods for registering/deregistering
 - Sends notifications (events)
- **Observer**
 - Provides event handler method



OBSERVER PATTERN IN JAVASCRIPT

- General subject module that can be reused
- Associative array for several different events
- Derive specific classes to obtain functionality

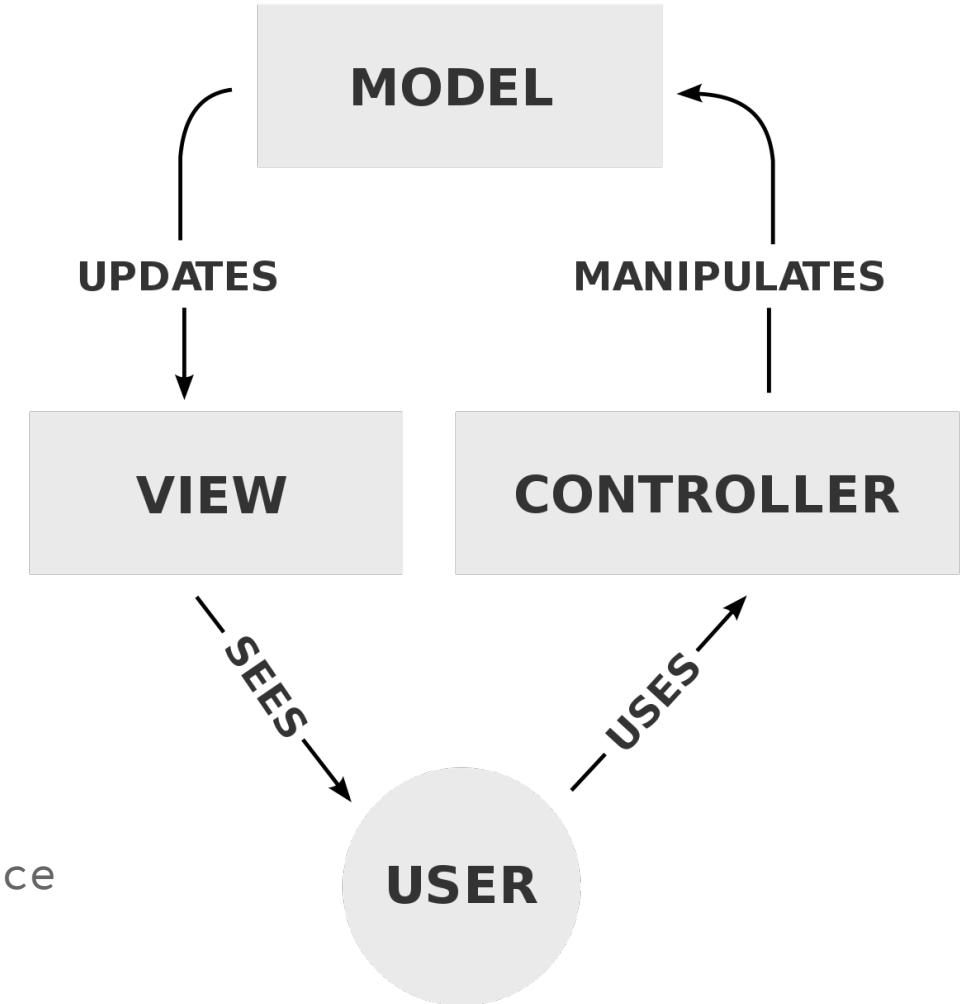
```
export default class Subject{  
    //assoziativer Array  
    observers =[];  
  
    subscribe(event, observer, callback) {  
        if(this.observers[event] === undefined){  
            this.observers[event] = [];  
        }  
        this.observers[event].push({  
            observer: observer,  
            callback: callback  
        });  
    }  
  
    unsubscribe(event,observer) {  
        let observersForEvent = this.observers[event];  
        if(observersForEvent === undefined){  
            throw "No observers for event " + event;  
        } else {  
            this.observers[event] = observersForEvent.filter(  
                o => o.observer !== observer  
            );  
        }  
    }  
  
    notify(event,obj){  
        let observersForEvent = this.observers[event];  
        if(observersForEvent === undefined){  
            throw "No observers for event " + event;  
        } else {  
            for(let o of observersForEvent){  
                o.callback.call(o.observer,obj);  
            }  
        }  
    }  
}
```

MODEL-VIEW-CONTROLLER PATTERN

- One of the most common patterns of software development
- Separates presentation and application logic
- Goals
 - Extensive separation of data model and its graphical representation
 - Better maintainability
 - Better reusability
- Builds on observer pattern
 - Decouples event-based inputs from their output
 - Several different views of the data possible
 - Propagation of changes in views
- Decoupling of development

MODEL-VIEW-CONTROLLER PATTERN

- **Model**
 - Contains the data of an application
 - DAO: Data access objects
 - Often obtained from server/API
 - Independent of logic and presentation
 - Notification of changes via observer pattern
 - Model is in the role of the subject
- **View**
 - Takes care of the presentation of the data (UI)
 - Representation in HTML
 - DOM manipulations
 - Observer of the model
- **Controller**
 - Processes user interactions
 - Interacts with model
 - Can also be an observer of the model
- **Many variations of the pattern exist in practice**
 - E.g. controller is often omitted



BUILDING A FRAMEWORK

Web Design und Programmierung

MOTIVATION

- What are Javascript frameworks?
- What are Javascript frameworks for and why do they exist?
- What are the advantages and disadvantages of frameworks?
- What are the three major Javascript frameworks and what special features do they have?

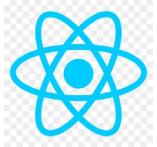
ADVANTAGES

- **Abstraction and efficiency**
 - Prefabricated structures, reduce code and speed up development
- **Consistency and best practices**
 - Uniform code standards & application of proven methods
- **Modularity and maintainability**
 - Facilitate structuring, promote reusability and code maintainability
- **Community support**
 - Often have active communities -> Documentation and exchange of knowledge and resources

DISADVANTAGES

- **Learning curve**
 - Time required to learn to use learn
- **Overhead**
 - Frameworks can add additional code -> for certain projects unnecessary
- **Flexibility limitations**
 - Have fixed structures -> may not fit perfectly to individual requirements
- **Dependencies**
 - Frameworks often introduce external dependencies -> can increase file size and loading times

COMMON JS FRAMEWORKS



REACT

- Developed by Facebook. Most widespread e.g. used by Uber, AirBnb, ...
- **Special feature**
 - Only deals with the view of MVC.
 - Must be extended using other components
- **Downside**
 - Only unidirectional data binding.

<https://react.dev/>



ANGULAR

- Developed by Google. Popular for larger projects.
- **Special feature**
 - Comprehensive MVC framework, handles all aspects of the application.
 - Works with TypeScript.
- **Downside**
 - Steeper learning curve, more complex

<https://angular.io/>



VUE

- Developed by Evan You. Popular for small to medium projects.
- **Special feature**
 - Lightweight and simpler framework, can be easily integrated into existing projects
- **Downside**
 - Less comprehensive -> can cause limitations for larger corporate projects.

<https://vuejs.org/>

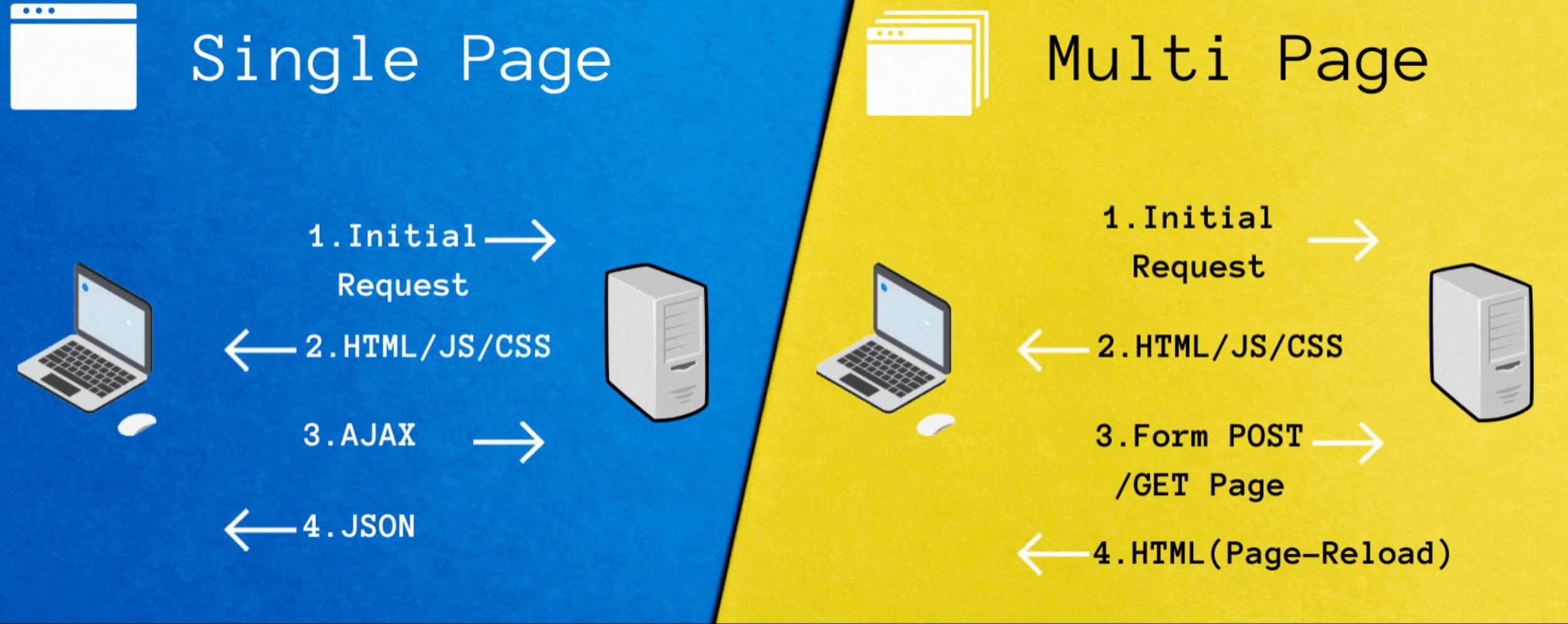
WDP-JS

- Developed by You 
 - **Special feature**
 - Covers the most important aspects of Single-Page-Applications (SPAs) and builds on well-known concepts (OOP, MVC, Observer,...)
 - Downside: Not quite production-ready
 - Shows the common principles of SPAs
 - Knowledge transferable to common modern frameworks
-

SINGLE PAGE APPS (SPAs)

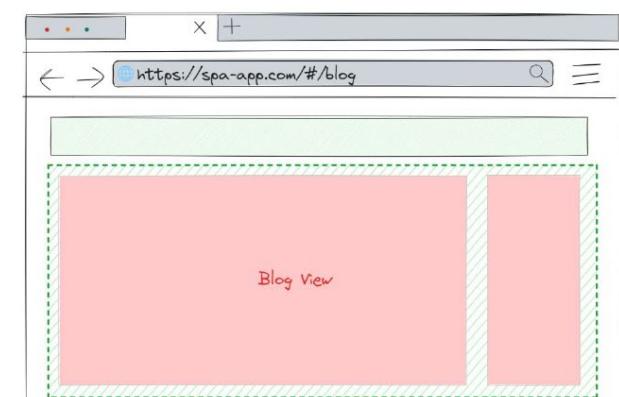
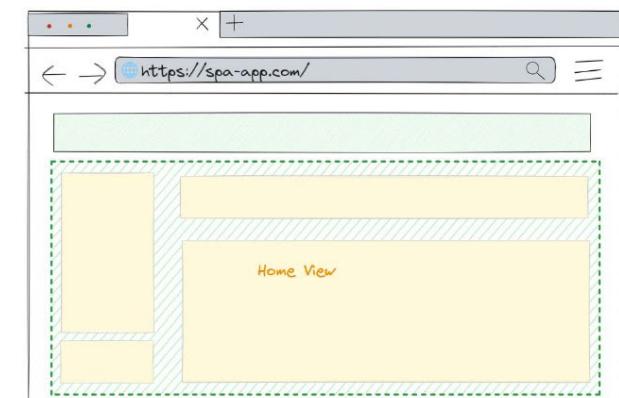
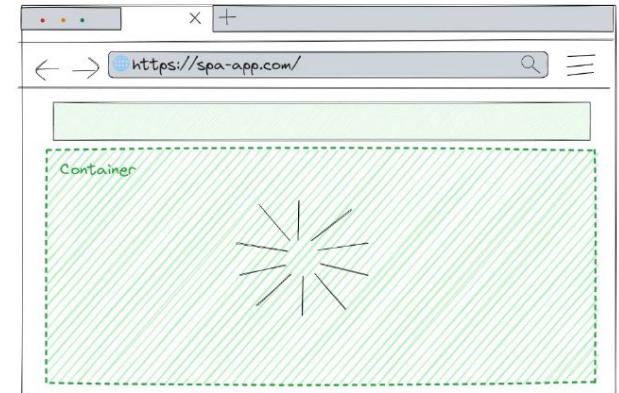
- What is a Single Page Application?
- What are the **advantages** and **disadvantages** of an SPA?
- How does an SPA **differ from a classic MPA**?
- What are the **components** of an SPA and how are they connected?
 - What are the advantages of **components & templates**?
 - How do **routers** and **routes** interact be able to navigate in SPAs?
 - What is **data binding** and how does it work?
- How does the (classic) architecture of an SPA relate to MVC?

MULTI PAGE APP VS SINGLE PAGE APP



SPA BASICS

- Access to application via **index.html** -> Container element is still empty
- JS checks current URL and **loads corresponding view** (~page) into defined container element
- URL changes (navigation via links) are intercepted with JS and the DOM content of the container element is dynamically replaced with the respective new view
- Additional necessary data and content are loaded via AJAX



SPA ADVANTAGES

- **User experience**
 - During use no visible page reload
- **Performance**
 - The entire document is not reloaded with every request
- **Datascope**
 - The data model consists across “Pages” (view) the entire session

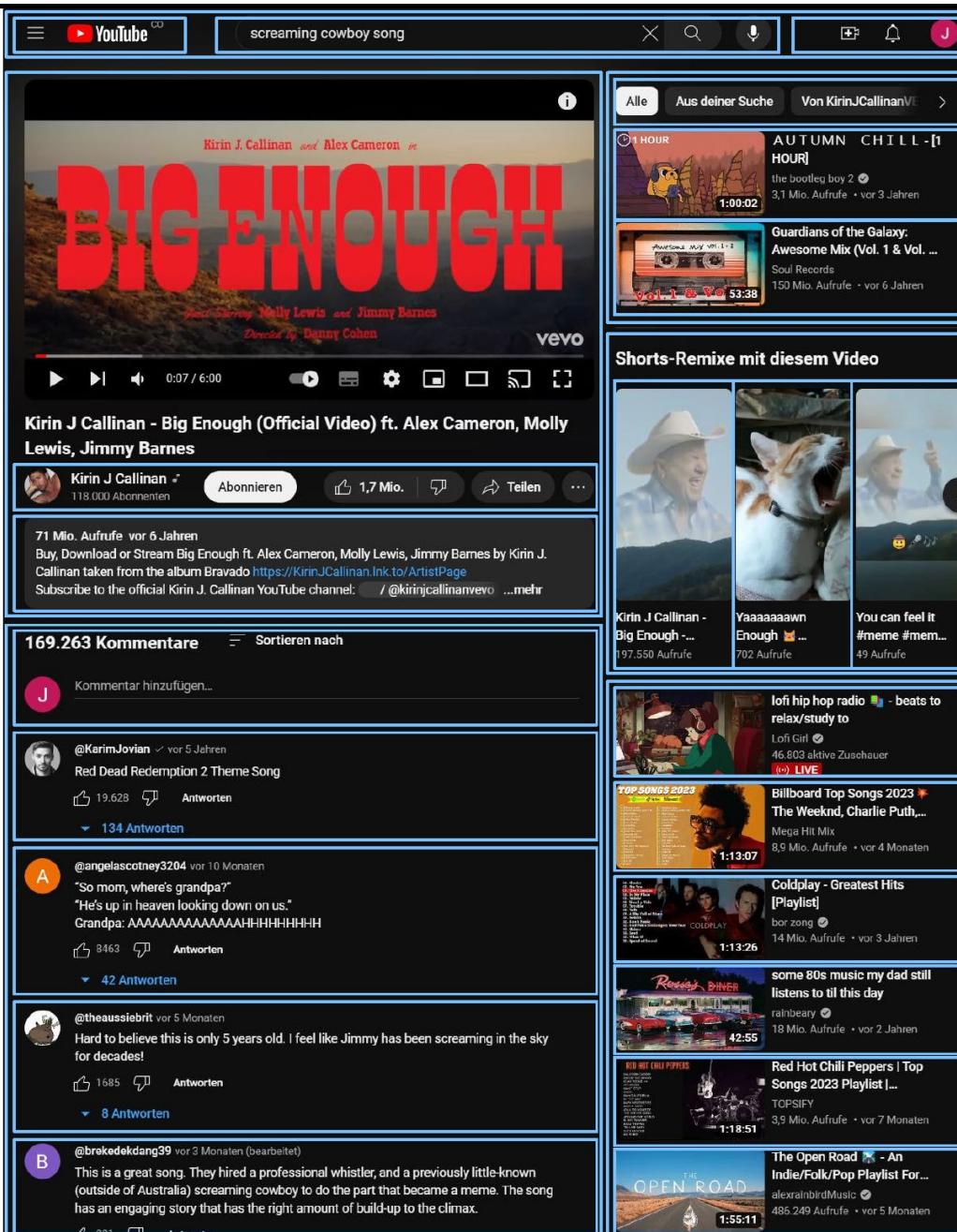
SPA DISADVANTAGES

- **SEO**
 - Only part of the content is visible to search engines
- **Preloading**
 - Depending on the application a lot of data must be preloaded
- **Effort**
 - More complex than static HTML pages and traditional MPA apps

COMPONENTS AND TEMPLATES

COMPONENTS

- Components are **key** in FWs
- **Encapsulates functionality** and structure into **reusable UI elements** (~ Building Blocks)
- Has a **template** -> Defines the HTML structure and can be **dynamically filled** with content
- Allow **modular structuring** of user interfaces
- Components can manage their own **state** -> e.g. Modal open/closed
- Components may have arbitrary number of child-components



Components of Youtube

TEMPLATES

- **Templates** = HTML templates which are dynamically filled with content / data
- **Filled template** (nothing else than a string with HTML tags) is then **displayed** in the DOM (~rendered)
- Different frameworks -> different template syntax & -formats & engines
BUT **always same schema**



```
<div>
  <h1>Angular Todo App</h1>
  <input [(ngModel)]="newTodo" placeholder="Add a new todo">
  <button (click)="addTodo()">Add Todo</button>
  <ul>
    <li *ngFor="let todo of todos" [key]="todo.id">
      <input
        type="checkbox"
        [checked]="todo.completed"
        (change)="toggleTodoCompletion(todo.id)">
      <span [ngClass]="{{ 'done' : todo.completed }}>{{ todo.text }}</span>
      <button (click)="deleteTodo(todo.id)">Delete</button>
    </li>
  </ul>
</div>
```

À LA ANGULAR



```
<f:section name="content">
  <h1>Typo3 Fluid Todo App</h1>
  <input type="text" placeholder="Add a new todo" id="newTodo">
  <button onclick="{f:uri.action(action: 'addTodo')}">Add Todo</button>
  <ul>
    <f:for each="{todos}" as="todo" iteration="iterator">
      <li>
        <input
          type="checkbox"
          checked="{todo.completed}"
          onclick="{f:uri.action(action: 'toggleTodoCompletion', arguments: '{todo.id}')}">
        <span class="{todo.completed ? 'done' : ''}>{todo.text}</span>
        <button onclick="{f:uri.action(action: 'deleteTodo', arguments: '{todo.id}')}">
          Delete
        </button>
      </li>
    </f:for>
  </ul>
</f:section>
```

À LA TYPO3-FLUID



```
<div>
  <h1>React Todo App</h1>
  <input
    type="text"
    value={newTodo}
    onChange={(e) => setNewTodo(e.target.value)}
    placeholder="Add a new todo"
  />
  <button id="addTodo" onClick={addTodo}>Add Todo</button>
  <ul>
    {todos.map((todo) => (
      <li key={todo.id}>
        <input
          type="checkbox"
          checked={todo.completed}
          onChange={() => toggleTodoCompletion(todo.id)}>
        <span className={todo.completed ? 'done' : ''}>{todo.text}</span>
        <button onClick={() => deleteTodo(todo.id)}>Delete</button>
      </li>
    ))}
  </ul>
</div>
```

À LA REACT

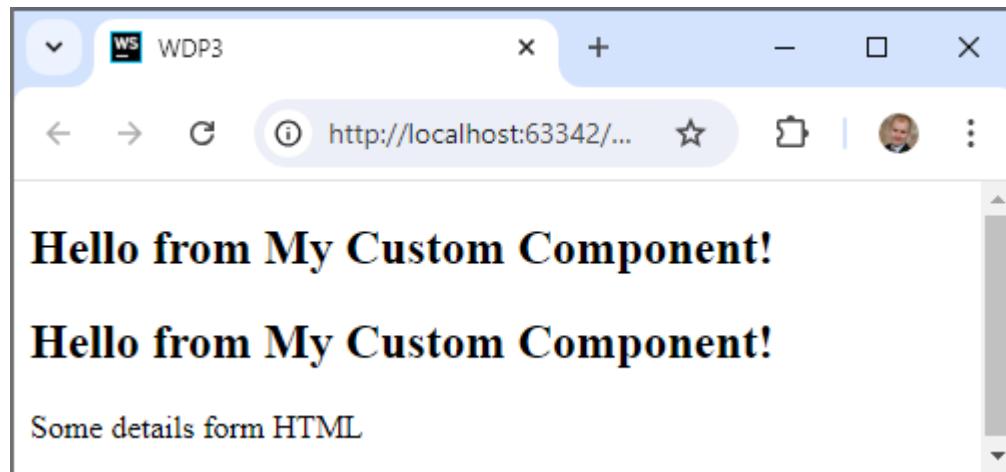
TEMPLATES

- **Easier to read** (than JS DOM Code)
- **HTML structure** clearly represented
- **Less code**
- **Reusable**
- We use **JavaScript Template Literals**
 - already included in JS
 - Similar to templates in common FWs (especially React)

```
// Example for a Blog-Template
template() {
    return /*html*/ `
        <article>
            <h1> ${this.heading}</h1>
            ${this.subheading ? `<h2>${this.subheading}</h2>` : ''}
            <p> ${this.text}</p>
        </article>`;
}
```

COMPONENTS

- We use **HTMLElement** as the base class and customize its structure and functionality as desired ([WebComponents](#))
 - Already available in Javascript
 - Very similar structure to components in large frameworks (e.g. React, Vue, Angular)



```
class MyCustomComponent extends HTMLElement {
  constructor() {
    super();
    this.shadow = this.attachShadow({ mode: 'open' });
    this.text = 'Hello from My Custom Component!';
  }
  connectedCallback() {
    this.shadow.innerHTML = this.template();
  }
  template() {
    return `<h2>${this.text}</h2>
      <p>${this.getAttribute('detail') ?? ''}</p>`;
  }
}
customElements.define('my-custom-component', MyCustomComponent);
```

```
<!-- Usage in HTML -->
<my-custom-component></my-custom-component>
<my-custom-component detail="Some details form HTML">
  </my-custom-component>

// Usage in JavaScript:
const myCustomComponent = new MyCustomComponent();
document.querySelector('body').appendChild(myCustomComponent);
```

COMPONENTS

- It is possible to define styles and e.g., IDs in the template
 - Somewhere else on a page, the same ID or styles for the same class are used
 - With multiple instances of the custom element on the same page, the IDs are duplicated and thus not unique anymore
- **Shadow DOM**
 - Separate DOM tree that belongs to the custom element
 - Open-Mode: access the shadow DOM from outside the custom element possible
 - Closed-Mode: No access from outside
 - If we want to access anything inside the Shadow DOM we have to first select the **shadowRoot** of our custom element and then search inside it

```
document.querySelector('my-custom-component').shadowRoot.querySelectorAll('h2');
```

- **Lifecycle methods** (more available)
 - Automatically run at certain points in the component's lifetime
 - **connectedCallback()**: is called every time the element is added to the page
 - Constructor is only called **once** when element is created
 - **disconnectedCallback()**: called when the element is removed from the page
 - **attributeChangedCallback()**: receives the name of the changed attribute, the old value, and the new value whenever an attribute is changed

COMPONENTS

- **Slots:** passing child elements to a component
- Needed when your component should be able to include any other components as children
- You can define a slot in your template

```
class MyCustomComponentWithSlot extends HTMLElement {  
  constructor() {  
    super();  
    this.shadow = this.attachShadow({ mode: 'open' });  
    this.text = 'Hello from My Custom Component with Slot!';  
  }  
  connectedCallback() {  
    this.shadow.innerHTML = this.template();  
  }  
  template() {  
    return `  
      <h2>${this.text}</h2>  
      <p>${this.getAttribute('detail') ?? ''}</p>  
      <slot name="content"></slot>  
    `;  
  }  
}  
customElements.define('my-custom-component-slot', MyCustomComponentWithSlot);
```

```
<my-custom-component-slot>  
  <div slot="content">  
    <h2>Some content</h2>  
  </div>  
</my-custom-component-slot>
```

STATELESS

- Do **not contain** & store any internal data or **states**
- Care more about **presentation** than functionality (~template only)
- **Examples:**
 - Button that triggers action
 - Blog article component that displays texts
 - ProductCard component, which displays a product in a gallery

STATEFUL

- **Save & manage** your own **state** (~Status), which can change over time
- Take care of **functionality** -> are more complex
- **Examples:**
 - Shopping cart to which products can be added or deleted
 - Slider -> must remember which slide is currently displayed

COMPONENTS LIFECYCLE

- = Different “**life phases**” of a component, while it is used in a script
- Three most important phases (common in frameworks) are
 - constructor / setup / init
 - Only executed once at the very beginning of the creation of the component, not yet available in the DOM
 - mounted / first render
 - Executed after component has been rendered for the first time and component is already available in the DOM
 - updated / rerender / changed
 - Executed each time after the component has been rendered and is available in the DOM
- Code can be executed for each of these phases.
- Depending on the framework, there may be additional or slightly different phases here

“Lifecycle Hooks”

Steps during the creation and “life time” of the component that we can “hook” into and execute code

1. **constructor** (~“setup”)
Executed **only once on component creation before render** (HTML is not available in DOM)

2. **onFirstRender** (~“mount”)
Executed **only once after first render** (HTML is already available in DOM)

3. **onRender** (~“update”)
Executed **every time after render**

COMMUNICATION BETWEEN NESTED COMPONENTS

- How does the child component know that it should update / adapt?
- How does the parent component know that something has happened in the child component?
- Parents communicate data through attributes / properties to their children
- Children communicate data to their parents through events
- Depending on the framework, there may be differences here



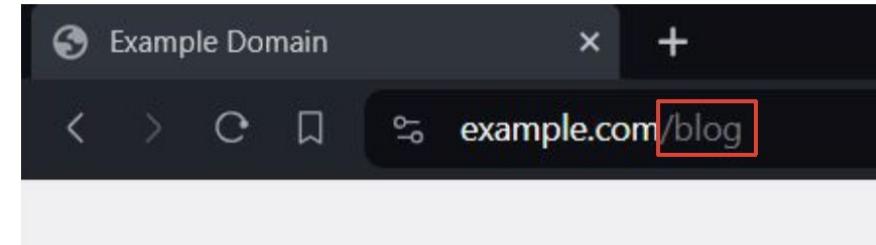
```
parentTemplate () {
  return /*html*/
    <div>
      <child-component childProp="Some data"></child-component>
    </div>;
}

// Child to parent (dispatching Events ~ emitting Events)
childElement. dispatchEvent (
  new CustomEvent ("child-event" , {
    bubbles: true,
    detail: 'Hello something happened in ChildComponent' ,
  }),
);
parentElement. addEventListener ("child-event" ,
  (event) => doSomething (event.detail));
```

ROUTES + ROUTER = ROUTING

ROUTE BASICS

- Navigable **target** point in the application (~page)
- Has a **slug** which identifies the route (~address)
 - Slug is compared with active URL in the browser.
 - If there is a match this route is currently active.
- Has a **view component** assigned which is displayed on this route is displayed

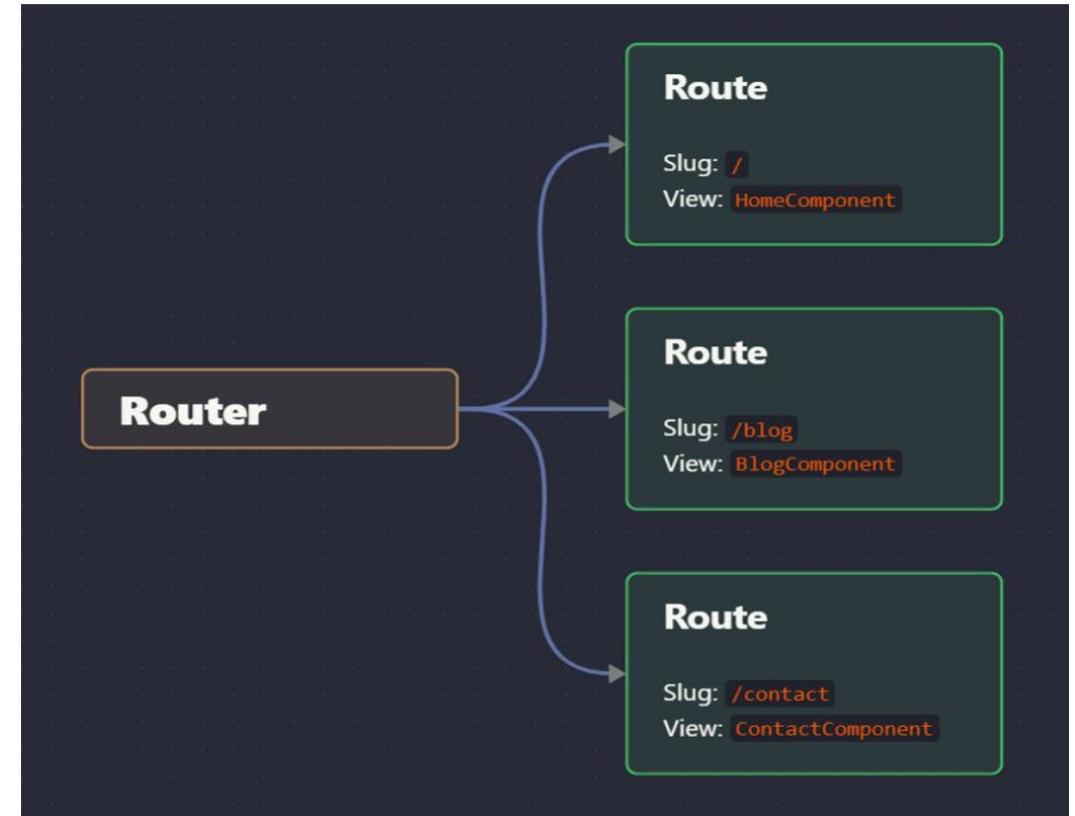


URL: Address with Slug /blog



ROUTER

- Enables navigation
 - SPA actually has only one URL, simulates other accessible URLs within the application
- Is the “view manager”
 - Searches for the appropriate route (with associated view) for the respective URL
- Has an array with all accessible routes
- Has special routes (e.g. start page & 404 page)
- Has a container element in which the routes are to be displayed are to be displayed



ROUTING

- **Hash-based routing**

(window.location.hash)

- Router is equipped with listener for the “hashchange” event
 - is triggered when the “#” in the URL is changed
- When hash changes, “**changeView**” is called

```
window.addEventListener('hashchange', this.changeView);
function changeView() {
  const hash = window.location.hash;
  // ... find the route that matches the hash
}
function redirect(slug) {
  window.location.hash = slug;
}
```

- **History API based routing**

(window.history)

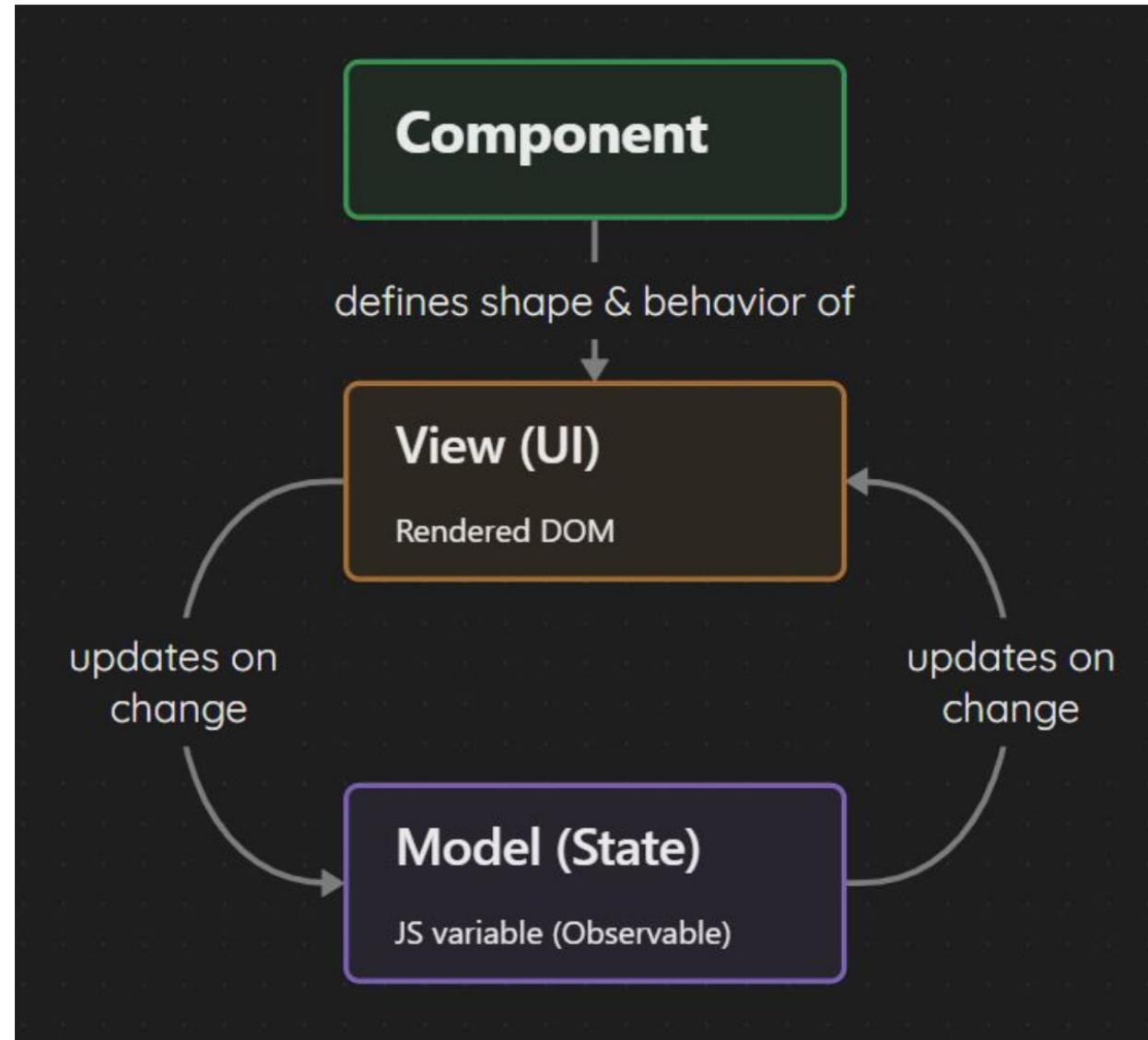
- The router is equipped with listeners for the “popstate” event
 - is triggered when the browser history changes
- When the browser history changes, the “**changeView**” function is called.

```
window.addEventListener('popstate', this.changeView);
function changeView() {
  const path = window.location.pathname;
  // ... find the route that matches the path
}
function redirect(slug) {
  window.history.pushState(null, null, slug);
}
```

DATA BINDING AND REACTIVITY

DATABINDING

- What does data binding mean?
 - Synchronization of display and data
- User interface (UI)
 - Presentation & input controller of the data (e.g. form)
- Data layer (state)
 - Raw data usually managed by the model or a component
- If one changes, the other should also change (unidirectional / bidirectional)



DATABINDING – HOW TO IMPLEMENT IT?

- **Event Driven**

- If input data in the UI changes a corresponding event is fired (e.g. “change”, “click”, “keydown”). Listener reacts accordingly and changes data in the data model

- **Observer model**

- If data in the data model changes the view (subscriber) is notified and triggers a ReRender / Updates in the DOM (Templates are filled with new data and displayed)

```
// HTML
<input id="searchtext" type="text" />

// JavaScript
const element = document.querySelector('#searchtext');

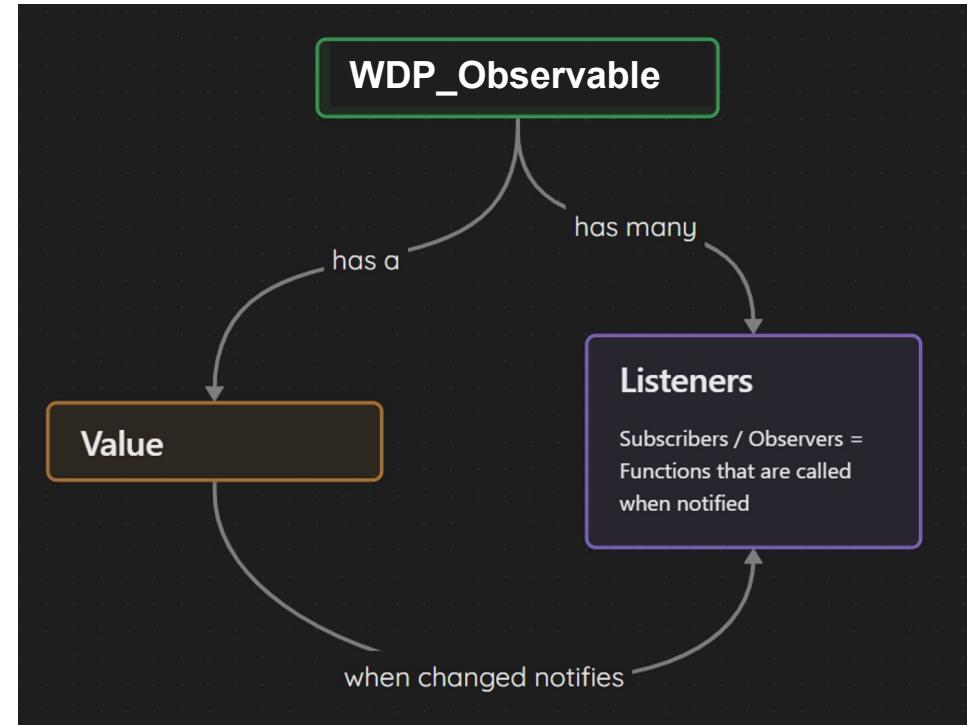
const searchText = new WDP_Observable('Hello World');

// When the input changes, update the observable
element.addEventListener('change', (event) => {
  searchText.value = event.target.value;
});

// When the observable changes, update the input
searchText.subscribe((value) => {
  element.value = value;
});
```

OBSERVABLE

- We want an object that holds a value.
- As soon as that value changes we want to be able to react to those changes by notifying listener functions.
- Other objects / functions should be able to subscribe listener functions to the changes and get notified.



OBSERVABLE

```
const name = new WDP_Observable("Hannes");

// will be called every time the value changes
const listenerFunction = (value) => console.log(`Name
changed to ${value}`);
name.subscribe(listenerFunction);

name.value = "Johannes";
```

```
export default class WDP_Observable {

  _value = null;

  constructor(value = "") {
    this._listeners = []; //stores the callback functions
    this._value = value;
  }

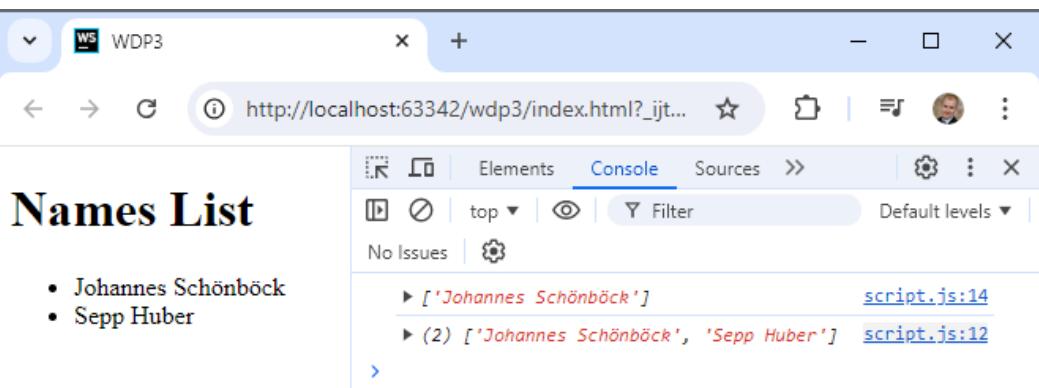
  // Notify listener Function and call them all with the new value
  //as argument
  notify() {
    this._listeners.forEach(listener => listener(this._value));
  }

  // Register a new listener function
  subscribe(listener) {
    this._listeners.push(listener);
  }

  get value() {
    return this._value;
  }

  set value(val) {
    if (val !== this._value) {
      this._value = val;
      this.notify();
    }
  }
}
```

OBSERVABLE – USE WITH CUSTOM COMPONENTS



```
import WDP_Observable from './wdp-observable.js';

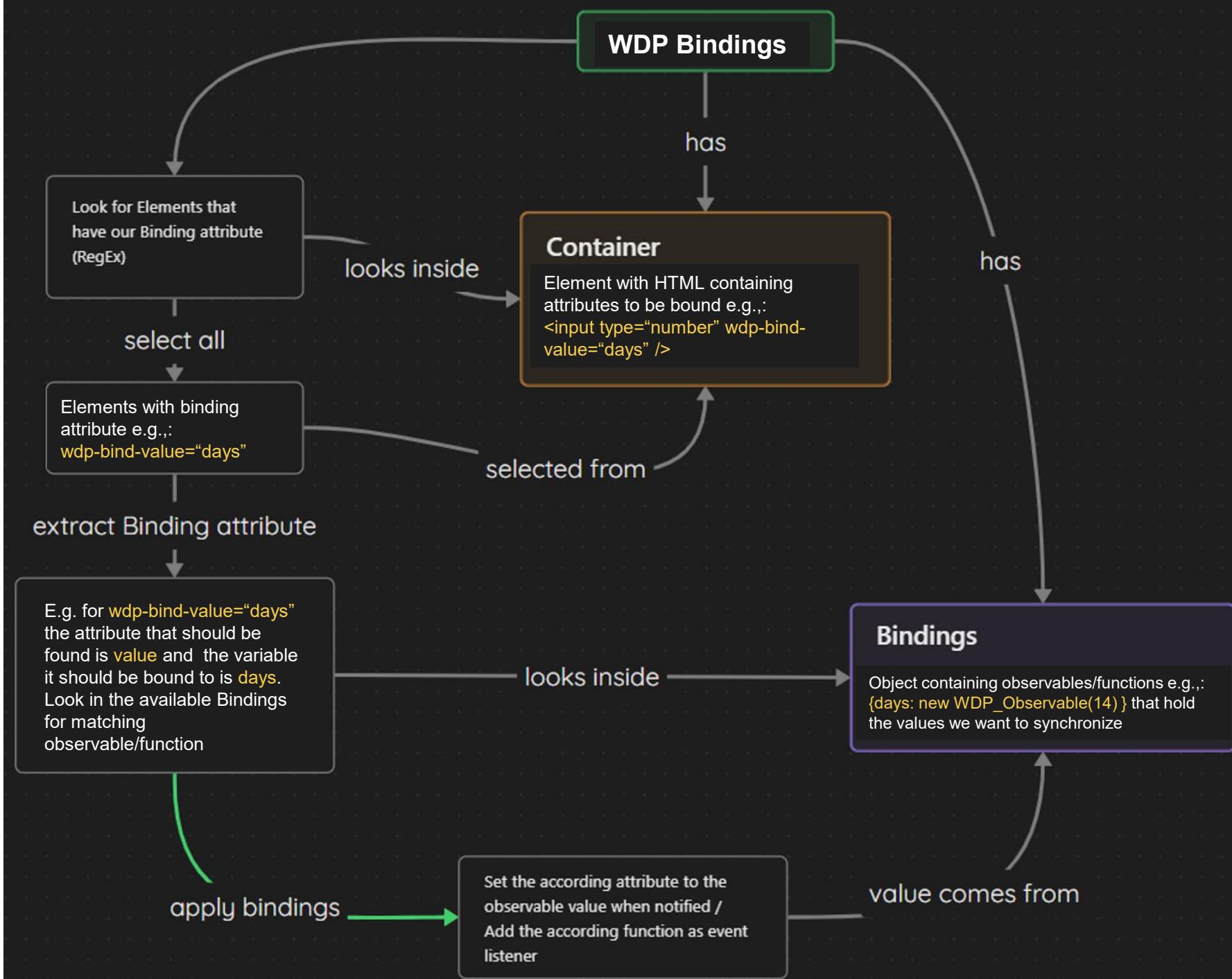
class MyCustomComponent extends HTMLElement {
  constructor() {
    super();
    this.shadow = this.attachShadow({ mode: 'open' });
    this.names = new WDP_Observable(["Johannes Schönböck"]);

    // Rerender our component when the value of observable changes
    this.names.subscribe(value => {
      this.shadow.innerHTML = this.template();
      console.log(value);
    });
    console.log(this.names.value)
  }
  connectedCallback() {
    this.shadow.innerHTML = this.template();
  }
  template() {
    return /*html*/
      <h1>Names List</h1>
      <ul>
        ${
          this.names.value.map(this.listItemTemplate).join('')
        }
      </ul>
    ;
  }
  listItemTemplate(name) {
    return /*html*/`<li>${name}</li>`;
  }
}
customElements.define('my-custom-component', MyCustomComponent);

const myCustomComponent = new MyCustomComponent();
document.querySelector('body').appendChild(myCustomComponent);
myCustomComponent.names.value = [...myCustomComponent.names.value, "Sepp Huber"];
```

DATABINDING

- We use upper **Observables** and **Bindings** to automate the registration of **listeners** & **subscribers** in the background.
 - Simple syntax, similar to established frameworks (e.g., React, Vue, Angular)
 - Our bindings can be used both “unidirectional” as well as “bidirectional”



DATABINDING - GOAL

- Automatically "bind" **HTML attributes** in template to observable JS values in Component
 - skipping the annoying step of manually subscribing, listening and setting the value.
- E.g. `<input wdp-bind-value="newTodoText" />` should bind the "text" observable from the bindings `{newTodoText: new WDP_Observable('someText')}` to the HTML element attribute "value".
- Automatically "bind" **functions** of component as event listeners in our HTML template.
- E.g. `<button wdp-listen-click="addTodo" />` should apply the "addTodo" function from the bindings object `{addTodo: (event) => ...}` as "click" event listener to the HTML element.

```
class TodoComponent extends HTMLElement {
  constructor() {
    super();
    this.newTodoText = new WDP_Observable('');
    //...
  }
  //...
  addTodo() {
    const newTodoText = this.newTodoText.value;
    //... add the new todo
  }
  //...
  template() {
    return `
      <section id="todo_app">
        <input type="text" wdp-bind-value="newTodoText" />
        <button wdp-listen-click="addTodo">Add Todo</button>
        <ul><!-- List rendering goes here --></ul>
      </section>`;
  }
}
```

DATABINDING – STEP 1

- bindAttribute takes an **HTML element elem** (the UI) an **observable** (the Data) and an **attribute name**.
- method sets the attribute called **attribute** of the Element to the value of the **observable** using the **setAttribute** helper
- Subscribes a new arrow function to the observable that sets the HTML attribute to the observable value, every time the observable changes (Unidirectional data binding: Data → UI)
- Bidirectional data binding (UI → Data): If it is an input or select field, we apply listeners
- **applyInputListeners** add according events listeners and updates the value of the observable

```
export default class WDP_Bindings {  
    static bindAttribute(elem, observable, attribute = 'value') {  
        WDP_Bindings.setAttribute(elem, observable, attribute);  
        observable.subscribe(() => WDP_Bindings.setAttribute(elem, observable, attribute));  
        // 2-way-data-binding - if applicable register input listeners  
        if (  
            (elem.tagName === 'SELECT' || elem.tagName === 'INPUT') &&  
            (attribute === 'value' || attribute === 'checked')  
        ) {  
            WDP_Bindings.applyInputListeners(elem, observable, attribute);  
        }  
    }  
  
    static setAttribute(elem, observable, attribute) {  
        elem[attribute] = observable.value;  
    }  
  
    static applyInputListeners(inputElem, observable) {  
        if (inputElem.tagName === 'SELECT') {  
            inputElem.addEventListener('change', () => observable.value =  
                inputElem.options[inputElem.selectedIndex].value);  
        } else if (inputElem.type === 'checkbox' || inputElem.type === 'radio') {  
            inputElem.addEventListener('change', () => observable.value = inputElem.checked);  
        } else {  
            inputElem.addEventListener('keyup', () => observable.value = inputElem.value);  
            inputElem.addEventListener('change', () => observable.value = inputElem.value);  
        }  
    }  
}
```

```
const bodyElem = document.querySelector('body');  
bodyElem.innerHTML += `  
    <input id="search" placeholder="searchText" />  
`;  
  
const searchElem = document.querySelector('#search');  
const searchText = new WDP_Observable();  
searchText.subscribe(value => console.log('Attribute Binding works - new "value" of search: ', value));  
WDP_Bindings.bindAttribute(searchElem, searchText, 'value');  
searchText.value = "Hello World";
```

DATABINDING – STEP 2

- Create the according regex in the constructor
- applyBindings: Search for the regex and bind the attributes or listeners
- No explicit binding in JS needed any more

```
const bodyElem = document.querySelector('body');
bodyElem.innerHTML += `
<input id="search" wdp-bind-value="searchText" placeholder="searchText" />
<input id="number" type="number" wdp-listen-change="changeHandler" />
`;

const searchText = new WDP_Observable('some search');
searchText.subscribe(value => console.log('Databinding works - new value of search:', value));

const bindingData = {
  searchText: searchText,
  changeHandler: (event) => console.log('Listener binding works - change event triggered:', event),
};

const bindings = new WDP_Bindings(bindingData, bodyElem);
bindings.bind();
```

```
export default class WDP_Bindings {
  constructor(objObservables, container) {
    this.bindingData = objObservables; // The State / Data
    this.container = container; // The UI / View
    // Pattern: wdp-bind-AttributeName="variableName"
    this.valueBindRegex = /(wdp-bind-([a-zA-Z]+))=(\"(\w+)\")/g;
    // Pattern: wdp-listen-listenerType="listenerFunctionName"
    this.listenerBindRegex = /(wdp-listen-([a-zA-Z]+))=(\"(\w+)\")/g;
  }

  bind() {
    this.applyBindings(this.container);
  }

  // Need to run AFTER the template / component was rendered
  applyBindings(container) {
    const containerHtml = container.innerHTML;

    // Bind values
    for (let [x, selector, attributeName, variableName] of containerHtml.matchAll(this.valueBindRegex)) {
      container.querySelectorAll(`[ ${selector}="${variableName}" ]`).forEach(elem => {
        const observable = this.bindingData[variableName.trim()];
        if (observable) {
          WDP_Bindings.bindAttribute(elem, observable, attributeName);
        } else console.warn(`No observable found with name "${variableName}" found in bindings:`, this.bindingData, elem);
      })
    }

    for (let [x, selector, listenerType, listenerFunctionName] of containerHtml.matchAll(this.listenerBindRegex)) {
      container.querySelectorAll(`[ ${selector}="${listenerFunctionName}" ]`).forEach(elem => {
        const listenerFunction = this.bindingData[listenerFunctionName.trim()];
        if (listenerFunction) {
          elem.addEventListener(listenerType, listenerFunction.bind(this.bindingData));
        } else console.warn(`No listenerFunction with name "${listenerFunctionName}" found in bindings:`, this.bindingData, elem);
      })
    }
  }
}
```

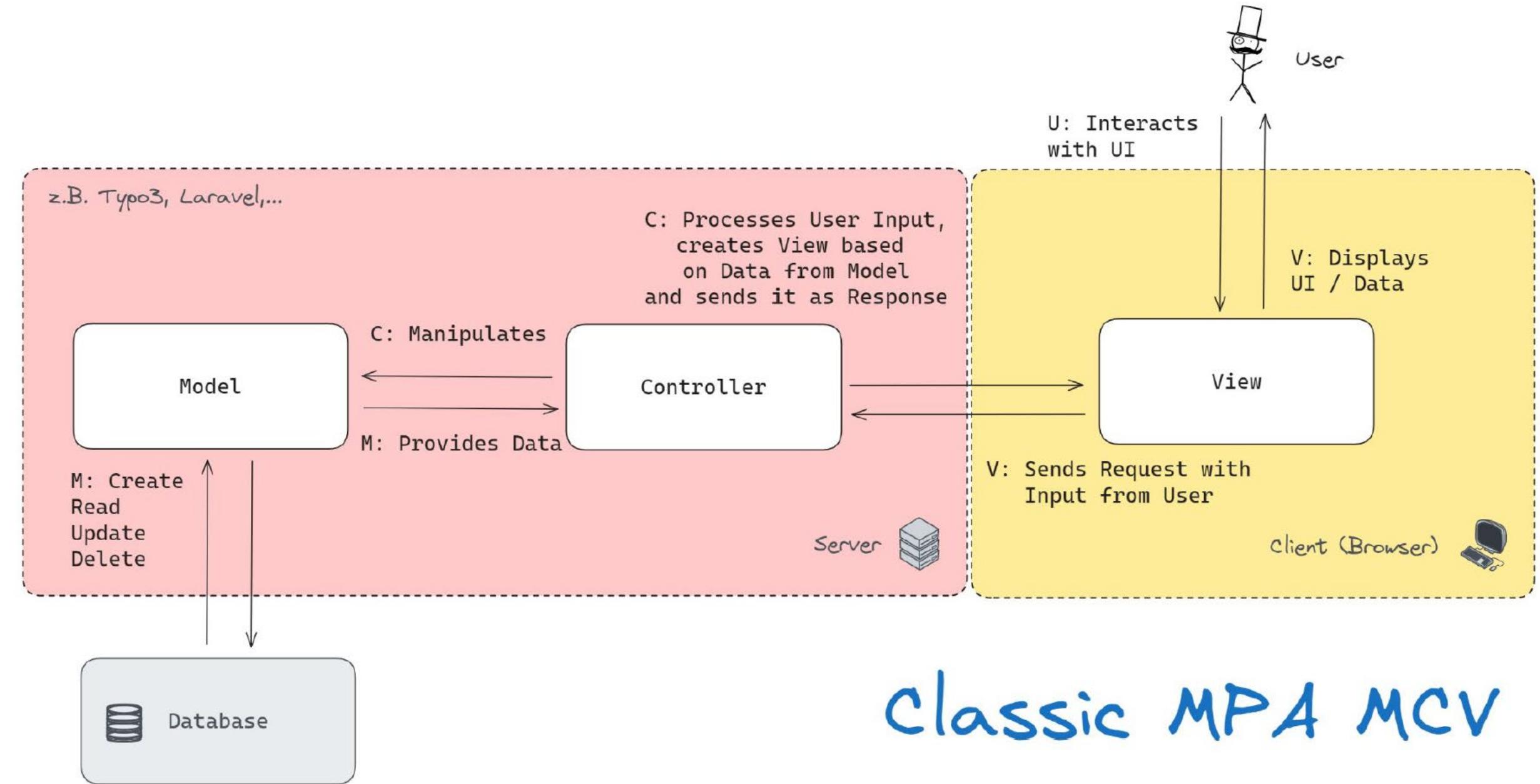
DATABINDING – STEP 3

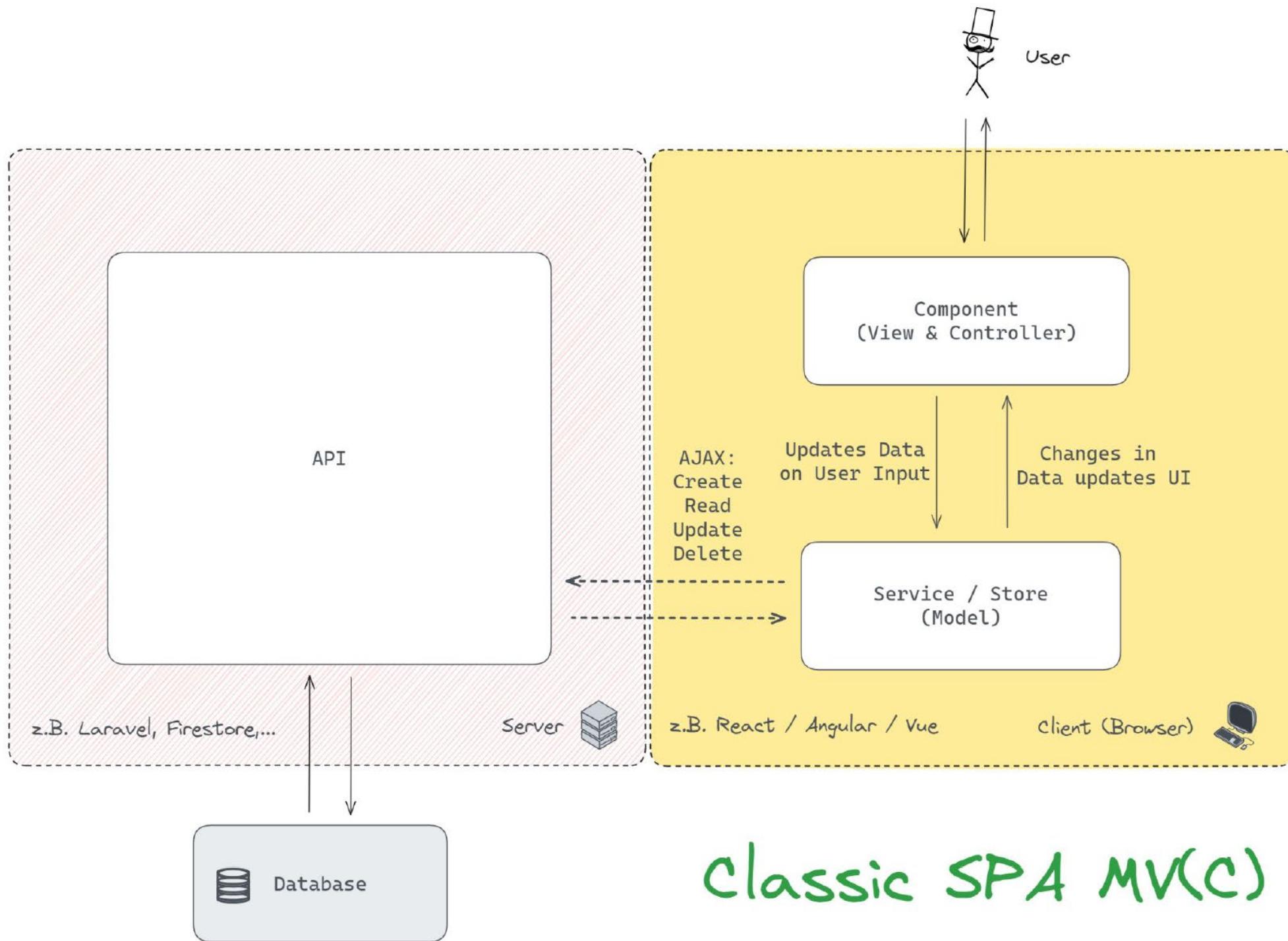
- Component automatically applies it's bindings every time the component is rendered
 - No manual registration of EventListeners needed
 - Could act as a base class for all components (override the actual rendering)

```
class DataBindingTestComponent extends HTMLElement {  
    constructor() {  
        super();  
        this.shadow = this.attachShadow({ mode: 'open' });  
        this._bindings = null;  
        this.newTodoText = new WDP_Observable('');  
        this.newTodoText.subscribe(console.log);  
        this.todos = new WDP_Observable([]);  
        this.registerRenderDependencies([this.todos]);  
    }  
  
    connectedCallback() {  
        this.render();  
    }  
  
    render(){  
        this.shadow.innerHTML = this.template();  
        if(!this._bindings) {  
            this._bindings = new WDP_Bindings(this, this.shadow);  
        }  
        this._bindings.bind();  
    }  
  
    addTodo() {  
        const newTodoText = this.newTodoText.value;  
        this.todos.value = [...this.todos.value, newTodoText];  
    }  
}
```

```
// Manually register dependencies to automatically re-render component when dependency (observable) changes  
registerRenderDependencies(observables = []) {  
    observables.forEach(obs => {  
        obs.subscribe(() => this.render());  
    });  
}  
  
template() {  
    return /*html*/'  
        <section id="todo_app">  
            <input type="text" wdp-bind-value="newTodoText" />  
            <button wdp-listen-click="addTodo">  
                Add Todo  
            </button>  
            <ul>  
                ${ this.todos.value.map(todo => /*html*/`<li>${todo}</li>`).join('') }  
            </ul>  
        </section>'  
    }  
}  
customElements.define('custom-component', DataBindingTestComponent);  
  
const dataBindingTestComponent = new DataBindingTestComponent();  
document.querySelector('body').appendChild(dataBindingTestComponent);
```

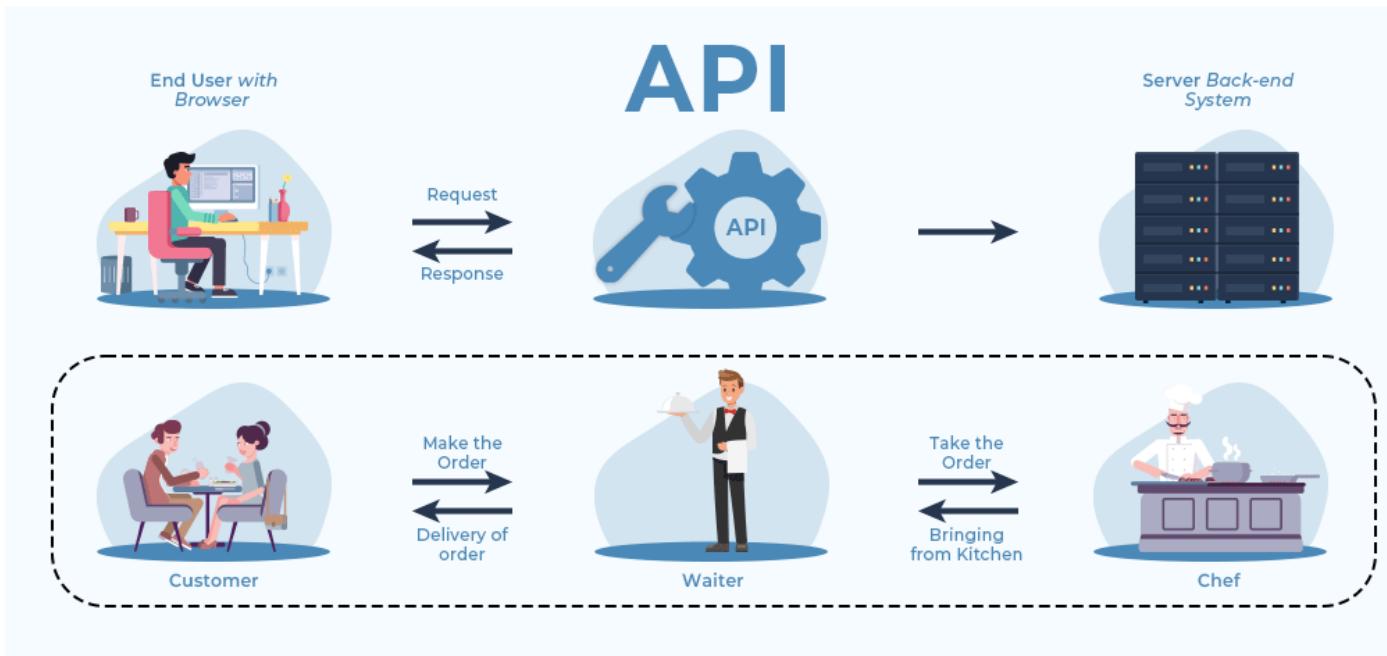
APIS





API

- Application Program Interface
- Data interface
 - Used to request and transmit data
- ~Data menu
 - You can see what is available and can “order” from it. You are served the corresponding data but the preparation is not visible
 - E.g., Weather API, News API
- Potential **disadvantages** of APIs
 - **Security**
 - A gate must be opened that needs to be secured accordingly (e.g. with API keys, JsonWebToken, rate limiting,...)
 - **Visibility**
 - You disclose which data you have available and possibly have to justify this
 - **Effort**
 - APIs ain't building themselves



API USAGE

- Call mostly **asynchronous** as API needs an (in)certain time to process the request process
- Different standards (e.g. REST) and protocols (e.g. SOAP) -> Decisive for the data format used data format used (e.g., JSON or XML)
- APIs offer routes (~menus) which can be queried

```
async function getDataFromApi (apiUrl ) {  
    return fetch(apiUrl )  
        . then(response => response .json())  
        . then(data => {  
            console .log(data);  
            return data;  
        });  
}
```

```
// Simplified Examaple of a Todo App with API  
function runTodoApplication () {  
    initApplication ();  
    console .log("Hey, App is looking so fresh! Let's add some Data!" );  
    getDataFromApi ('https://dummyjson.com/todos' )  
        . then((todos) => displayNewTodos (todos));  
    console .log("Oh Snap! I am executed before the Data is here?" );  
}
```

STORING DATA IN THE BROWSER

**Cookies vs. Local Storage
vs. Session Storage**



LOCAL DATA STORAGE

- **Usage Scenarios**
 - Shopping Cart, Favourites, Offline Notes, User Preferences (e.g., Theme, Language,...), User Identifikation
- **Offline support**
 - Enable applications that work without an active Internet connection
- **Data control**
 - More control for users over their personal information, as it is not sent to servers -> user can delete data themselves
- **Simple implementation**
 - Using data storage like **localStorage** is relatively simple and does not require complex server configurations.
- **Persistence**
 - Data can be persisted between browser sessions.
- **User experience**
 - User can seamlessly access previously entered information, settings or the status of the application.
- **Performance**
 - Locally stored data does not have to be loaded from the server -> reduces server load and saves loading time.
- **Limited memory capacity**
 - Browser has limited capacity for local data. Too much locally stored data -> possible storage and performance problems
- **Data security**
 - Local data storage is more susceptible to data loss or corruption (e.g. browser crash)
- **Security risks**
 - Local data storage is susceptible to various risks such as Cross-site scripting (XSS) or cross-site request forgery (CSRF) -> possible access or manipulation of data -> CAUTION for sensitive information (e.g., login data)

COOKIES



- **Small memory**
 - Used to store small information, such as the preferred language
- **Temporary**
 - Cookies have an expiration date, after which they are deleted by the browser
- **Are sent along with every HTTP request**
- Usually stored as plain text and only allow data in the form of text
- Maximum size of usually only 4kB
- Number is limited per domain to mostly max. 20
- Can be deactivated / deleted by the user

```
document.cookie = "username=John Doe; expires=Thu, 28 Aug 2024 12:00:00 UTC;";  
console.log(document.cookie);  
//delete cookie  
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC;";
```

The screenshot shows the Chrome DevTools Application tab. In the left sidebar, under 'Storage', the 'Cookies' section is expanded, showing a cookie for the domain 'localhost:63342' with the name 'username' and value 'John Doe'. The main pane displays the following table:

Name	Value	Domain	Path	Expires / Max-Age
username	John Doe	localhost	/wdp3	2024-08-28T12:00:00

Select a cookie to preview its value.

LOCAL STORAGE

- **Key/value** pair memory
- Can only store **plain text**
- Memory size up to **max 5MB**
- **Unlimited**
 - Data in LocalStorage has no expiration date
- **Synchronous**
 - Blocks the main thread when read / write (use with care)
- **LocalStorage-API**
 - `setItem(key,value)` : Add a Key/Value pair
 - `getItem(key)`: Get the value to a certain key
 - `removeItem(key)`: remove Key/Value pair
 - `clear()`: Delete all entries



```
// insert into Local Storage
localStorage.setItem('name', 'John Doe');
// or
localStorage.name = 'John Doe';
// read from Local Storage
const name = localStorage.getItem('name');
// or
const name = localStorage.name;
// delete item from Local Storage
localStorage.removeItem('name');
// delete all entries in Local Storage
localStorage.clear();
```



SESSION STORAGE

- Similar to Local Storage, only plain text
- Limited to the duration of a browser session, after which data is deleted
- Memory size up to **max 5MB**
- **SessionStorage-API**
 - `setItem(key,value)` : Add a Key/Value pair
 - `getItem(key)`: Get the value to a certain key
 - `removeItem(key)`: remove Key/Value pair
 - `clear()`: Delete all entries

```
// insert into Local Storage
sessionStorage.setItem('name', 'John Doe');
// or
sessionStorage.name = 'John Doe';
// read from Local Storage
const name = sessionStorage.getItem('name');
// or
const name = sessionStorage.name;
// delete item from Local Storage
sessionStorage.removeItem('name');
// delete all entries in Local Storage
sessionStorage.clear();
```

OBJECTS IN SESSION/LOCAL STORAGE

- Convert Array/Object to string
 - `JSON.stringify(...)`
- Convert text to object
 - `JSON.parse(...)`

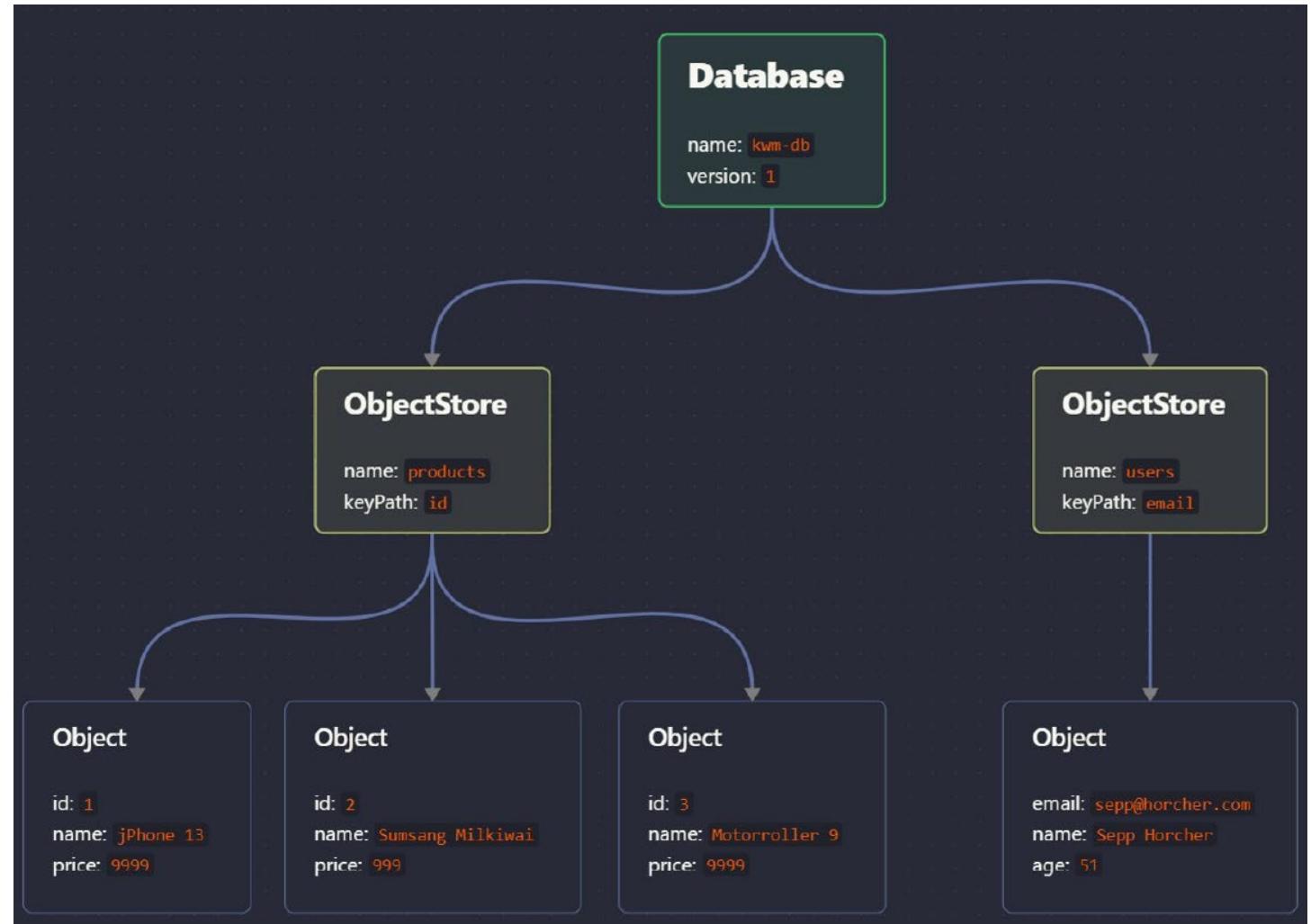
```
const myObject = { name: 'John Doe', age: 25 };
const myObjectAsString = JSON.stringify(myObject);
const myArray = [1, 2, 3, 4, 5];
const myArrayAsString = JSON.stringify(myArray);

localStorage.myObject = JSON.stringify(myObject);
const mySavedObject = JSON.parse(localStorage.myObject);
```



INDEXED DB

- Local **NoSQL** database in the browser
- **Non-relational** database
- Consists of **ObjectStores** (~Tables)
- Stores can contain **objects** (~Rows)
- Objects are sorted according to their **key** in ascending order and are retrieved via this key



INDEXED DB

- Can save and read complex JavaScript objects directly
- **Large capacity**
 - Up to several GBs depending on the system
- **High speed**
 - Uses indexes for high-performance search
- **Asynchronous**
 - Block operations
 - Main thread not (callback based)
- **Same Origin Policy**
 - Security protocol ensures that data is only available under the same domain (origin)
- **Transactions**
 - Complex data operations can be executed atomically to ensure data consistency
- **Complexity**
 - Usage is more complex compared to other storage options

#	Key (Key path: "id")	Value
0	1	<pre>{id: 1, name: 'jPhone13', price: 999} id: 1 name: "jPhone13" price: 999</pre>
1	2	<pre>{id: 2, name: 'Sumsang Milikwai', price: 999} id: 2 name: "Sumsang Milikwai" price: 999</pre>
2	3	<pre>{id: 3, name: 'Motorroller 9', price: 99} id: 3 name: "Motorroller 9" price: 99</pre>

WORKING WITH INDEXED DB

- Basic steps to work with IndexedDB:

- open database

```
window.indexedDB.open(dbName, dbVersion);
```

- create

```
db.createObjectStore(objectStorageName, { keyPath: "id" });
```

- start transaction (request / insert / delete objects)

```
request = db.transaction([objectStorageName], "readwrite").objectStore(objectStorageName).put(object);
request = db.transaction([objectStorageName], "readwrite").objectStore(objectStorageName).delete(key);
request = db.transaction([objectStorageName], "readonly").objectStore(objectStorageName).get(key);
request = db.transaction([objectStorageName], "readonly").objectStore(objectStorageName).getAll();
```

- wait until the result is available (asynchronous)

```
request.onsuccess = (e) => {
  const result = e.target.result; // do something with the result
};
```

EXAMPLE – OPEN DATABASE

```
class MyIndexDB {
  constructor(objectStoreName, keyPath= 'id', dbName = 'wdp-db', dbVersion = 1) {
    this.objectStoreName = objectStoreName;
    this.db = null; // Reference to the db
    // Open the DB
    const indexedDB = window.indexedDB || window.mozIndexedDB || window.webkitIndexedDB || window.msIndexedDB;
    const requestOpen = indexedDB.open(dbName, dbVersion);
    requestOpen.onsuccess = (e) => {
      this.db = e.target.result; // Save the db reference for later use
      console.log('DB opened', e);
    };
    requestOpen.onerror = (e) => {
      console.log('Error opening DB', e);
    };
    requestOpen.onupgradeneeded = (e) => {
      const db = e.target.result;
      // Define Data Schema here (which object stores / indices / keys)
      const objectStore = db.createObjectStore(this.objectStoreName, { keyPath: keyPath });
    }
  }
}
```

EXAMPLE – READ/SAVE OBJECTS

```
getAll() {
  return new Promise((resolve, reject) => {
    if(!this.db) reject('DB not open');
    let transaction = this.db.transaction(this.objectStoreName, 'readonly');
    let objectStore = transaction.objectStore(this.objectStoreName);
    let request = objectStore.getAll();
    request.onsuccess = (e) => {
      resolve(e.target.result);
    };
    request.onerror = (e) => {
      reject(e);
    };
  });
}

insert(object) {
  return new Promise((resolve, reject) => {
    if(!this.db) reject('DB not open');
    let transaction = this.db.transaction(this.objectStoreName, 'readwrite');
    let objectStore = transaction.objectStore(this.objectStoreName);
    let request = objectStore.put(object);
    request.onsuccess = (e) => {
      resolve(e.target.result);
    };
    request.onerror = (e) => {
      reject(e);
    };
  });
}
```

DEXIE.JS

- Dexie is a minimalistic Wrapper to IndexedDB
- Based on Promises

```
const db = new Dexie('MyDatabase');

// Declare tables, IDs and indexes
db.version(1).stores({
    friends: '+id, name, age'
});

// Find some old friends
const oldFriends = await db.friends
    .where('age').above(75)
    .toArray();

// or make a new one
await db.friends.add({
    name: 'Camilla',
    age: 25,
    street: 'East 13:th Street',
    picture: await getBlob('camilla.png')
});
```