Web Application Development
(LTAT.05.004)

JAVASCRIPT - IIII

MOHAMAD GHARIB
UNIVERSITY OF TARTU
What can JavaScript do? - Recap

• Changes all HTML elements and attributes in the page;
• Changes all the CSS styles in the page;
• Creates new HTML elements and attributes;
• Removes existing HTML elements and attributes;
• Creates new events in the HTML page;
• Reacts to existing events in the HTML page.

To do that, we need to learn the following:
• DOM nodes inter-relationships;
• DOM data types;
• DOM methods and properties;
• Locating DOM elements;
• Accessing, modifying and removing DOM elements/ CSS properties;
• Creating, appending, and removing DOM elements;
• DOM traversing;
• DOM events and event listeners.
The nodes in the DOM have a **hierarchical relationship** to each other. The terms parent, child, and sibling are used to describe the relationships.

- In a node tree, the top node is called the root (or **root node**).
- Every **node** has exactly one **parent**, except the root (which has no parent).
- A **node** can have any number of **children**.
- **Siblings** are nodes with the same **parent**.
JavaScript - IIII
Before we start ...

What do you need to know before starting?
• HTML, CSS and Document Object Model.

What we will cover:
• What JavaScript is,
• JavaScript data types, variables, loops, etc.
• How we can use JavaScript to dynamically create, update our page content, etc.

What can JavaScript do?
JavaScript can dynamically modify/manipulate an HTML page, validate user input (Form validation), react to events and user input (special effects), basic math calculations, etc.

How this lecture contributes to the course

HTML
The structure of the webpage

CSS
The styling of the webpage

JavaScript
Makes a webpage interactive
Synchronous vs Asynchronous
Synchronous is known as a blocking architecture. As a single-thread model, it follows a strict set of sequences, which means that operations are performed one at a time. While one operation is being performed, other operations’ instructions are blocked. The completion of the current task triggers the next, and so on.
Synchronous is known as a blocking architecture.

As a single-thread model, it follows a strict set of sequences, which means that operations are performed one at a time.

While one operation is being performed, other operations’ instructions are blocked.

The completion of the current task triggers the next, and so on.

```javascript
console.log("Synchronous");
console.log("Task 1"); // Task 1
console.log("Task 2"); // Task 2
window.alert("Synchronous operation"); // wait
console.log("Task 3"); // Task 3
console.log("Task 4"); // Task 4
```
Asynchronous is a non-blocking architecture, which means it does not block execution while one or more operations are in progress.

With asynchronous programming, multiple related operations can run concurrently without waiting for other tasks to complete.
Synchronous vs Async. programming

Asynchronous is a non-blocking architecture, which means it does not block execution while one or more operations are in progress.

With asynchronous programming, multiple related operations can run concurrently without waiting for other tasks to complete.

console.log("Asynchronous");
console.log("Task 1");
console.log("Task 2");
setTimeout(() => console.log("Asynchronous operation"), 0); // It will not wait, it will continue executing other statements. Then, get back to it to execute it.
console.log("Task 3");
console.log("Task 4");

// Task 1
// Task 2
// Task 3
// Task 4
// Asynchronous operation
Asynch. and synch. methods both offer advantages:

**Synchronous** programming is much easier to code. It is well supported among all programming languages, and as the default programming method.

**Asynchronous** programming decrease unnecessary the lag time between when a function is called and when the value of that function is returned.
JavaScript is single-threaded, like sync, but also non-blocking, like async.

Why do we need both Synch/Asynch?

Ex. Sell transaction, money/payment transaction - Synch.
Ex. for fetching data - Asynch

How Synch/Asynch is implemented in JavaScript?
A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete a task.

```javascript
let calledFun = () => {
    console.log('Hello');
}

function synchCallBck(string, callback) {
    callback();
    console.log(string);
}

synchCallBck('World', calledFun)
// Hello
// World
While **callbacks** are important to understand, they can lead to something **callback hell**.

```javascript
setTimeout(() => {
    console.log("CB1 - Do something");
    setTimeout(() => {
        console.log("CB2 - Do another thing");
        setTimeout(() => {
            console.log("CB3 - Do another thing");
            setTimeout(() => {
                console.log("CB4 - Do a final thing");
            }, 1000);
        }, 1000);
    }, 1000);
}, 1000);
```

A good reference for understanding and avoiding callback hell [http://callbackhell.com/](http://callbackhell.com/)
Promises
Promises

A promise is a proxy for a value not necessarily known when the promise is created.

A promise allows you to associate handlers with an asynchronous action's eventual success value or failure reason.

A promise states:

- **pending**: initial state, neither fulfilled nor rejected.
- **fulfilled**: the operation was completed successfully.
- **rejected**: the operation failed.

```javascript
let myPromise = new Promise(function(myResolve, myReject) {
    // "Producing Code" (May take some time)
    myResolve(); // when successful
    myReject(); // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise
    .then(function(value) {
        /* code if successful */
    })
    .catch(function(error) {
        /* code if some error */
    })
    .finally(function() {
        /* code to be executed in any case */
    });
```
Promises

Promises have the **then**, **catch** and **finally** methods for doing different things depending on the outcome of a promise.

**then:** when a promise is **successful**, you can then use the **resolved data**.

**catch:** when a promise **fails**, you catch the **error**, and do something with the **error information**.

**finally:** when a promise **settles** (fails or passes), you can finally do something.

```javascript
let myPromise = new Promise(function(myResolve, myReject) {
  // "Producing Code" (May take some time)
  myResolve(); // when successful
  myReject(); // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise
  .then(function(value) {
    /*code if successful*/
  },
  function(error) {
    /*code if some error*/
  }
  .finally(function() {
    /*code to be executed in any case*/
  });
```
Promises

Promises Chaining

A common need is to execute two or more asynchronous operations, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step.

We accomplish this by creating a promise chain.

doSomething()
  .then(
    function(result){
      return doSomethingElse(result);
    })
  .then(
    function(newResult){
      return doAnotherthing(newResult);
    })
  .then(
    function(finalResult){
      return finalResult;
    })
  .catch(
    function(error){/*code if some error*/}
  )
  .finally(
    function(){/*code to be executed in any case*/}
  );
Promises

Promises Chaining

A common need is to execute two or more asynchronous operations, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step.

We accomplish this by creating a promise chain.

```
doSomething()
 .then(
   function(result){
     return doSomethingElse(result);
   }
 )
 .then(
   function(newResult){
     return doAnotherthing(newResult);
   }
 )
 .then(
   function(finalResult){
     return finalResult;
   }
 )
 .catch(
   function(error){ /*code if some error*/}
 )
 .finally(
   function(){ /*code to be executed in any

Important: Always return results, otherwise callbacks won't catch the result of a previous promise. If the previous handler started a promise but did not return it, there's no way to track its settlement anymore, and the promise is said to be "floating".
```
Unlike passed-in callbacks, a promise will never be called before the completion of the current run of the JavaScript event loop.

Callbacks added with `then()` after the success or failure of the asynchronous operation, will be called, as above.

Multiple callbacks may be added by calling `then()` several times.

Each callback is executed one after another, in the order in which they were inserted.

doSomething() .then( function(result){ return doSomethingElse(result); } ) .then( function(newResult){ return doAnotherthing(newResult); } ) .then( function(finalResult){ return finalResult; } ) .catch( function(error){/*/code if some error*/} ) .finally( function(){/*/code to be executed in any case*/} );
JavaScript Call Stack, Event loop, Callback and Job queues
JavaScript can do one single thing at a time because it has only one call stack. The call stack is a mechanism that helps the JavaScript interpreter to keep track of the functions that a script calls.
Every time a script or function **calls a function**, it is added to the **top of the call stack**.

```javascript
function add(a, b) {
    return a + b;
}
function average(a, b) {
    return add(a, b) / 2;
}
let x = average(10, 10);
```
Call stack

```javascript
function add(a, b) {
  return a + b;
}

function average(a, b) {
  return add(a, b) / 2;
}

let x = average(10, 10);
```

Every time a script or function calls a function, it is added to the top of the call stack.
Every time a script or function calls a function, it is added to the top of the call stack.
A function either **exits** through a **return statement** or by reaching the **end of the scope**. The order in which the *stack* processes each function call follows the **LIFO** principle (Last In, First Out).
A function either **exits** through a **return statement** or by reaching the end of the scope. The order in which the **stack** processes each function call follows the **LIFO** principle (Last In, First Out).
A function either **exits** through a **return statement** or by reaching the **end of the scope**. The order in which the **stack** processes each function call follows the **LIFO** principle (**Last In, First Out**).
Call stack, event loop, callback queue

```javascript
function add(a, b) {
    return a + b;
}

function average(a, b) {
    return add(a, b) / 2;
}

setTimeout(() => console.log('a'), 0);
let x = average(10, 10);
setTimeout(() => console.log('b'), 0);
```
Although JavaScript could only do one thing at a time, it can still do things concurrently in the browser (Browser/Web API). If your statement is asynchronous: `setTimeout()`, `ajax()`, `promise`, or `click event`, then, that code gets forwarded to “Browser/Web API”, which is responsible for moving asynchronous code to callback/event queue after specified time.
Callback Queue: where asynchronous code gets pushed to, and waits for the execution.

Event Loop: run continuously and checks the Call stack, if it does not has any code to run (empty), then it checks the Callback queue, if Callback queue has codes to execute then it pops the message from it to the Call stack for the execution.
Function `add(a, b)`:

```javascript
function add(a, b) {
    return a + b;
}
```

Function `average(a, b)`:

```javascript
function average(a, b) {
    return add(a, b) / 2;
}
```

Set `setTimeout()` to `console.log('a'), 0`;

Set `x = average(10, 10)`;

Set `setTimeout(() => console.log('b'), 0)`;

**Callback Queue**: where asynchronous code gets pushed to, and waits for the execution.

**Event Loop**: run continuously and checks the Call stack, if it does not have any code to run (empty), then it checks the Callback queue, if Callback queue has codes to execute then it pops the message from it to the Call stack for the execution.
**Callback Queue:** where asynchronous code gets pushed to, and waits for the execution.

**Event Loop:** run continuously and checks the Call stack, if it does not has any code to run (empty), then it checks the Callback queue, if Callback queue has codes to execute then it pops the message from it to the Call stack for the execution.
**Call stack, event loop, callback queue**

```javascript
function add(a, b) {
    return a + b;
}

function average(a, b) {
    return add(a, b) / 2;
}

setTimeout(() => console.log('a'), 0);
let x = average(10, 10);
setTimeout(() => console.log('b'), 0);
```

**Callback Queue:** where asynchronous code gets pushed to, and waits for the execution.

**Event Loop:** run continuously and checks the **Call stack**, if it does not have any code to run (empty), then it checks the Callback queue, if Callback queue has codes to execute then it **pops** the message from it to the **Call stack** for the **execution**.
If Callback queue has codes to execute then it **pops** the message from it to the **Call stack** for the execution.

**FIFO** (First-In-First-Out) approach
Call stack, event loop, callback queue

```javascript
function add(a, b) {
    return a + b;
}

function average(a, b) {
    return add(a, b) / 2;
}

setTimeout(() => console.log('a'), 0);
let x = average(10, 10);
setTimeout(() => console.log('b'), 0);
new Promise((resolve, reject) => {
    resolve();
}.then(() => { console.log('c'); }));
```

If Callback queue has codes to execute then it **pops** the message from it to the Call stack for the **execution**.

**FIFO** (First-In-First-Out) approach
**Call stack, event loop, call/job queues**

```javascript
function add(a, b) {
    return a + b;
}

function average(a, b) {
    return add(a, b) / 2;
}

setTimeout(() => console.log('a'), 0);
let x = average(10, 10);
setTimeout(() => console.log('b'), 0);

new Promise((resolve, reject) => {
    resolve();
}.then(() => { console.log('c'); }));
```

**Job Queue:** browsers have introduced one more queue which is “Job Queue”, reserved only for new `Promise()` functionality.

The `.then` methods are added to **Job Queue** once the promise has returned/resolved, and then gets executed.
Call stack, event loop, call/job queues

```javascript
function add(a, b) {
    return a + b;
}

function average(a, b) {
    return add(a, b) / 2;
}

setTimeout(() => console.log('a'), 0);
let x = average(10, 10);
setTimeout(() => console.log('b'), 0);
new Promise((resolve, reject) => {
    resolve();
    .then(() => { console.log('c'); });
});
```

**Job Queue**: browsers have introduced one more queue which is "Job Queue", reserved only for new `Promise()` functionality.

The `.then` methods are added to **Job Queue** once the promise has returned/resolved, and then gets executed.
console.log('Task 1: Sync');
setTimeout(function() {
    console.log('Task 2: 1st callback - setTimeout');
}, 0);
var promise = new Promise(function(resolve, reject) {
    resolve();
});
promise
    .then(function(resolve) {
        console.log('Task 3: 1st Promise');
    })
    .then(function(resolve) {
        console.log('Task 4: 2nd Promise');
    });
console.log('Task 5: Sync');
console.log('Task 1: Sync');
setTimeout(function() {
    console.log('Task 2: 1st callback - setTimeout');
}, 0);
var promise = new Promise(function(resolve, reject) {
    resolve();
});
promise
    .then(function(resolve) {
        console.log('Task 3: 1st Promise');
    })
    .then(function(resolve) {
        console.log('Task 4: 2nd Promise');
    });
console.log('Task 5: Sync');

// Task 1: Sync
// Task 5: Sync
// Task 3: 1st Promise
// Task 4: 2nd Promise
// Task 2: 1st callback - setTimeout
AJAX
(Asynchronous JavaScript and XML)
AJAX (Asynchronous JavaScript + XML) is not a technology in itself, but it is a term coined in 2005 by Jesse James Garrett, that describes a "new" approach to using a number of existing technologies together, including HTML or XHTML, CSS, JavaScript, DOM, XML, and most importantly the XMLHttpRequest object.

When these technologies are combined in the AJAX model, web applications are able to:

- Making background HTTP requests after the page is loaded using JavaScript.
- Sending data to a web server - in the background.
- Handling the response of those HTTP requests with JavaScript.
- Updating a webpage without reloading it, i.e., no page refresh necessary.
Client:
1. Create XMLHttpRequest; and 2. Send XMLHttpRequest.

Server:
1. Receive XMLHttpRequest; and 2. Create and send a response.
**Example:** any webpage: **inspect** -> Network tab. Then, reload the page. How many **xhr requests** can you see? What are these requests?
Go to Google: **inspect -> Network**

Keep your eyes of the **Network tab** while entering a word letter by letter in the search bar, **what is happening?**
All modern browsers support the XMLHttpRequest object.

With the XMLHttpRequest object you can define a function to be executed when the request receives an answer.

The function is defined in the onreadystatechange property of the XMLHttpRequest object.

```javascript
let xhttp = new XMLHttpRequest();

xhttp.onreadystatechange = function() {
    if(this.readyState === 4 &&
        this.status === 200) {
        // code here
        console.log(this.responseText);
    }
}

xhttp.open("GET", "URI");

xhttp.send();
```
The XMLHttpRequest Object

Readystate property

The `readyState` property holds the status of the XMLHttpRequest.

The `onreadystatechange` property defines a function to be executed when the `readyState` changes, i.e., it is called every time the `readyState` changes.

`readyState` status

0: request not initialized
1: server connection established
2: request received
3: processing request
4: request finished and response is ready
The XMLHttpRequest Object

Readystate property

The `readyState` property holds the status of the `XMLHttpRequest`.

The `onreadystatechange` property defines a function to be executed when the `readyState` changes, i.e., it is called every time the `readyState` changes.

**readyState status**

0: request not initialized  
1: server connection established  
2: request received  
3: processing request  
4: request finished and response is ready

```javascript
let xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    console.log(this.readyState)
}
```
The XMLHttpRequest Object – Week 1

Response codes classes:

1XX - Informational
2XX - Successful
3XX - Redirection
4XX - Client Error
5XX - Server Error

Common response codes:

200 – OK
301 - Moved to new URL
304 – Not modified (Cached version)
400 - Bad Request
401 - Unauthorized
403 - Forbidden
404 - Not found
500 - Internal Server Error
502 - Bad Gateway
503 - Service Unavailable

HTTP response status codes indicate the status of an HTTP.
AJAX & JSON - an example of use

```javascript
let xhttp = new XMLHttpRequest();
xhttp.open("GET", "https://jsonplaceholder.typicode.com/posts");
xhttp.send();
xhttp.onreadystatechange = function() {
  if (this.readyState === 4 && this.status === 200) {
    let posts = JSON.parse(this.responseText);
    for (let i = 0; i < posts.length; i++) {
      let div = document.createElement("div");
      div.className = 'post';
      let postTitle = document.createElement("h3");
      postTitle.innerText = posts[i].title;
      div.appendChild(postTitle);
      document.body.appendChild(div);
    }
  }
};
```
Fetch API
Fetch API

The **Fetch API** provides an interface for fetching resources.

**Fetch API** will seem familiar to anyone who has used XMLHttpRequest, but the new API provides a more powerful and flexible feature set.

The **`fetch()` method** takes one mandatory argument, the path to the resource you want to fetch.

It returns a **Promise** that resolves to the **Response** to that request.

```javascript
fetch('URI')
  .then((response) => response.json())
  .then((data) => console.log(data));
```

No need for XMLHttpRequest anymore...
The **Response** object does not directly contain the actual **JSON response body** but is instead a representation of the entire **HTTP response**.

To extract the **JSON body** content from the **Response** object, we use the **json()** method, which returns a **second promise** that resolves with the result of parsing the response body text as JSON.

```javascript
fetch('URI')
  .then((response) => response.json())
  .then((data) => console.log(data));
```

**No need for XMLHttpRequest anymore...**
```javascript
fetch('https://jsonplaceholder.typicode.com/posts')
    .then((response) => response.json())
    .then(json => {
        for (let i = 0; i < json.length; i++) {
            let div = document.createElement("div");
            div.className = 'post';
            let postTitle = document.createElement("h3");
            postTitle.innerText = json[i].title;
            div.appendChild(postTitle);
            document.body.appendChild(div);
        }
    })
```
JSON
(JavaScript Object Notation)
JavaScript Object Notation (JSON) is a lightweight data-interchange format.

JSON is plain text written in JavaScript object notation.

JSON is especially useful for JavaScript-based apps, including websites and browser extensions.

Although JSON closely resembles a subset of JavaScript syntax, many programming languages support JSON.

```json
{
    "name": "John",
    "age": 21,
    "city": "Tartu"
}
```
JSON Syntax

- Data is in name/value (key/value) pairs format.
- Names require double quotes.
- Data is separated by commas.
- Curly braces hold objects.
- Square brackets hold arrays.

```json
{
    "name": "John",
    "age": 21,
    "city": "Tartu"
}
```
JSON Data Types

A string
A number
An object (JSON object)
An array
A boolean
A null

JSON values cannot be one of the following data types:
- a function
- a date
- undefined

```json
{
    "fName": "John",
    "mName": null,
    "lName": "Doe",
    "age": 32,
    "maneger": true,
    "address": {
        "country": "Estonia",
        "city": "Tartu",
        "street": "Narva mnt 18"
    },
    "experience": ["HTML", "CSS"]
}
```
A **string** must be written in double quotes.

A **number** must be an integer or a floating point.

A **boolean** can be true/false.

An **object (JSON object)** are written inside curly braces, and can contain multiple name/value pairs.

An **array** are written inside square brackets.

A **null** is a special value that can be set on any type of data including arrays, objects, number and boolean types.

```json
{
    "fName": "John",
    "mName": null,
    "lName": "Doe",
    "age": 32,
    "maneger": true,
    "address": {
        "country": "Estonia",
        "city": "Tartu",
        "street": "Narva mnt 18"
    },
    "experience": ["HTML", "CSS"]
}
```
A common use of JSON is exchanging data from a client to a webserver, and visa versa. When exchanging data between a browser and a server, the data can only be text.
A common use of JSON is **exchanging** data from a client to a server, and visa versa. When exchanging data between a **browser** and a **server**, the data can only be **text**.

When receiving **data** from a **web server**, the data is always a **string**. Parsing the data with **JSON.parse**() converts its type from string to **JS object**.

```javascript
let obj = JSON.parse('{ "name": "John", "age": 21, "city": "Tartu"}');
```
A common use of JSON is **exchanging** data from a client to a webserver, and visa versa. When exchanging data between a **browser** and a **server**, the **data can only be text**.

```javascript
let obj = JSON.parse('{ "name": "John", "age": 21, "city": "Tartu"}');

let myJson = JSON.stringify(obj);
```
**JSON vs XML**

- Both **JSON** and **XML** are "self describing".
- Both **JSON** and **XML** are hierarchical.
- Both **JSON** and **XML** can be parsed and used by lots of programming languages.
- Both **JSON** and **XML** can be fetched with an XMLHttpRequest.
- **JSON** does not use end tag.
- **JSON** is shorter.
- **JSON** is quicker to read and write.
- **JSON** can use arrays.
JSON

Try it yourself …
Online JSON editor

A JSON Editor [https://jsoneditoronline.org/](https://jsoneditoronline.org/)
Can be very useful in helping you verifying the correctness of your JSON data
Online JSON store

A JSON store (API) [https://myjson.dit.upm.es/](https://myjson.dit.upm.es/)
Can be very useful in helping you to store your JSON document online.
Extra reading/exercises

• W3Schools is a free educational website for learning to code online. With their "Try it Yourself" editor, you can edit the CSS code and view the result immediately: https://www.w3schools.com/js
Announcement

Next lecture (21/10/2022) will be in room 1025 instead of 1037 because it is not available due to maintenance. Room 1025 is small (max 30 students), but like always the lecture will be streamed and recorded.

Lecturer on Friday 28 October (Vue.js workshop) – Guest lecturer(s)

The workshop will be in two parts:

• Part 1 - 12:15-13:45 - (Room 1037); and
• Part 2 - 14:15-15:45 - (Room 1037): this part is optional, but it is also very recommended to attend it.
Thank You for your attention

Mohamad Gharib
mohamad.gharib@ut.ee

unitartu
tartuuniversity