



JavaScript

1	Basics of JavaScript	4
1.1	Expressions (lauseke)	4
1.2	Type (tietotyyppi)	4
1.2.1	Primitive types (alkeistietotyytit)	4
1.2.1.1	Boolean	4
1.2.1.2	Null	5
1.2.1.3	Undefined	5
1.2.1.4	Number	5
1.2.1.5	String	5
1.2.1.6	Symbol	7
1.2.2	Object types (oliotietotyytit)	7
1.2.3	Type conversion	7
1.2.3.1	Type coercion	7
1.2.3.2	String to number	7
1.2.3.3	Number to string	8
1.3	Variables	8
1.3.1	Var	8
1.3.2	Let	8
1.3.3	Const	8
1.3.4	Scope	8
1.3.4.1	Global scope	8
1.3.4.2	Function scope	8
1.3.4.3	Block scope	8
1.4	Objects (oliot)	9
1.4.1	Object	9
1.4.2	Properties	9
1.4.2.1	Property shorthands	9
1.4.2.2	Property methods	10
1.4.3	Object literals	10
1.4.4	Common object methods, functions and properties	10
1.4.5	Constructors	11
1.4.6	Destructuring objects and arrays	11
1.4.6.1	'This'- keyword	11
1.4.7	Prototypes	13
1.4.7.1	Prototype chain	13
1.4.7.2	Constructor prototype	13
1.4.7.3	Prototypal inheritance	13
1.4.8	Object inheritance	13
1.4.9	Array object	14
1.4.9.1	Common methods for Arrays	14
1.4.9.2	High-order array methods	14
1.4.9.3	Array concatenation and spread	16
1.4.9.4	Array-like objects	16
1.4.10	Arguments object	16
1.4.11	Math object	17
1.4.12	Date object	17
1.4.12.1	Common date methods	17
1.4.12.2	Timezone and date objects	17
1.4.12.3	Event object	17



23.11.2023 15.21.00

1.4.12.4	event.target	17
1.4.13	Promises.....	18
1.4.14	JSON	21
1.5	Classes	22
1.5.1	Class declaration	22
1.5.2	Class inheritance	22
1.5.3	Set and Get methods (setters and getters).....	22
1.6	Operators and operands	23
1.6.1	Operands.....	23
1.6.2	Binary operators	23
1.6.3	Unary operators.....	23
1.6.4	Comparison operators	23
1.6.4.1	Equality	23
1.6.4.2	Strict equality.....	24
1.6.4.3	Other comparison operators	24
1.6.5	Logical operators.....	24
1.6.5.1	Logical assignment	24
1.6.6	Assignment operators.....	24
1.6.7	Spread operator	25
1.6.8	in -operator	25
1.7	Statements (lause)	26
1.7.1	Conditional statements.....	26
1.7.1.1	If, else if, else	26
1.7.1.2	Switch statement	26
1.7.2	Loop statements	27
1.7.2.1	While- loop.....	27
1.7.2.2	For- loop.....	27
1.7.3	With- statement.....	29
1.7.4	Try and catch (error handling)	29
1.8	Functions	30
1.8.1	Basic function syntax	30
1.8.2	Function expressions (funktiolauseke)	30
1.8.3	Function parameters.....	30
1.8.3.1	Objects and arrays as parameters	30
1.8.3.2	Rest parameter (using spread operator).....	30
1.8.3.3	Arrow functions	31
1.8.4	Inner functions.....	31
1.8.4.1	Closures.....	32
1.8.4.2	Immediately invoked function expression (IIFE)	32
1.8.5	Callback functions	32
1.8.6	Function object	33
1.8.6.1	Arguments object.....	33
1.8.6.2	Function properties and methods	33
1.8.6.3	Functions inside data structures	33
1.8.7	Execution context	33
1.8.7.1	Memory creation and Execution phases.....	34
1.8.7.2	Call stack	35
1.9	Asynchronous JavaScript	35
2	BOM (Browser Object Model)	36
3	DOM (Document Object Model).....	36
3.1	Document object properties and methods	36
3.1.1	Example document object properties.....	36



23.11.2023 15.21.00

3.1.2	DOM Selectors (document object methods)	37
3.1.2.1	Query Selector	37
3.2	DOM Node types	37
3.2.1	DOM Node relationships	38
3.3	DOM Elements	39
3.3.1	Element relationships	39
3.3.2	Create and append (to parent elements) elements	39
3.3.3	Inserting elements using insertAdjacentElement()	40
3.3.4	Replacing elements	41
3.3.5	Remove elements	42
3.3.6	Element text properties	42
3.3.7	Working with element classes	42
3.3.8	Working with CSS styling	43
3.4	Events	43
3.4.1	Event object	43
3.4.2	Event listeners	43
3.4.3	Mouse events	43
3.4.3.1	Event object properties for mouse events	44
3.4.4	Key events	44
3.4.4.1	Event object properties for key events	45
3.4.5	Input events	45
3.4.5.1	Event object properties for input events	45
3.4.6	Form submissions and form object	46
3.4.6.1	FormData object	46
3.4.6.2	Form HTML	46
3.4.7	Event bubbling	47
3.4.8	Event delegation and multiple events	47
3.4.9	Page loading and window events	47
3.5	Storing information from the user	48
3.5.1	Local storage	48
3.5.1.1	Common methods	48
3.5.2	Session storage	49
3.5.3	Cookies	49
3.6	AJAX and Fetch API	49
3.6.1	XML	50
3.6.2	JSON	51
3.6.3	DOM Parser	51
3.6.3.1	Common DOM Parser methods and properties	51
3.6.4	Parsing XML data	51
3.6.5	XMLHttpRequest (XHR)	51
3.6.5.1	Common XMLHttpRequest properties	52
3.6.5.2	Common XMLHttpRequest methods	52
3.6.6	Fetch API	53
3.6.6.1	Response object	55
3.6.7	HTTP response status codes	55
4	jQuery	55
4.1	Basics of jQuery	55
4.2	Common methods	56
4.3	Common properties	57



23.11.2023 15.21.00

1 Basics of JavaScript

1.1 Expressions (lauseke)

Any statement that JavaScript engine handles. Can be only a declaration of a variable or an entity of operators and operands.

Examples:

“this is a expression”

“10 + 20”

“number1 + number2”

1.2 Type (tietotyyppi)

Variables are typeless when defined. When assigned a value, type is established. `typeof` “variable” states the type of an variable/object. When using methods on a variable that is not an object, a temporary wrapper is created to treat it as an object.

1.2.1 Primitive types (alkeistietotyyppi)

1.2.1.1 Boolean

True or false.

Falsy values in JavaScript are values that are considered “false” when evaluated in a Boolean context. They are:

- `false`: The literal `false`.
- `0`: The number zero.
- `-0`: Negative zero.
- `0n`: BigInt zero.
- `""`: An empty string.
- `null`: Represents the absence of any object value.
- `undefined`: Represents an uninitialized variable or missing property.
- `NaN`: Stands for “Not-a-Number” and represents an unrepresentable value in arithmetic.

Truthy values in JavaScript are values that are considered “true” when evaluated in a Boolean context. They are:

- `true`: The literal `true`.
- Numbers: Any non-zero number or non-zero BigInt.
- Strings: Any non-empty string.
- Objects: Including arrays, functions, and other objects.
- `[]`: An empty array (an object) is still considered truthy.
- Functions: Any defined function.
- Special Objects: Certain objects like `new Boolean(true)` or `new Number(1)` can be truthy.
- Any expression that results in a truthy value when evaluated.



23.11.2023 15.21.00

1.2.1.2 Null

Absence of any value in object or variable

1.2.1.3 Undefined

Variable has been declared, but not assigned a value

1.2.1.4 Number

Integers or floating-point numbers. Also NaN (not a number)

- Octal numbers have the prefix 0 e.g. 8 = 010.
- Hexadecimal have prefix 0X

Can be created with Number-object with

```
const x = new Number(5);
```

1.2.1.4.1 Common number and Math methods

```
const num = Number.parseInt("42");
```

```
const rounded = Math.round(3.6); // Result: 4
```

```
const floored = Math.floor(3.9); // Result: 3
```

```
const ceiled = Math.ceil(2.1); // Result: 3
```

```
const maxNum = Math.max(10, 5, 20); // Result: 20
```

```
const minNum = Math.min(10, 5, 20); // Result: 5
```

```
const randomNum = Math.random(); // Creates a pseudorandom floating-point number between 0 and 1
```

```
const squareRoot = Math.sqrt(25); // Result: 5
```

```
const power = Math.pow(2, 3); // Result: 8
```

```
let someNumber = 2.66666
```

```
someNumber.toFixed(1) // Outputs 2.7
```

1.2.1.4.2 NaN

Usually an error parsing the code. `isNaN()` function checks whether something is a number

1.2.1.5 String

Sequence of characters in single or double quotes.

Can also be created with the String-object:

```
const someString = new String("Hoi maailma")
```



23.11.2023 15.21.00

1.2.1.5.1 Common string methods

toUpperCase() = transforms all string characters to uppercase

toLowerCase() = transforms all string characters to lowercase

charAt() = returns a character at index

```
const text = "JavaScript";  
const characterAtIndex2 = text.charAt(2);  
console.log(characterAtIndex2); // Output: "v"
```

charCodeAt() = returns a character code at index

trim() = removes whitespace (spaces, tabs, line breaks) from beginning and end

```
const str = " Hello, World! ";  
const trimmedStr = str.trim();  
console.log(trimmedStr); // Output: "Hello, World!"
```

repeat() = takes argument of no. times should repeat

indexOf() = returns index of a character

substring() = returns: index, index2 substring(1,4)

replace() = replace("Maailma", "World")

includes() = returns true or false whether string is found

split() = split(" ") splits string into array with a divider specified in the arguments

startsWith("string")

endsWith("string")

String.raw = interpret string without special characters or escape characters

1.2.1.5.2 Regular Expressions

Create regex object using RegExp constructor or a literal (enclosed in "/").

```
// Using RegExp constructor  
const regex1 = new RegExp("pattern");
```

```
// Using regex literal  
const regex2 = /pattern/;
```

Use test() function to test whether a string matches the expression pattern:

```
const regex = /hello/;  
const text = "Hello, world!";  
const isMatch = regex.test(text);  
console.log(isMatch); // Output: false (case-sensitive)
```

Find all matches with match():



23.11.2023 15.21.00

```
const regex = /l/g; // 'g' flag for global matching
const text = "Hello, world!";
const matches = text.match(regex);
console.log(matches); // Output: ['l', 'l', 'l']
```

Replace text with `replace()`:

```
const regex = /world/;
const text = "Hello, world!";
const newText = text.replace(regex, "universe");
console.log(newText); // Output: "Hello, universe!"
```

1.2.1.5.3 Template literals (template strings)

Way to embed expressions and variables inside strings. Useful way to make multiline strings.

They are enclosed in backticks ``.

```
const name = "Alice";
const greeting = `Hello, ${name}!`;
```

1.2.1.6 Symbol

In ECMScript 6. unique and immutable values. They are often used as keys in objects for creating private properties. For example:

```
const uniqueID = Symbol("description");
```

1.2.2 Object types (oliotietotyytit)

Kts. [Objects](#)

1.2.3 Type conversion

1.2.3.1 Type coercion

If one of the variables in an addition [operation](#) is a string, both variables are assigned the type of string.

```
X = 5 + "5" // type of X would be a string
```

Other cases, type is converted to a number.

```
X = 5 */- "5" // type of X would be a number
```

1.2.3.2 String to number

```
const number = parseInt("42"); // Parses "42" as a decimal integer
console.log(number); // Output: 42
```



23.11.2023 15.21.00

```
const binaryNumber = parseInt("1010", 2); // Parses "1010" as a binary integer
console.log(binaryNumber); // Output: 10
```

```
const floatNumber = parseFloat("3.14"); // Parses "3.14" as a floating-point number
console.log(floatNumber); // Output: 3.14
```

1.2.3.3 Number to string

```
const number = 42;
const numberAsString = number.toString();
console.log(numberAsString); // Output: "42"
```

1.3 Variables

1.3.1 Var

Global or function scope.

1.3.2 Let

1.3.3 Const

1.3.4 Scope

1.3.4.1 Global scope

If a variable is defined without var, let or const keyword (even if inside a function), it will have global scope. This means that the variable is accessible even if it is inside a function.

1.3.4.2 Function scope

If defined with a var-keyword and inside a function, a variable will have a function scope inside that function even if a new variable would be defined inside a code block (like an if-statement). To achieve block scope, see [Block scope](#).

1.3.4.3 Block scope

Variable (defined with let or const- keywords) that is only accessible inside {}, so in a code block they were defined in.



23.11.2023 15.21.00

1.4 Objects (oliot)

Can have functions, which are functions that are properties of an object and are called methods. Almost everything in JS is an object. The document and the browser are objects too. For instance, “write” is method of the document expressed as document.write().

There are pre-built objects (kantaoliot) in JS. Most new entities of objects can be created with either using a constructor or a literal. Most common pre-built objects are:

- Array
- Object
- Date
- Math
- String
- RegExp
- Boolean
- Function

Dot notation = objectName.property

Bracket notation = objectName[property]

1.4.1 Object

Highest level of object hierarchy.

Use Object() to define a new object:

```
var car = new Object()
car.brand = "Audi"
car.model = "S4"
car.modelYear = "2006"
```

1.4.2 Properties

1.4.2.1 Property shorthands

Convenient way to create objects when the property names and variable names are the same. They allow you to simplify the object literal notation.

Example:

```
const name = "John";
const age = 30;
```

```
// Using property shorthand
const person = { name, age };
```

```
// Equivalent to:
// const person = { name: name, age: age };
```



23.11.2023 15.21.00

1.4.2.2 Property methods

User can define functions for objects they've created.

Example:

```
const person = {  
  firstName: "John",  
  lastName: "Doe",  
  fullName: function() {  
    return this.firstName + " " + this.lastName;  
  }  
};
```

1.4.3 Object literals

Use the JSON-format-like object literal:

```
var car2 = {  
  make: "Toyota",  
  model: "Camry",  
  year: 2022  
}
```

When using a dynamic property as a object value name, use bracket syntax.

1.4.4 Common object methods, functions and properties

delete objectName.property = to delete a property

Object.keys(objectNameHere) = returns the property names (keys) of object as array

Object.values() = returns the values of the keys as an array

Object.entries() = returns key value pairs as an array

Object.hasOwnProperty() = check if an object has a property with a specific name. It returns a boolean value, `true` if the object has the specified property, and `false` if it doesn't.

Object.assign() = copy the values of all enumerable properties from one or more source objects to a target object. It's often used for creating shallow copies of objects and for merging the properties of multiple objects into a single object.

```
Object.assign(target, source1, source2, ...);
```

Example:

```
const target = { a: 1, b: 2 };  
const source = { b: 3, c: 4 };  
  
// Copy properties from source to target  
Object.assign(target, source);  
  
console.log(target);
```



23.11.2023 15.21.00

```
// Output: { a: 1, b: 3, c: 4 }
```

1.4.5 Constructors

Functions that are used to create and initialize objects.

```
// Define an object constructor for a 'Person' object
function Person(firstName, lastName, age) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
}
```

```
// Create instances (objects) using the constructor
const person1 = new Person("John", "Doe", 30);
const person2 = new Person("Alice", "Smith", 25);
```

1.4.6 Destructuring objects and arrays

Extract values from objects or arrays to variables.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
};
```

```
const { firstName, lastName, age } = person;
console.log(firstName); // "John"
console.log(lastName);  // "Doe"
console.log(age);       // 30
```

Can also be used in function arguments:

```
function printPersonInfo({ firstName, lastName, age }) {
  console.log(` Name: ${firstName} ${lastName}, Age: ${age}`);
}
```

```
printPersonInfo(person); // Outputs: "Name: John Doe, Age: 30"
```

1.4.6.1 'This'- keyword

Identifier that refers to the current execution context or the current object, depending on how it is used.



23.11.2023 15.21.00

1.4.6.1.1 This in Global Context

Refers to the global object ('window' in web browser or 'global' in Node.js).

```
console.log(this === window); // In a web browser, this is true
```

1.4.6.1.2 This in Function Context

If called in a standalone function, 'this' refers to the global object ('window')

```
function myFunction() {  
  console.log(this === window); // true  
}  
myFunction();
```

If called as a method of an object 'this' refers to the object that owns the method:

```
const person = {  
  name: "John",  
  greet: function() {  
    console.log(`Hello, ${this.name}!`);  
  }  
};  
person.greet(); // Outputs: "Hello, John!"
```

1.4.6.1.3 This in Arrow Functions

Arrow functions do not have their own 'this'. They inherit the 'this' value from their enclosing (lexical) scope.

```
const myObject = {  
  greeting: "Hello",  
  sayHello: function() {  
    setTimeout(() => {  
      console.log(this.greeting); // "Hello after timeout of 1000ms/1s"  
    }, 1000);  
  }  
};  
  
myObject.sayHello();
```

1.4.6.1.4 This in Event Handlers

'This' refers to the DOM element that triggered the event.

```
document.getElementById("myButton").addEventListener("click", function() {  
  console.log(this.id); // ID of the clicked button  
})
```



23.11.2023 15.21.00

```
});
```

1.4.7 Prototypes

Fundamental mechanism that allows objects to inherit properties and methods from other objects. A way to share properties and methods among objects allowing you to create efficient and memory saving code.

1.4.7.1 Prototype chain

Every object in JS has a prototype from which they inherit properties and methods. Objects can have prototypes and those prototypes can have prototypes. If a property or method is not found on the object itself, JavaScript looks up the prototype chain until it finds it.

1.4.7.2 Constructor prototype

```
function Person(name) {  
  this.name = name;  
}  
  
Person.prototype.sayHello = function() {  
  console.log(`Hello, my name is ${this.name}`);  
};  
  
const john = new Person("John");  
john.sayHello(); // Outputs: "Hello, my name is John"
```

1.4.7.3 Prototypal inheritance

```
const parent = {  
  name: "Parent"  
};  
  
const child = Object.create(parent);  
console.log(child.name); // Outputs: "Parent"
```

1.4.8 Object inheritance

Inheriting properties and methods of already existing object.

```
const parent = {  
  greet: function() {  
    console.log("Hello, I'm the parent!");  
  }  
};  
  
const child = Object.create(parent); // Creates a new object inheriting from 'parent'  
child.greet(); // Calls the 'greet' method from 'parent'
```



23.11.2023 15.21.00

1.4.9 Array object

Data structure for “lists” of data like strings, objects, or other arrays.

Defined with the Array() object or with [] brackets: `const myArray = [1, 2, 3];`

Access array length with `myArray.length` property.

1.4.9.1 Common methods for Arrays

`Filter()` = filter arrays

`Flat()` = make arrays within arrays into the same array

`From()` = make array from string, HTMLcollection or other

`Includes()` = returns true or false

`IndexOf()` = returns index of value passed in as argument. Negative 1 is false!

`isArray()`

`Array[1] = 42;` = assign new value

`Of()` = make array from values of multiple variables

`Pop()` = remove element from the end

`Push()` = add element to end

`Reverse()`

`Shift()` = remove element from beginning

`Slice()` = (startIndex, endIndex)

`Sort()` = sort arrays

`Splice()` = like `slice()`, but modifies original array

`Unshift()` = add element to the beginning

Chain on methods `splice().reverse().toString().charAt()`

1.4.9.2 High-order array methods

1.4.9.2.1 filter()

Creates a new array by filtering out elements from the original array based on a provided callback function.

Syntax:

```
const newArray = array.filter(function(element, index, array) {  
  // Return true to keep the element, false to remove it  
});
```

Parameters:

- `element`: The current element being processed.
- `index` (optional): The index of the current element.
- `array` (optional): The array that `filter()` is being called on.

Example:



23.11.2023 15.21.00

```
const numbers = [1, 2, 3, 4, 5, 6];
```

```
const evenNumbers = numbers.filter(function(number) {  
  return number % 2 === 0; // Return true for even numbers  
});
```

```
console.log(evenNumbers); // Output: [2, 4, 6]
```

1.4.9.2.2 map()

Creates a new array by applying a provided callback function to each element of the original array.

Syntax:

```
const newArray = array.map(function(element, index, array) {  
  // Return the transformed value for each element  
});
```

Parameters:

- **element**: The current element being processed.
- **index (optional)**: The index of the current element.
- **array (optional)**: The array that `map()` is being called on.

Example:

```
const numbers = [1, 2, 3, 4, 5];
```

```
const doubledNumbers = numbers.map(function(number) {  
  return number * 2; // Double each number  
});
```

```
console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]
```

1.4.9.2.3 reduce()

Accumulating values in an array and reducing them to a single value. It iterates through the elements of an array and applies a callback function that you provide, which accumulates values into an accumulator (also called the "reducer") to produce a final result.

Syntax:

```
const result = array.reduce(function(accumulator, currentValue, index, array) {  
  // Update the accumulator based on the currentValue  
  return accumulator;  
}, initialValue);
```

Parameters:

- **accumulator**: The accumulated result of the reduction process.
- **currentValue**: The current element being processed.
- **index (optional)**: The index of the current element.
- **array (optional)**: The array that `reduce()` is being called on.
- **initialValue (optional)**: An initial value for the accumulator.



23.11.2023 15.21.00

Example:

```
const numbers = [1, 2, 3, 4, 5];
```

```
const sum = numbers.reduce(function(accumulator, currentValue) {  
  return accumulator + currentValue; // Accumulate the sum  
}, 0);
```

```
console.log(sum); // Output: 15
```

1.4.9.3 Array concatenation and spread

Concat() = combine

Spread operator = ...

Copy array:

```
const originalArray = [1, 2, 3];  
const copyArray = [...originalArray];
```

Merge arrays:

```
const array1 = [1, 2];  
const array2 = [3, 4];  
const mergedArray = [...array1, ...array2]; // [1, 2, 3, 4]
```

1.4.9.4 Array-like objects

HTMLcollections

NodeList

1.4.10 Arguments object

Arguments object contains only 1 property: length.

Usage: functionName.arguments.length tells the number of arguments passed to the function
Is an array-like object that holds all the arguments/values/parameters passed to a function.

Access it inside function with

arguments[indexForTheArgument]

```
function add() {  
  let sum = 0;  
  for (let i = 0; i < arguments.length; i++) {  
    sum += arguments[i];  
  }  
  return sum;  
}
```




23.11.2023 15.21.00

```
console.log(add(1, 2, 3)); // Outputs: 6
```

1.4.11 Math object

See [Common number methods](#).

1.4.12 Date object

Calculates dates in milliseconds after 1.1.1970

```
Var dateVar = new Date()  
Date("07-10-2023")
```

1.4.12.1 Common date methods

```
getDay()  
getFullYear()  
getHours()  
getMilliseconds()  
getMinutes()  
getMonth()  
getSeconds()  
getTime()  
now()
```

Compare dates:

```
const date1 = new Date('2023-09-30');  
const date2 = new Date('2023-10-01');  
const isDate1BeforeDate2 = date1 < date2; // true
```

1.4.12.2 Timezone and date objects

```
x = Intl.DateTimeFormat("en-US").format(dateObjectHere)
```

Can also use

```
dateObjectHere.toLocaleString()
```

1.4.12.3 Event object

More about events in chapter [Events](#)

1.4.12.4 event.target

The callback function that is the 2nd parameter of the event listener methods, takes a parameter that is the event object itself.

Example:

```
element.addEventListener('click', function(event) {  
  // Use the 'event' parameter to access event properties  
  console.log('Event type: ' + event.type);  
  console.log('Target element: ' + event.target.tagName);  
});
```



23.11.2023 15.21.00

```
console.log('Mouse X: ' + event.clientX + ', Mouse Y: ' + event.clientY);  
});
```

event.target = This property provides a reference to the HTML element that triggered the event. It is often used to access and manipulate properties of the element.

event.target.tagName = Returns the tag name of the element that triggered the event. It is often used to check the type of element.

```
document.addEventListener('click', function(event) {  
  if (event.target.tagName === 'BUTTON') {  
    console.log('A button was clicked.');  }  
});
```

event.target.id and **event.target.className** = These properties allow you to access the `id` and `class` attributes of the element that triggered the event.

```
document.addEventListener('click', function(event) {  
  if (event.target.id === 'myElement') {  
    console.log('The element with id "myElement" was clicked.');  }  
});
```

event.target.value = For form elements like input fields and textareas, this property retrieves the current value of the element.

```
Document.getElementById('myInput').addEventListener('input', function(event) {  
  console.log('Input value:', event.target.value);  
});
```

event.target.parentElement = Accesses the parent element (the element containing the event target).

```
Document.addEventListener('click', function(event) {  
  const parentElement = event.target.parentElement;  
  console.log('Parent element:', parentElement);  
});
```

event.target.classList = Provides access to the `classList` of the element, allowing you to add, remove, or check for the presence of CSS classes.

```
document.getElementById('myElement').addEventListener('click', function(event) {  
  event.target.classList.add('highlight');  
});
```

1.4.13 Promises

Information on Fetch API, see [Fetch API](#)

An object that represents the eventual completion or failure of an asynchronous operation and its resulting value. Promises are a crucial part of managing asynchronous code, providing a cleaner and more structured way to handle callbacks and avoid callback hell.

States:



23.11.2023 15.21.00

- Pending: The initial state of a promise. The operation is still in progress, and the promise hasn't been fulfilled or rejected yet.
- Fulfilled: The asynchronous operation completed successfully, and the promise has a resulting value.
- Rejected: The asynchronous operation encountered an error, and the promise has a reason for the failure.

Example:

```
let kanyeIsCrazy = false

const kanyePromise = new Promise((resolve, reject) => {
  if (kanyeIsCrazy) {
    const result = "Kanye is indeed crazy mf"
    resolve(result)
  } else {
    const reason = "He took his medicine"
    reject(reason)
  }
})

kanyePromise
  .then(kanyeState => {
    console.log(kanyeState)
    // Outputs "Kanye is indeed crazy mf"
  })
  .catch(kanyeFail => {
    console.log(kanyeFail);
    // Outputs "He took his medicine"
  })
```

Creating a promise:

Promises can be created using the `Promise` constructor. It takes a function with two parameters, `resolve` and `reject`, which are functions used to fulfill or reject the promise.

`resolve(value)`: This function is used when the asynchronous operation is successful. It transitions the promise from the "pending" state to the "fulfilled" state, and the `value` parameter represents the result of the successful operation. This value can be any JavaScript object, such as a string, number, array, or even another Promise and its value is transferred to `.then` as a parameter.

`reject(reason)`: This function is used when there's an error or failure in the asynchronous operation. It transitions the promise from the "pending" state to the "rejected" state, and the `reason` parameter represents the reason for the failure. Typically, `reason` is an instance of the `Error` class or a string providing information about the error.

```
const myPromise = new Promise((resolve, reject) => {
  // Asynchronous operation
  // If successful, call resolve(value)
```



23.11.2023 15.21.00

```
// If there's an error, call reject(reason)
});
```

Chaining:

Promises allow for chaining multiple asynchronous operations. You can use the `.then()` method to handle the fulfillment and the `.catch()` method to handle rejections. Then()-method gets the value from the resolved function as a parameter and catch() method the value from reject function.

```
myPromise
  .then((result) => {
    // Handle successful fulfillment
  })
  .catch((error) => {
    // Handle rejection
  });
```

Promise.all:

The `Promise.all()` method takes an array of promises and returns a single promise that resolves when all of the promises in the iterable argument have resolved, or rejects with the reason of the first promise that rejects.

```
const promise1 = Promise.resolve('Hello');
const promise2 = 10;
const promise3 = new Promise((resolve, reject) => {
  // Asynchronous operation
});
```

```
Promise.all([promise1, promise2, promise3])
  .then((values) => {
    console.log(values); // Array of resolved values
  })
  .catch((error) => {
    console.error(error); // Rejection reason of the first promise that rejects
  });
```

1.4.13.1.1 Async and Await:

The `async/await` syntax provides a more concise way to work with promises. The `async` keyword is used to define an asynchronous function, and `await` is used to pause the execution until the promise is settled.

```
async function myAsyncFunction() {
  try {
    const result = await myPromise;
    // Handle successful fulfillment
  } catch (error) {
    // Handle rejection
  }
}
```



23.11.2023 15.21.00

```
}  
}
```

1.4.14 JSON

More about handling JSON data in chapter: [AJAX](#)

Lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. JSON is often used to transmit data between a server and a web application, or between different parts of an application. It's widely used in web development and has become a standard for data exchange on the web.

It can contain various data types, including strings, numbers, objects, arrays, booleans, and null values.

Example:

```
{  
  "name": "John Doe",  
  "age": 30,  
  "isStudent": false,  
  "hobbies": ["reading", "gaming", "cooking"],  
  "address": {  
    "street": "123 Main St",  
    "city": "Exampleville"  
  }  
}
```

Convert JS object to JSON string:

```
const person = {  
  name: "Alice",  
  age: 30,  
};  
  
const jsonString = JSON.stringify(person);  
  
console.log(jsonString);  
{"name":"Alice","age":30}
```

Parse JSON string back to JS object:

```
const jsonString = '{"name":"Bob","age":35}';  
const person = JSON.parse(jsonString);  
console.log(person.name); // Outputs: "Bob"
```



23.11.2023 15.21.00

1.5 Classes

A way to create objects with shared properties and behaviors. Introduced in ECMAScript 6 and provide a more modern approach to defining object blueprints.

1.5.1 Class declaration

Example:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  sayHello() {
    console.log(`Hello, my name is ${this.name}.`);
  }
}
```

This creates a class called Person which contains **constructor method** for initializing object **properties** and sayHello() method.

Create new object that belongs to the Person class by:

```
const person1 = new Person("John", 30);
const person2 = new Person("Alice", 25);
```

1.5.2 Class inheritance

Inheritance allows you to create a subclass that inherits the properties and methods from a parent class. Use the **Extend** keyword.

Example:

```
class Student extends Person {
  constructor(name, age, studentID) {
    super(name, age); // Call the parent class constructor
    this.studentID = studentID;
  }

  study() {
    console.log(`${this.name} is studying.`);
  }
}
```

1.5.3 Set and Get methods (setters and getters)

Special methods used to control the access and modification of class properties. They provide a way to define custom behavior for reading and writing the values of object properties.

Setters: methods that are used to assign values to object properties. They are defined using the `set` keyword followed by the property name.



23.11.2023 15.21.00

Getters: methods that are used to retrieve the value of object properties. They are defined using the `get` keyword followed by the property name.

1.6 Operators and operands

Operators combine expressions into more complex expressions.

Example:

`let sum = 5 + 3; // '+' is the operator, '5' and '3' are operands`

1.6.1 Operands

Values that the operators operate on.

Literal values = "5", "3,14", "hello"

Variables = x, y, result etc.

Expressions = $x + y$, $a * (b - c)$

1.6.2 Binary operators

Require 2 operands. Example: $1 + 2$

$+$ = addition

$-$ = subtraction

$*$ = multiplication

$/$ = division

$\%$ = modulo (jakokäännös)

1.6.3 Unary operators

Require only one operand. Example: $++1$

$++$ = increment. Adds 1 to the value.

$--$ = decrement. Removes 1 from the value.

1.6.4 Comparison operators

1.6.4.1 Equality

$==$ = equality. Tests whether two values are equal after type coercion.

Example:

`5 == "5"; // true (string "5" is converted to a number)`

`1 == true; // true (boolean true is converted to the number 1)`



23.11.2023 15.21.00

1.6.4.2 Strict equality

=== = strict equality

Example:

`5 === "5"; // false (values are equal, but types are different)`

`1 === true; // false (types are different)`

1.6.4.3 Other comparison operators

`!=` = inequality

`!==` = strict inequality

`<=` or `>=` = less or greater than

1.6.5 Logical operators

`&&` = AND

Will return first falsy value or the last value

Can be used in variable: `var a = 10 && 20 && 30`

`||` = OR

Will return the first truthy value or the last value

`!` = NOT

1.6.5.1 Logical assignment

The `&&=` operator combines logical AND (`&&`) with assignment (`=`). It assigns the value on the right side to the variable on the left side only if the left-side variable is truthy.

`let x = 5;`

`x &&= 10; // x is assigned 10 because 5 (x) is truthy`

The `||=` operator combines logical OR (`||`) with assignment (`=`). It assigns the value on the right side to the variable on the left side only if the left-side variable is falsy.

`let y = 0;`

`y ||= 20; // y is assigned 20 because 0 (y) is falsy`

1.6.6 Assignment operators

`let x = 10; // Assigns the value 10 to the variable x`

`let y = 5;`

`y += 3; // Equivalent to y = y + 3, assigns the value 8 to y`



23.11.2023 15.21.00

```
let z = 8;  
z -= 4; // Equivalent to z = z - 4, assigns the value 4 to z
```

```
let a = 6;  
a *= 2; // Equivalent to a = a * 2, assigns the value 12 to a
```

```
let b = 16;  
b /= 4; // Equivalent to b = b / 4, assigns the value 4 to b
```

```
let c = 17;  
c %= 5; // Equivalent to c = c % 5, assigns the value 2 to c
```

```
let d = 3;  
d **= 4; // Equivalent to d = d ** 4, assigns the value 81 to d
```

1.6.7 Spread operator

Used to combine or copy arrays and objects into single array or object.

Copy:

```
const originalArray = [1, 2, 3];  
const copiedArray = [...originalArray];  
console.log(copiedArray); // [1, 2, 3]
```

Combine:

Arrays:

```
const array1 = [1, 2, 3];  
const array2 = [4, 5, 6];  
const concatenatedArray = [...array1, ...array2];  
console.log(concatenatedArray); // [1, 2, 3, 4, 5, 6]
```

Objects:

```
const object1 = { x: 1, y: 2 };  
const object2 = { z: 3 };  
const mergedObject = { ...object1, ...object2 };  
console.log(mergedObject); // { x: 1, y: 2, z: 3 }
```

1.6.8 in -operator

Used to check for the existence of a specified property in an object. It returns a boolean value, `true` if the property is found in the object, and `false` if it is not found.

Example:

```
const person = {  
  name: "John",  
  age: 30  
};
```



23.11.2023 15.21.00

```
// Check if the "age" property exists in the "person" object
const hasAge = "age" in person;
console.log(hasAge); // This will print true
```

```
// Check if the "city" property exists in the "person" object
const hasCity = "city" in person;
console.log(hasCity); // This will print false
```

1.7 Statements (lause)

; marks the end of a statement

1.7.1 Conditional statements

1.7.1.1 If, else if, else

```
if (condition) {
    // Executed if the condition is true
} else if (anotherCondition) {
    // Executed if the first condition is false and the second condition is true
} else {
    // Executed if none of the conditions are true
}
```

1.7.1.1.1 Ternary operator (conditional expression/operator)

Syntax:

```
condition ? expressionIfTrue : expressionIfFalse
```

If no else, set expressionIfFalse to "null"

1.7.1.2 Switch statement

Alternative to else if- statement structure, sometimes easier to read.

Example:

```
switch (expression) {
    case value1:
        // Code to execute if expression === value1
        break;
    case value2:
        // Code to execute if expression === value2
        break;
```



23.11.2023 15.21.00

```
// More cases can be added here
default:
  // Code to execute if none of the cases match expression
}
```

1.7.2 Loop statements

1.7.2.1 While- loop

```
let i = 0;
while (i < 5) {
  // Code to repeat (executes 5 times)
  i++;
}
```

1.7.2.1.1 Do...while- loop

Similar to the while loop, but guarantees that the code will be executed at least once before checking the condition, even if the condition is false.

```
let i = 0;
do {
  // Code to repeat (executes at least once)
  i++;
} while (i < 5);
```

1.7.2.2 For- loop

Good for iterating arrays or other data structures.

```
for (initialization; condition; update) {
  // Code to be executed in each iteration
}
```

```
for (let i = 0; i < 5; i++) {
  // Code to repeat (executes 5 times)
}
```

1.7.2.2.1 for...in- loop

Good for objects!



23.11.2023 15.21.00

Useful when working with objects and their properties. It doesn't loop through indexes, which means it doesn't guarantee that elements/items are iterated in order. Good for sparse arrays (harvoille taulukoille)

```
for (let key in object) {  
  if (object.hasOwnProperty(key)) {  
    // Code to work with object[key]  
  }  
}
```

```
const person = {  
  firstName: "Alice",  
  lastName: "Smith",  
  age: 30,  
};
```

```
for (const key in person) {  
  console.log(key + ": " + person[key]);  
}
```

1.7.2.2.2 for...of- loop

Good for arrays!

The for...of loop is used to iterate over the values of iterable objects, such as arrays, strings, maps, sets, etc.

```
for (let value of iterable) {  
  // Code to work with value  
}
```

```
const colors = ["red", "green", "blue"];
```

```
for (const color of colors) {  
  console.log(color);  
}
```

This code will print each color in the array to the console.

1.7.2.2.3 forEach() method for arrays

Does not work with HTML collections!

The forEach() method is available for arrays and node lists and is used to iterate over the elements of an array, applying a function to each element.

```
array.forEach(function(element, index, array) {  
  // Code to be executed for each element  
});
```



23.11.2023 15.21.00

Takes 3 arguments:

- `element`: The current element being processed.
- `index` (optional): The index of the current element.
- `array` (optional): The array that `forEach` is being called on.

Example:

```
const numbers = [1, 2, 3, 4, 5];

numbers.forEach(function(number, index) {
  console.log(`Element at index ${index}: ${number}`);
});
```

Result:

```
Element at index 0: 1
Element at index 1: 2
Element at index 2: 3
Element at index 3: 4
Element at index 4: 5
```

1.7.3 With- statement

Used to simplify repeated references to the properties of an object.

```
let person = {
  firstName: "John",
  lastName: "Doe",
};

with (person) {
  console.log(firstName); // "John"
  console.log(lastName); // "Doe"
}
```

NOTE! Not recommended and not available in strict mode. Can cause confusion on larger code-bases.

1.7.4 Try and catch (error handling)

Enables handling of errors. Offers an alternative thing to do when something can cause an error.

```
try {
  // Code that may cause an error
  let result = x / y;
} catch (error) {
  // catch block receives and error object that contains info about the error
  // Handle the error
  console.error("An error occurred:", error.message);
}
```



23.11.2023 15.21.00

1.8 Functions

Can be assigned to a variable or exist inside other functions as *inner functions*.

Return stops function execution.

1.8.1 Basic function syntax

```
function name(arguments) {  
    function contents  
}
```

1.8.2 Function expressions (funktio-lauseke)

Anonymous function defined in a variable.

```
const functionName = function(arguments) {  
    // Function body: code to be executed  
};
```

To call the function, use the variable name like a function: `functionName()`. If you call it without the `()`, function contents will be returned.

1.8.3 Function parameters

Are variables declared in a function's definition. Arguments are the actual values passed into the function. But these are most commonly used interchangeably.

1.8.3.1 Objects and arrays as parameters

```
function incrementAge(person) {  
    person.age++;  
}
```

```
const myPerson = { firstName: "Alice", lastName: "Smith", age: 25 };  
incrementAge(myPerson);  
console.log(myPerson.age); // Outputs: 26
```

1.8.3.2 Rest parameter (using spread operator)

Allows to having indefinite amount of arguments passed to a function. Turns them into an array.

```
function sum(...numbers) {  
    let result = 0;  
    for (let number of numbers) {  
        result += number;  
    }  
    return result;  
}
```



23.11.2023 15.21.00

```
const total = sum(1, 2, 3, 4, 5);  
console.log(total); // Outputs: 15
```

1.8.3.3 Arrow functions

Shorthand form of function expressions with unique features.

```
const functionName = (arguments) => {  
  // Function body: code to be executed  
};
```

```
Const add = (a, b) => {  
  Return a + b;  
};
```

Implicit return with shorter syntax:

```
Const add = (a, b) => a + b;
```

```
function sum(a, b) {  
  return a + b  
}  
  
let sum2 = (a, b) => a + b
```

```
function isPositive(number) {  
  return number >= 0  
}  
  
let isPositive2 = number => number >= 0
```

1.8.4 Inner functions

```
function outerFunction() {  
  function innerFunction() {  
    console.log("Inner function is called");  
  }  
}
```

```
// You can call the inner function from within the outer function  
innerFunction();  
}  
// You call the outer function to start the execution
```



23.11.2023 15.21.00

`outerFunction();`

1.8.4.1 Closures

Closures allow functions to retain access to variables from their containing (enclosing) lexical scope, even after the outer function has finished executing.

```
function outer() {  
  const outerVar = "I'm from outer!";  
  
  function inner() {  
    console.log(outerVar); // inner function has access to outerVar  
  }  
  
  return inner;  
}
```

```
const closureFunction = outer(); // outer function is invoked, and it returns inner  
closureFunction(); // inner function is invoked, still has access to outerVar
```

1.8.4.2 Immediately invoked function expression (IIFE)

Defining a function and immediately executing it.

Can be used to prevent global scope pollution if there are many global variables with same name.

```
(function () {  
  // This function is defined and immediately invoked  
  const privateVar = "I'm private!";  
  console.log(privateVar); // This logs "I'm private!"  
})();  
  
// privateVar is not accessible here  
console.log(typeof privateVar === 'undefined'); // This logs "true"
```

1.8.5 Callback functions

Asynchronous programming concept. Callback function is a function planned to execute at a certain time or when a condition is met. It is passed as an argument to another function.

```
function process(callback) {  
  // Do some work  
  callback(); // Call the provided callback function  
}  
  
function onComplete() {  
  console.log("Process completed.");  
}
```




23.11.2023 15.21.00

```
process(onComplete); // Pass onComplete as a callback
```

1.8.6 Function object

In JS, functions are a type of object and can have properties and methods like other objects.

Function() constructor can be used to create a function.

1.8.6.1 Arguments object

Kts. [Arguments object](#)

1.8.6.2 Function properties and methods

As they are objects, properties and methods can be added.

```
function greet(name) {  
  console.log(`Hello, ${name}!`);  
}
```

```
greet.message = "Welcome"; // Adding a property to the greet function  
console.log(greet.message); // Outputs: Welcome
```

1.8.6.3 Functions inside data structures

Functions can be stored inside arrays, objects or other data structures.

```
const functionArray = [greet, someOtherFunction];  
console.log(functionArray[0]("Alice")); // Outputs: Hello, Alice!
```

1.8.7 Execution context

An essential concept that helps you understand how code is executed. It's like an environment or container that holds information about the code being executed. There are three types of execution contexts in JavaScript:

Global Execution Context:

- The global execution context is the outermost context and represents the entire JavaScript program.
- It's created when your script is initially loaded or when a web page is opened.
- It contains two key components:
 - The global object (in a browser, this is `window`).
 - The `this` keyword, which refers to the global object.
- Variables and functions declared in the global scope are attached to the global object.

Function Execution Context:

- A function execution context is created whenever a function is called or invoked.
- Each function has its own execution context, which is responsible for managing the function's local variables and parameters.

23.11.2023 15.21.00

- When a function is called, a new function execution context is pushed onto the call stack, and when the function returns, its context is removed from the stack.
- Function execution contexts can be nested when functions are called within other functions.


1.8.7.1 Memory creation and Execution phases

Memory Creation Phase (Creation Phase):

- Variable declarations are hoisted and initialized with `undefined`.
- Function declarations are hoisted, and the entire function is stored.

Code Execution Phase:

- Variable values are assigned during execution.
- Functions are executed when called in the code.
- JavaScript looks up variable references in lexical (parent) scopes.
- Understanding these phases helps with debugging and scope management.



```

1 let x = 100
2 let y = 50
3 function getSum(n1, n2) {
4   let sum = n1 + n2
5   return sum
6 }
7 let sum1 = getSum(x, y)
8 let sum2 = getSum(10, 5)

```

Creation Phase:

Line 1: `x` variable is allocated memory and stores `"undefined"`

Line 2: `y` variable is allocated memory and stores `"undefined"`

Line 3: `getSum()` function is allocated memory and stores all the code

Line 7: `sum1` variable is allocated memory and stores `"undefined"`

Line 8: `sum2` variable is allocated memory and stores `"undefined"`


Execution Phase:

Line 1: Places the value of `100` into the `x` variable

Line 2: Places the value of `50` into the `y` variable

Line 3: Skips the function because there is nothing to execute

Line 7: Invokes the `getSum()` function and creates a new function execution context



```

1 let x = 100
2 let y = 50
3 function getSum(n1, n2) {
4   let sum = n1 + n2
5   return sum
6 }
7 let sum1 = getSum(x, y)
8 let sum2 = getSum(10, 5)

```

Function EC Creation Phase:

Line 3: `n1` & `n2` variables are allocated memory and stores `"undefined"`

Line 4: `sum` variable is allocated memory and stores `"undefined"`

Function EC Execution Phase:

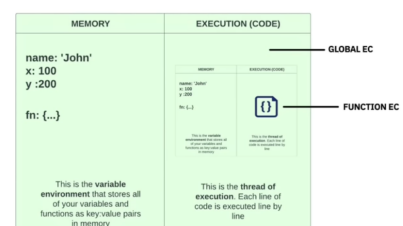
Line 3: `n1` & `n2` are assigned `100` and `50`

Line 4: Calculation is done and `150` is put into the `sum` variable

Line 5: `return` tells the function EC to return to the global EC with value of `sum` (`150`)

Line 7: Returned `sum` value is put into the `sum1` variable

Line 8: Open another function execution context and do the same thing with different parameters



1.8.7.2 Call stack

A data structure that keeps track of function calls in JavaScript. It operates on a "last in, first out" (LIFO) basis, where the most recently called function is the first to be executed and removed. This stack helps manage the order of function execution and keeps track of the current execution context.

The Call Stack

- ✓ Stack of functions to be executed
- ✓ Manages **execution contexts**
- ✓ Stacks are LIFO **last in first out**



1.9 Asynchronous JavaScript



23.11.2023 15.21.00

2 BOM (Browser Object Model)

Provides various objects and properties that allow you to interact with and control the web browser itself.

Some high level BOM objects, methods and functions:

- **window Object:** The top-level object representing the browser window, providing access to properties like `window.innerWidth` and methods like `window.open()`.
- **document Object:** Represents the HTML document loaded in the browser, offering methods for DOM manipulation like `document.getElementById()` and `document.querySelector()`.
- **navigator Object:** Provides information about the user's browser, such as `navigator.userAgent` and `navigator.language`.
- **location Object:** Contains information about the current URL and allows you to navigate to different URLs using `location.href` or `location.assign()`.
- **history Object:** Allows you to manipulate the browser's session history using methods like `history.back()` and `history.forward()`.
- **localStorage and sessionStorage Objects:** Provide storage options for web applications to store data locally on the user's device.
- **alert(), confirm(), and prompt() Methods:** Used for displaying dialog boxes to the user for alerts, confirmations, and user input.
- **setTimeout() and setInterval() Functions:** Allow you to schedule the execution of functions after a specified delay or at regular intervals.
- **XMLHttpRequest Object:** Used for making asynchronous HTTP requests to fetch data from a server.
- **screen Object:** Provides information about the user's screen, such as screen dimensions and color depth.

3 DOM (Document Object Model)

Represents the document Object loaded into the browser.

3.1 Document object properties and methods

Document object properties returns a `HTMLCollection`. It is not an array. Needs to be converted into an array to access the array object methods.

3.1.1 Example document object properties

```
Document.html
```

```
Document.body
```

```
Document.body.children;
```



23.11.2023 15.21.00

Document.URL

Document.characterSet

Document.contentType

Document.links

Document.links[0].href

Document.links[0].id = "google-link"

Document.images[0].source

3.1.2 DOM Selectors (document object methods)

Document.getElementById()

Document.getElementById().getAttributes

Document.getElementById().setAttributes

```
const myImage = document.getElementById('myImage');
```

```
// Set the 'src' attribute of an image
```

```
myImage.setAttribute('src', 'new_image.jpg');
```

Document.getElementById.title

3.1.2.1 Query Selector

Ways to use querySelector:

document.querySelector('div'): Tag Selector

document.querySelectorAll('.myClass'): Class Selector

document.querySelector('#myElement'): ID Selector

document.querySelectorAll('[data-attribute="value"]'): Attribute Selector

parentElement.querySelectorAll('.childClass'): Descendant Selector

parentElement.querySelectorAll('> .childClass'): Child Selector

document.querySelectorAll('input[type="text"]:valid'): Pseudo-Classes

document.querySelectorAll('ul > li'): Combinators

3.2 DOM Node types

Element Node: Represents HTML elements, such as <div>, <p>, and <a>.

Attribute node: attributes of HTML elements

Text Node: Contains text content within an element, including spaces and line breaks.

Comment Node: Represents comments within HTML, created with <!-- comment -->.

Document Node: Represents the entire HTML document and is the root of the DOM tree.

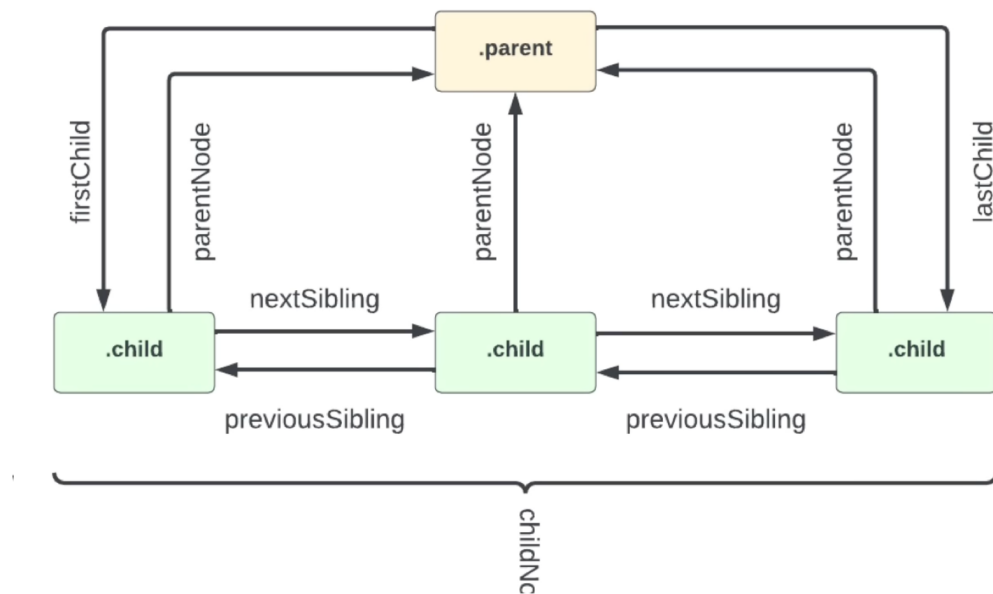
Document Type Node: Represents the <!DOCTYPE> declaration of the document.

23.11.2023 15.21.00

Document Fragment Node: Represents a lightweight container for holding multiple nodes, often used for efficiency.

3.2.1 DOM Node relationships

DOM Node Relationships



Accessing parent nodes:

```
const childNode = document.getElementById('child');
const parentNode = childNode.parentNode;
```

Accessing child nodes:

```
const parentNode = document.getElementById('parent');
```

```
// Using childNodes (includes all types of nodes)
const childNodes = parentNode.childNodes;
```

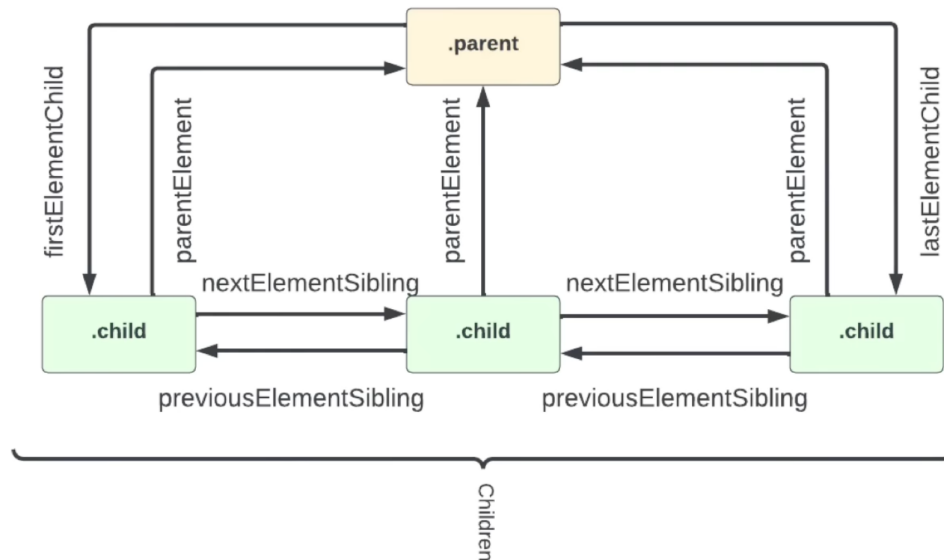
Accessing sibling nodes:

```
const siblingNode = node.nextSibling;
const previousSiblingNode = node.previousSibling;
```

3.3 DOM Elements

3.3.1 Element relationships

DOM Element Relationships



Accessing the parent element:

```
const childElement = document.getElementById('child');
const parentElement = childElement.parentNode;
```

Accessing child elements:

```
const parentElement = document.getElementById('parent');
```

```
// Using children (returns only element nodes)
const childElements = parentElement.children;
```

```
// Using childNodes (includes text and comment nodes)
const allChildNodes = parentElement.childNodes;
```

Accessing siblings:

```
const siblingElement = element.nextSibling;
const previousSiblingElement = element.previousSibling;
```

3.3.2 Create and append (to parent elements) elements



23.11.2023 15.21.00

Create element with tag name:

```
const newDiv = document.createElement('div');
```

Create text node to be placed in an element:

```
const newText = document.createTextNode('Hello, world !');
```

Append child to a parent element:

```
const parentElement = document.getElementById('parent');  
parentElement.appendChild(newDiv);
```

Insert node before a reference node within a parent element:

```
const parentElement = document.getElementById('parent');  
parentElement.insertBefore(newDiv, referenceNode);
```

Set or get the inner content of HTML element: `element.innerHTML`

Set or get the content of HTML element, including the element itself: `element.outerHTML`

Set or get the text content of HTML element: `element.textContent`

3.3.3 Inserting elements using `insertAdjacentElement()`

Allows you to insert HTML content at a specified position relative to the element calling the method:

Example:

```
<div id="myDiv">This is a div.</div>
```

```
<script>
```

```
// Select the element where you want to insert content  
const targetElement = document.getElementById('myDiv');
```

```
// Create a new paragraph element  
const newParagraph = document.createElement('p');  
newParagraph.textContent = 'This is a new paragraph.';
```

```
// Use insertAdjacentElement() to insert the new paragraph before the target element  
targetElement.insertAdjacentElement('beforebegin', newParagraph);
```

```
</script>
```

Parameters:

The `position` parameter can take values like "beforebegin", "afterbegin", "beforeend", and "afterend".

23.11.2023 15.21.00

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

3.3.4 Replacing elements

`replaceWith()` = replace an element in the DOM (Document Object Model) with another element or content

Syntax:

```
oldElement.replaceWith(newContent);
```

oldElement: The element you want to replace. This is the element you currently have in the DOM.

newContent: The content you want to replace the old element with. This can be an HTML element, text, or a combination of both.

Example:

```
<div id="container">
  <p>Original content</p>
</div>

<script>
  const container = document.getElementById('container');
  const newElement = document.createElement('div');
  newElement.textContent = 'New content';

  const originalElement = container.querySelector('p');
  originalElement.replaceWith(newElement);
</script>
```

Result:

```
<div id="container">
```



23.11.2023 15.21.00

```
<div>New content</div>
</div>
```

3.3.5 Remove elements

Remove an element:

```
const elementToRemove = document.getElementById('toBeRemoved');
elementToRemove.remove();
```

Remove a child element:

```
const parentElement = document.getElementById('parent');
const childElement = document.getElementById('child');
parentElement.removeChild(childElement);
```

3.3.6 Element text properties

textContent: The `textContent` property is available on most HTML elements. It represents the text content within the element, including any child elements.

innerText: It represents the text content within the element, excluding any child elements that are hidden with CSS or have no text content. It's similar to `textContent` but takes into account the CSS styling that affects visibility. You can use it on elements such as `<div>`, `<p>`, ``, etc.

innerHTML: The `innerHTML` property is available on most HTML elements as well. It allows you to access or modify the HTML content within an element, including any child elements and their attributes. You can use it on elements like `<div>`, `<p>`, ``, and others to manipulate the HTML content. Includes the HTML element tags.

3.3.7 Working with element classes

`classList` property returns a `DOMTokenList` object.

.add. Adds one or more class names to the element. If a class is already present, it won't be duplicate:

```
const element = document.getElementById('myElement');
element.classList.add('active', 'highlight');
```

.remove:

```
const element = document.getElementById('myElement');
element.classList.remove('active', 'highlight');
```

.toggle:

```
const element = document.getElementById('myElement');
element.classList.toggle('active'); // Toggles the 'active' class.
element.classList.toggle('highlight', true); // Adds the 'highlight' class.
```



23.11.2023 15.21.00

```
element.classList.toggle('active', false); // Removes the 'active' class.
```

.contains:

```
const element = document.getElementById('myElement');  
const hasClass = element.classList.contains('active');
```

.item:

```
const element = document.getElementById('myElement');  
const firstClass = element.classList.item(0); // Get the first class name.
```

.length:

```
const element = document.getElementById('myElement');  
const classCount = element.classList.length;
```

3.3.8 Working with CSS styling

3.4 Events

3.4.1 Event object

See earlier chapter [Event object](#)

3.4.2 Event listeners

```
// addEventListener()  
clearBtn.addEventListener('click', () => alert('Clear Items'));  
  
// Use named function  
clearBtn.addEventListener('click', onClear);  
  
// removeEventListener()  
setTimeout(() => clearBtn.removeEventListener('click', onClear), 5000);  
  
// Fire off event from JS  
setTimeout(() => clearBtn.click(), 5000);
```

3.4.3 Mouse events

`addEventListener("click")` = Listens for a single mouse click on an element.

`addEventListener("dblclick")` = Listens for a double-click (two rapid clicks) on an element.

`addEventListener("contextmenu")` = Listens for a right-click or context menu trigger on an element.



23.11.2023 15.21.00

`addEventListener("mousedown")` = Listens for the mouse button being pressed down on an element.

`addEventListener("mouseup")` = Listens for the mouse button being released after being pressed.

`addEventListener("wheel")` = Listens for mouse wheel scrolling (e.g., scrolling up or down) on an element.

`addEventListener("mouseover")` = Listens for the mouse pointer entering the area of an element.

`addEventListener("mouseout")` = Listens for the mouse pointer leaving the area of an element.

`addEventListener("dragstart")` = Listens for the start of a drag operation on an element (e.g., dragging a draggable item).

`addEventListener("drag")` = Listens for the drag operation itself while an element is being dragged.

`addEventListener("dragend")` = Listens for the end of a drag operation on an element.

3.4.3.1 Event object properties for mouse events

`target` - The element that triggered the event.

`currentTarget` - The element that the event listener is attached to (These are the same in this case).

`type` - The type of event that was triggered.

`timestamp` - The time that the event was triggered.

`clientX` - The x position of the mouse click relative to the window.

`clientY` - The y position of the mouse click relative to the window.

`offsetX` - The x position of the mouse click relative to the element.

`offsetY` - The y position of the mouse click relative to the element.

`pageX` - The x position of the mouse click relative to the page.

`pageY` - The y position of the mouse click relative to the page.

`screenX` - The x position of the mouse click relative to the screen.

`screenY` - The y position of the mouse click relative to the screen.

3.4.4 Key events

keydown: Triggered when a key is pressed.

keyup: Triggered when a key is released.

keypress: Triggered when a character key is pressed.

input: Triggered when input value changes.



23.11.2023 15.21.00

focus: Triggered when an element gains focus.

blur: Triggered when an element loses focus.

change: Triggered when the value of a form element changes.

3.4.4.1 Event object properties for key events

`event.key`: Represents the value of the key pressed (e.g., "a", "Enter", "Shift").

`event.keyCode`: Represents the numeric key code of the pressed key.

`event.ctrlKey`: Indicates whether the Ctrl key was pressed during the event.

`event.altKey`: Indicates whether the Alt key was pressed during the event.

`event.shiftKey`: Indicates whether the Shift key was pressed during the event.

`event.metaKey`: Indicates whether the Meta key (Command key on Mac) was pressed during the event.

3.4.5 Input events

input: Listens for changes in the value of an input element and records the value. Useful for getting value of form fields.

change: Listens for changes in the value of a form element. NOTE: with checkboxes and radios does not return a value when box is unchecked. When checked outputs true, when unchecked returns nothing.

focus: Listens for when an element gains focus.

blur: Listens for when an element loses focus.

select: Listens for text selection within an input element.

submit: Listens for the submission of a form element.

reset: Listens for the reset action on a form element.

3.4.5.1 Event object properties for input events

In the event listener callback function, arguments return a object. Useful with forms.

Get value with:

`event.target.value`

Get checkbox status with:

`event.target.checked`



23.11.2023 15.21.00

3.4.6 Form submissions and form object

Validate data on both front-end and back-end!

To prevent form submitting on front-end JS:

`event.preventDefault()`

3.4.6.1 FormData object

Object in JavaScript that allows you to easily work with HTML forms and their data. It provides methods to create, manipulate, and send form data in an HTTP request.

`let formData = new FormData(document.getElementById('myForm'));`

3.4.6.2 Form HTML

`<input>` Element Attributes:

- `type`: This attribute specifies the type of input control to be displayed. Common values include:
 - `text`: Text input.
 - `password`: Password input (characters are hidden).
 - `checkbox`: Checkbox for binary choices.
 - `radio`: Radio button for exclusive choices.
 - `file`: File upload control.
 - `submit`: Submit button for form submission.
 - `reset`: Reset button to clear form fields.
 - `number`: Input for numeric values.
 - `email`: Input for email addresses.
 - `date`: Input for date values.

Many more, depending on the desired input type.

- `id`: A unique identifier for the input element, used for associating labels with the input element using the `for` attribute in the `<label>`.
- `name`: The name of the input element, which is used when submitting the form to identify the form data.
- `value`: The default or current value of the input element.
- `placeholder`: A short hint that describes the expected value of the input. It's displayed in the input field when it's empty.
- `required`: When present, it indicates that the input field must be filled out before submitting the form.
- `disabled`: When present, it indicates that the input element is disabled and cannot be interacted with.



23.11.2023 15.21.00

3.4.7 Event bubbling

Event bubbling is a concept in JavaScript and the DOM (Document Object Model) where events triggered on an HTML element will also trigger events on its parent elements, propagating up the DOM tree. This propagation happens in the order in which the elements are nested, from the innermost child element to the outermost parent element.

You can do this using the `event.stopPropagation()` method within your event handler:

```
document.getElementById("child").addEventListener("click", function(event) {  
  console.log("Child element clicked");  
  event.stopPropagation(); // Stops the event from bubbling up  
});
```

3.4.8 Event delegation and multiple events

Event delegation is a technique in JavaScript where you attach a single event listener to a common ancestor of multiple elements instead of attaching individual event listeners to each element.

```
const listItems = document.querySelectorAll('li');  
const list = document.querySelector('ul');
```

```
//Add an event listener on all items  
listItems.forEach((item) => {  
  item.addEventListener('click', (e) => {  
    e.target.remove();  
  });  
});
```

3.4.9 Page loading and window events

Page loading and window events in JavaScript allow you to respond to various events related to the loading and interaction with a web page. Use `defer` if script in the head.

```
document.addEventListener("DOMContentLoaded", function() {  
  // Your code here  
});
```

```
window.addEventListener("load", function() {  
  // Your code here  
});
```

```
window.addEventListener("resize", function() {  
  // Your code here  
});
```

```
window.addEventListener("scroll", function() {
```



23.11.2023 15.21.00

```
// Your code here  
});
```

```
window.addEventListener("unload", function(event) {  
  // Your code here  
  event.preventDefault(); // This may prompt the user to confirm leaving  
});
```

3.5 Storing information from the user

Cookies vs. Local Storage vs. Session Storage

	Cookies	Local Storage	Session Storage
Capacity	4kb	10mb	5mb
Browsers	HTML4 / HTML 5	HTML 5	HTML 5
Accessible from	Any window	Any window	Same tab
Expires	Manually set	Never	On tab close
Storage Location	Browser and server	Browser only	Browser only
Sent with requests	Yes	No	No

3.5.1 Local storage

Client-side web storage feature that allows you to store key-value pairs in a user's web browser. It provides a way to store data persistently, even when the user closes their browser or navigates away from the page.

Storage limit 5-10MB.

Data stored as strings.

Storage is specific to the domain, from which it's accessed.

3.5.1.1 Common methods

[localStorage.setItem\(key, value\)](#)

Stores a key-value pair in local storage. If the key exists, it updates the value.

[localStorage.getItem\(key\)](#)

Retrieves the value associated with a specific key from local storage.

[localStorage.removeItem\(key\)](#)

Removes the key-value pair associated with a specified key from local storage.



23.11.2023 15.21.00

`localStorage.clear()`

Clears all key-value pairs from local storage for the current domain.

`localStorage.length`

Returns the number of key-value pairs currently stored in local storage.

`localStorage.key(index)`

retrieve the name of the key at a specific index.

// Getting the key at a specific index

const index = 0; // Replace this with the desired index

const key = localStorage.key(index);

```
if (key !== null) {  
    console.log(`Key at index ${index} is: ${key}`);  
} else {  
    console.log(`No key found at index ${index}`);  
}
```

3.5.2 Session storage

3.5.3 Cookies

3.6 AJAX and Fetch API

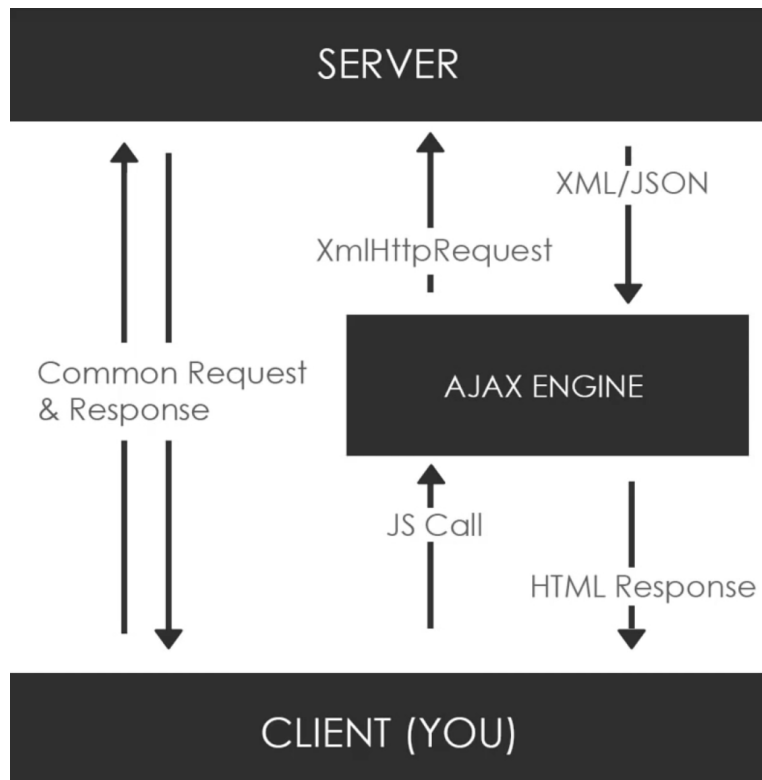
Set of web development techniques and technologies used to create asynchronous web applications. Enables sending and receiving data from web server without requiring page reload.

Components of AJAX:

1. **Asynchronous Communication:** AJAX allows web applications to make requests to the server in the background without interfering with the user's interaction on the page. This asynchronous behavior is crucial for creating responsive and interactive web applications.
2. **XMLHttpRequest:** In the early days of AJAX, the XMLHttpRequest object was used to send and receive data from the server. It's a JavaScript API that provides the foundation for making asynchronous HTTP requests.
3. **Data Formats:** While the "X" in AJAX originally stood for XML, it's important to note that modern AJAX applications often use various data formats, such as JSON (JavaScript Object Notation), HTML, or plain text, for data exchange. XML is no longer a strict requirement.
4. **DOM Manipulation:** AJAX is often used in combination with the Document Object Model (DOM) to update specific elements on a web page without requiring a full page reload. This enables developers to create dynamic and interactive user interfaces.
5. **Event Handling:** Event handling is a fundamental part of AJAX. You can define functions to be executed when a request is completed, enabling you to respond to the server's response data effectively.

23.11.2023 15.21.00

6. **Cross-Origin Requests:** AJAX allows web applications to make requests to different domains (cross-origin requests). However, this is subject to security restrictions, and you may need to implement mechanisms like Cross-Origin Resource Sharing (CORS) to handle such requests securely.
7. **Frameworks and Libraries:** Many JavaScript frameworks and libraries, such as jQuery, React, and Angular, offer AJAX capabilities or have built-in features for making asynchronous requests, simplifying the implementation of AJAX in web applications.



3.6.1 XML

eXtensible Markup Language, is a widely used markup language designed to store and transport data in a structured format. It's often used in various applications, including web development, data interchange, configuration files, and more.

- **Markup Language:** XML is a markup language, similar to HTML, but it's not focused on describing how data should be presented like HTML. Instead, XML focuses on describing the structure of data. It uses tags to enclose data elements, which are usually user-defined, and attributes to provide additional information about those elements.
- **Self-Descriptive:** XML documents are self-descriptive, meaning that they contain both the data and the metadata needed to understand that data. Each XML document typically starts with a declaration specifying the version of XML being used.
- **Hierarchical Structure:** XML data is organized in a hierarchical or tree-like structure. Elements can have child elements, creating a parent-child relationship. This hierarchical structure is suitable for representing a wide range of data formats.
- **Tags and Elements:** XML documents consist of tags that define elements. Elements can contain text, other elements, or a combination of both. For example:



23.11.2023 15.21.00

```
<book>
  <title>XML Basics</title>
  <author>John Doe</author>
  <price>19.99</price>
</book>
```

Attributes: Elements can have attributes, which provide additional information about the element. Attributes are written as key-value pairs within the element's opening tag. Example:

```
<book id="123" category="programming">...</book>
```

3.6.2 JSON

See chapter: [JSON](#)

3.6.3 DOM Parser

A DOM Parser, short for Document Object Model Parser, is a software tool or API used in web development and programming to parse and manipulate XML or HTML documents. It allows developers to access and interact with the content and structure of web documents as a tree-like structure, where each element in the document is represented as an object, making it easier to work with and manipulate.

3.6.3.1 Common DOM Parser methods and properties

DOMParser Constructor: This is used to create a new `DOMParser` instance. Example:

```
var parser = new DOMParser();
```

parseFromString(text, type): This method is used to parse a string of markup (e.g., HTML or XML) and returns a `Document` object. Example:

```
var doc = parser.parseFromString("<h1>New Heading</h1>", "text/html");
```

3.6.4 Parsing XML data

Use the `DOMParser` API. Here's how to do it:

```
var parser = new DOMParser();
var xmlDoc = parser.parseFromString(xmlText, "text/xml");
```

`xmlText` is the XML content retrieved from the `XMLHttpRequest`. `xmlDoc` is now a `Document` object representing the parsed XML data.

3.6.5 XMLHttpRequest (XHR)

Example:

```
const XHR = new XMLHttpRequest()

XHR.open("GET", "http://iceberg-cycle.codio.io/5: Asynchronous JavaScript (AJAX)/famous-quotes.xml", true)
```



23.11.2023 15.21.00

```
XHR.send()

XHR.onreadystatechange = () => {
  if (XHR.readyState === 4 && XHR.status == 200) {
    // do something here
  }
}
```

Object that provides an easy way to make HTTP requests to a server from a web page without having to refresh the page. It allows you to retrieve data from a URL and update parts of a web page without a full page reload.

- API in the form of an object
- Provided by the browser's JS environment
- Methods transfer data between client/server
- Can be used with other protocols than HTTP
- Can work with data other than XML (JSON, plain text)

3.6.5.1 Common XMLHttpRequest properties

onreadystatechange: This property allows you to define a function to be called when the `readyState` property changes, indicating the progress of the request.

readyState: It represents the current state of the XMLHttpRequest. It can have values from 0 to 4, indicating different states such as uninitialized (0), open (1), sent (2), receiving (3), and loaded (4).

responseText: This property contains the response data as a text string when the request is complete. It's commonly used for text-based responses.

responseXML: When the response is in XML format, this property holds the response data as an XML Document object.

response: This property returns the response data as an ArrayBuffer, Blob, Document, JavaScript object, or a DOMString, depending on the value of the `responseType` property.

responseType: You can set this property to specify the type of data you expect in the response. Common values include "text," "json," "document," "blob," and "arraybuffer."

timeout: This property sets a timeout in milliseconds for the request. If the request takes longer than the specified time, it will be aborted, and the `onload` event handler with a `status` of 0 will be triggered.

withCredentials: When set to `true`, this property allows cookies and authentication headers to be sent with the request in a cross-origin request. It's often used in conjunction with CORS.

status: This property contains the HTTP status code returned by the server, like 200 for a successful request, 404 for not found, etc.

statusText: It provides a textual description of the HTTP status code, such as "OK" for 200, "Not Found" for 404, etc.

3.6.5.2 Common XMLHttpRequest methods

open(method, url, async): This method initializes the request. You specify the HTTP method (e.g., "GET" or "POST"), the URL to send the request to, and whether the request should be asynchronous (`true`) or synchronous (`false`).



23.11.2023 15.21.00

send(data): It sends the request to the server. If the request is asynchronous, this method returns immediately, and the request continues in the background. You can optionally include data as a parameter for POST requests.

setRequestHeader(header, value): This method sets a request header with the given name and value. You can use it to include custom headers in your HTTP request.

abort(): This method cancels the current request if it's in progress. It halts the request and calls the `onreadystatechange` event handler with a `readyState` of 4 (loaded) and `status` set to 0.

getAllResponseHeaders(): It returns all the response headers as a single string.

getResponseHeader(header): This method returns the value of a specific response header, given its name.

- **addEventListener(event, callback)**: In addition to the traditional `onreadystatechange` event, you can use this method to add event listeners for various events such as "load," "error," "abort," and others.
- **getAllResponseHeaders()**: This method returns all the response headers as a single string. You can parse this string to access individual headers.
- **getResponseHeader(header)**: Like the previous response method, this one returns the value of a specific response header, given its name.
- **sendAsBinary(string)**: This method is used to send binary data as a string. It's not commonly used, and binary data is often sent as `ArrayBuffer` or `Blob` using the `send()` method.
- **setResponseType(type)**: This method allows you to set the `responseType` property to a new value, changing the expected response type dynamically.
- **abort()**: As mentioned earlier, this method cancels the current request if it's in progress.

3.6.6 Fetch API

For information about Promises, see [Promises](#)

The primary function for making requests is the `fetch()` function, which returns a Promise that resolves to the `Response` to that request, whether it is successful or not.

Basic usage:

```
fetch('https://api.example.com/data')
  .then(response => {
    // Handle the response
  })
  .catch(error => {
    // Handle errors
  });
```

Request and Response objects:

The `fetch()` function takes a mandatory parameter, which is the URL to the resource you want to fetch. Additionally, you can pass an optional `init` object to configure the request, such as method, headers, body, etc.

```
fetch('https://api.example.com/data', {
```



23.11.2023 15.21.00

```
method: 'GET', // HTTP method
headers: {
  'Content-Type': 'application/json'
  // Other headers as needed
},
// Other options like body, mode, credentials, etc.
})
```

Handling responses:

The `fetch()` function returns a Promise that resolves to a `Response` object representing the response to the request. You can then use various methods on the `Response` object to extract information.

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json(); // Parse the response as JSON
  })
  .then(data => {
    // Handle the parsed data
  })
  .catch(error => {
    // Handle errors
  });
```

Request methods:

The `fetch()` function supports various HTTP methods such as GET, POST, PUT, DELETE, etc. The method can be specified in the `init` object.

```
fetch('https://api.example.com/data', {
  method: 'POST',
  body: JSON.stringify({ key: 'value' }),
  headers: {
    'Content-Type': 'application/json'
  }
})
```

Async and Await:

You can also use the `async/await` syntax to make the code more readable and synchronous-looking when dealing with asynchronous operations.

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    const data = await response.json();
    // Handle the parsed data
  }
}
```



23.11.2023 15.21.00

```
} catch (error) {  
  // Handle errors  
}  
}
```

3.6.6.1 Response object

Represents the response to a request made by the `fetch` function or other similar APIs.

3.6.7 HTTP response status codes

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

1. Informational responses: 100 - 199
2. Successful responses 200 - 299
3. Redirection messages 300 - 399
4. Client error responses 400 - 499
5. Server error responses 500 - 599

4 jQuery

4.1 Basics of jQuery

jQuery uses the `$` (dollar sign) as a shorthand for the `jQuery` function. It's the entry point for using jQuery in your code.

Chaining: jQuery allows you to chain multiple actions together. This is achieved by attaching additional actions to the end of the selection, separated by dots.

Example:

```
// Example: Chaining actions to change text and add a CSS class  
$("p").text("New text content").addClass("highlight");
```

Document ready: To ensure that your jQuery code runs after the DOM has fully loaded, you often wrap your code inside the `$(document).ready()` function. A shorthand for this is `$(function() { /* your code here */ });`.

Example:

```
// Example: Document ready function  
$(document).ready(function() {  
  // Your code here  
});
```

Shorthand example:

```
// Example: Shorthand for document ready
```



23.11.2023 15.21.00

```
$(function() {  
    // Your code here  
});
```

4.2 Common methods

`var paragraphs = $("p");` // Selects all <p> elements

Change text content:

```
// Changing text content  
$("#p").text("New text content");
```

Classes:

```
// Adding or removing CSS classes  
$("#p").addClass("highlight");  
$("#p").removeClass("highlight");
```

```
// Checking if an element has a class  
$("#myElement").hasClass("highlight");
```

```
// Filtering by class  
$("div").filter(".highlight");
```

Events:

```
// Click event  
$("#button").on("click", function() {  
    // Your event handling code  
});
```

Ajax:

```
// AJAX GET request  
$.ajax({  
    url: "https://api.example.com/data",  
    method: "GET",  
    success: function(response) {  
        // Handle successful response  
    },  
    error: function(error) {  
        // Handle error  
    }  
});
```

Animation:

```
// Fading out an element  
$("#myElement").fadeOut(1000); // 1000 milliseconds (1 second) duration
```

Find child elements



23.11.2023 15.21.00

```
// Finding child elements  
$("#parentElement").find("p");
```

For each- type functionality

```
// Iterating over selected elements  
$("li").each(function(index, element) {  
    // Your iteration code  
});
```

4.3 Common properties

length:

Property that indicates the number of elements in the jQuery object.

```
var count = $("p").length;  
context:
```

Property that refers to the document object associated with the jQuery object.

```
var doc = $("p").context;  
selector:
```

Property that contains the selector used to create the jQuery object.

```
var selector = $("p").selector;
```