JAVASCRIPT

INTERNET ENGINEERING

Fall 2022

@1995parham

- Introduction
- JavaScript Basic
- JavaScript & DOM & CSS
- Event Handling
- Web Applications
- Summary

INTRODUCTION (WHY JAVASCRIPT?)

- HTML + CSS create *Static* webpages
- We need
  - interact with user (event handling)
  - check input data (input validation)
  - ...
  - Programming in web pages
- Solution: *Client Side* scripting (why not server-side?)
  - JavaScript
  - TypeScript, Dart, CoffeeScript, ...
    - (mostly) transcompiled into JavaScript

WHAT IS JAVASCRIPT?

- JavaScript is an *prototype-based*, *client-side*, and *scripting* languages to make web pages dynamic

- Prototype-based: NOT class-based
    - Uses generalized objects, which can then be cloned and extended.
    - Using fruit as an example, a "fruit" object would represent the properties and functionality of fruit in general. A "banana" object would be cloned from the "fruit" object and general properties specific to bananas would be appended.
    - Each individual "banana" object would be cloned from the generic "banana" object.

- Client-side: Run by client side program (the web browser)
- Scripting: Doesn't require to be compiled before run. All interpretation is done on-the-fly by the client

- Additional facts about JS:
    - JavaScript is not related to Java.
    - Its standard name is ECMAScript (Stable version: ES 2020)
    - Sever side and other applications (e.g., GNOME Shell) are not discussed here.

WHAT IS JAVASCRIPT IN WEB?

- JavaScript is a part of HTML document
    - Web browser reads, interprets, and runs it
- JavaScript can
    - put dynamic text into an HTML page
    - react to events
    - read and write HTML elements
    - validate input data
    - access to cookies
    - access to browser data (history, ...)
    - read HTTP headers
    - ...

- JavaScript scope is limited to the browser window is written in
    - Scripts run in a *sandbox* in which they can only perform Web-related actions
    - We can't access files or other system resources using JavaScript

HOW TO ADD JAVASCRIPT TO HTML?

- Embed JavaScript into HTML

```
<script type="text/javascript">
// JavaScript code goes here...
</script>
```

- Link to external JavaScript

```
<script type="text/javascript" src="external.js" defer></s
```

- JavaScript code can be put in both head and body

- Browser reads the HTML file, when it reaches to JavaScript code, it runs the code, except functions that should be called and the deferred scripts
- defer specifies that the script is executed when the page has finished parsing.
- You can use integrity attribute to verify resource. (Specially when we fetch them from a CDN)

JAVASCRIPT PROGRAMMING LANGUAGE

- Focus on JavaScript as a general programming language, not a web scripting language
- What do we need to learn?
    - Language basic syntax
    - Language type: functional, OO, ...
    - Operations
    - Execution flow control
    - Input/output
    - Libraries
    - ???

HELLO WORLD

```
console.log('Hello World')
```

JAVASCRIPT SYNTAX

- Syntax is very similar to C/Java/C++ (But with some differences)
- Semicolon is required between multiple statements in a line
    - If a single statement in line, in most cases, semicolon is optional
- There is not any `main` function
- Functions are defined by `function` (and also arrows)

- Variables are declared by `var` and `let` (is optional)
    - Required for local variables (inside functions)

```
i = j = 0;
function f(){
  let i;
  i = j = 10;
}
f();
//i = 0, j = 10
```

- It is Not Recommended to declare a variable without keyword.
- It can accidently overwrite an existing global variable.
- Scope of the variables declared without keyword become global irrespective of where it is declared.
- Global variables can be accessed from anywhere in the web page.

- `var` has a function scope
- `let` has a block scope

JAVASCRIPT BASIC: VARIABLES

- No type is given in variable declaration
  - Similar to other scripting (interpreted) languages
  - Type is identified by value
  - Type of variable can *be changed*

- Two main types
  - Primitive: Number, String, Boolean, BigInt, Symbol, Undefined, and Null
    - They are *immutable*

```
let x = "abc";
x[0]='Z';
// does NOT work!!!
```

- Objects: e.g., `w = window;`
    - Wrapper objects for primitives, String for string
        - When we treat a primitive value like it was an object (i.e. by accessing properties and methods), JavaScript creates, under the hood, a wrapper to wrap this value and expose it as an object.
    - Arrays are some special objects
    - Objects are *mutable*

```
function mutateArray(A) {
  A[0] = 0;
}
let A = [1];
console.log(A);  // [1]
mutateArray(A);
console.log(A);  // [0]
```

```
function mutateNumber(n) {
  n = 0;
}
let n = 1;
console.log(n);  // 1
mutateNumber(n);
console.log(n);  // 1
```

both arrays and numbers are passed by sharing. Whereas arrays are mutable, numbers are not.

```
function mutateArray(A) {
  A = [0];
}
A = [1];
console.log(A);  // [1]
mutateArray(A);
console.log(A);  // [1]
```

Here we are no longer mutating the array; we are now binding the name A to a new array.

Objects can have properties and methods while primitive values can't

```
const name = 'Darth Vader';
name.alignment = 'Lawful evil';
name.tellTheTruth = () => {
  console.log('Luke, I am your father!');
};

console.log(name.alignment); // undefined
name.tellTheTruth(); // Uncaught TypeError: name.tellTheTruth is not a function
```

- A wrapper object is disposed right after a single use.
- When you interact with a primitive value like it was an object (by calling a method or reading a property from it), JavaScript creates a wrapper object on the fly.
- Due to this auto-disposal mechanism, properties and methods injected into wrapper objects are immediately lost.

```
const name = new String('Darth Vader');
name.alignment = 'Lawful evil';
name.tellTheTruth = () => {
  console.log('Luke, I am your father!');
};

console.log(name.alignment);
name.tellTheTruth();
```

JAVASCRIPT BASIC: VARIABLES SCOPE

- By default (either `var` or not) scope is either *global* or *function*
  - Variables defined in a block (other than functions) are accessible outside the block

```
function f() {
  while(true) {
     var x = 20;
  }
  // x = 20 here
}
```

- From ECMA 2015, `let` and `const` can be used to define *block* scope

```
function f() {
  while(true) {
    let x = 20;
  }
  // x is not accessible here
}
```

JAVASCRIPT BASIC: OPERATORS

- Arithmetic: `+ - * / % **`
  - String concatenation: `+`
  - Power: `**`
- Assignment: `= += -= *= /= %= ++ --`
- Comparison: `== === != !== > >= < <=`
  - `2 == "2"` returns true, `2 === "2"` returns false
  - `2 != "2"` returns false, `2 !== "2"` returns true
- Logical: `&& || !`
- Comments: `// /* */`

JAVASCRIPT BASIC: CONDITIONAL STATEMENTS & LOOPS

- Conditional statements (the same as C)
  - `if-else`
  - `switch-case`
  - Ternary operator `? :`
- Loops (the same as C)
  - `while`
  - `for`
  - `do-while`
  - `break` and `continue`

JAVASCRIPT BASIC: FUNCTIONS

- Function definition

```
function name(input1, input2, ...){
  ...
  return result;
}
```

- No output type, no input arguments type
- Function call: `retVal = name(input1, input2, ...);`
  - Input argument are called by value for primitive types
  - Call by reference for objects

```
function changeMe(value) {
  value = 10;
}

function changePropertyInMe(value) {
  value.x = 20;
}

point = { x: 10, y: 10, toString: () => console.log(`(${this.x}, ${this.y})`) };
console.log(point);

changeMe(point);
console.log(point);

changePropertyInMe(point);
console.log(point);
```

- Function assignment: `object.onclick = func;`
- Function in Function definition is allowed
- Function declarations are not part of the regular top-to-bottom flow of control (JavaScript Hoisting)

```
square(5);

function square(y) {
  return y * y;
}
```

JAVASCRIPT BASIC: FUNCTIONS (CONTD.)

- Function as a value

```
let f = function (input1, input2, ...
    ...
    return result;
}
```

- Arrow Functions

```
let f = (input1, input2, ...) =
  ...
  return result;
}
```

```
let sq = x => x * x
```

```
let f = () => 0
```

JAVASCRIPT BASIC: INPUT & OUTPUT

- To prompt a dialog to user and get input

```
let input = prompt("Please enter your name");
```

Run

- `window.prompt` returns *string*
    - To convert string to integer:

    ```
    i = Number.parseInt("10");
    ```

    - To convert string to float:

    ```
    f = Number.parseFloat("20.2");
    ```

- To get confirmation from user

```
let question = confirm("Do you want to continue?");
```

Run

  - `window.confirm` returns a boolean
    - Ok: `true`
    - Cancel: `false`

JAVASCRIPT BASIC: INPUT & OUTPUT

- To show a message in an alert window

```
window.alert("We study 'Internet Engineering'"
```

Run

- Object `document` has a `write` method

```
document.write("A sample message");
```

Run

- Most objects have `innerHTML` attribute

```
<span id="testbox">This is my innerHTML</span>

<script>
document.getElementById("testbox").innerHTML = "This is
</script>
```

This is my innerHTML

ARRAYS

- Neither the length of a JavaScript array nor the types of its elements are fixed
- Data can be stored at non-contiguous locations in the array
- Setting or accessing via non-integers using bracket notation (or dot notation) will not set or retrieve an element from the array list itself, but will set or access a variable associated with that array's object property collection.

```
let a = [1, 2, 3];
let a = new Array (10, 20, ...);
let a = new Array();
a[10] = "HTML";
a[120] = "JS";
```

- Methods: `concat`, `shift`, `unshift`, `sort`, `reverse`, `indexOf`, ...

```
let friends = ["Saman", "Sepehr", "Hessam"];
friends.length // 3
friends.push("Ali") // 4
friends.concat("Ali") // ["Saman", "Sepehr", "Hessam", "Ali", "Ali"]
let last = friends.pop() // "Ali"
last // "Ali"
friends.map(i => i.toUpperCase()) // ["SAMAN", "SEPEHR", "HESSAM"]
friends.filter(i => i.startsWith("S")) // ["Saman", "Sepehr"]
```

```
let num = [1, 2, 3, 4, 5, 6, 7, 8, 9];

num.reduce((sum, i) => sum + i, 0) // 45
```

- for/of - looping over iterable objects

```
for (let friend of friends) {
  console.log(friend)
}
```

- Also there are typed arrays
    - The contents are initialized to 0.
    - You can reference elements in the array using the object's methods, or using standard array index syntax (that is, using bracket notation).
    - Fix-sized

```
// From a length
let uint8 = new Uint8Array(2);
uint8[0] = 42;
console.log(uint8[0]); // 42
console.log(uint8.length); // 2
console.log(uint8.BYTES_PER_ELEMENT); // 1

uint8[2] = 1;
console.log(uint8[2]); // undefined
console.log(uint8.length) // 2
```

SAMPLE LIBRARIES: MATH, NUMBER & DATE OBJECTS

- Math: `abs`, `sin`, `asin`, `ceil`, `floor`, `log`, `exp`, `pow`, `random`, ...
- Number:

```
let num = 1.1;
num.toExponential() // '1.1e0'
num.toFixed() // '1'
```

- Date:

```
let d = new Date();
d.toString(); // "Thu Nov 05 2020 09:50:38 GMT+0330 (Iran Standard
```

Fri Dec 23 2022 20:45:01 GMT+0330 (Iran Standard Time)`d.`set/get `FullYear`, `Month`, `Date`, `Hours`, `Minutes`, `Seconds`

OBJECT

- An object is a collection of variables (fields) and functions (methods)

```
let book = {
  name: "OOP in JS",
  price: 100,
  publish: 2020,
  getPrice: function(){
    return this.price;
  }
};

book.setPrice = function (x){this.price=x;};
book.setPrice(1000);
window.alert("name is "+ book.name +", price is "+ book.getPrice());
```

OBJECT (CONT.)

- Creating an (empty) object by the `Object` object and new

```
let book = new Object();
book.name = "OOP in JS";
book.price = 100;
book.publish = 2020;
book.getPrice = function() { return this.price; };
book.setPrice = function(x) { this.price=x; };
book.setPrice(1000);
window.alert("name is "+ book.name +", price is "+ book.getPrice());
```

OBJECT ACCESSORS

- To control to access to object's fields use getter and setter
  - The fields are not protected/private (like Java)

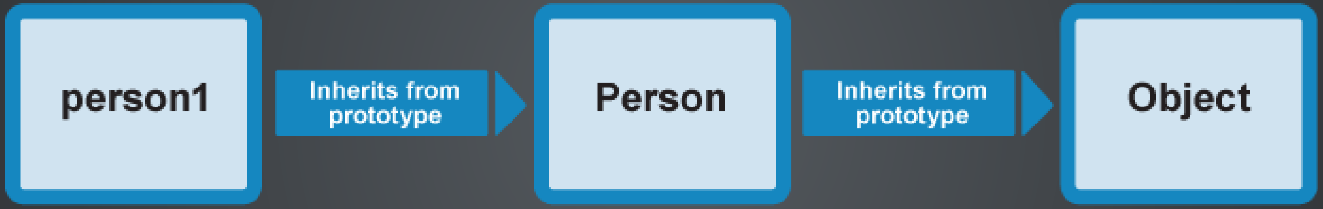
```
let book = {
  name: "OOP in JS",
  _price: 0,
  get price(){
    return this._price + "$";
  },
  set price(val){
    if (val < 0)
      window.alert("Invalid price");
    this._price = val;
  }
};

book.price = 1000;
window.alert("name is "+ book.name +", price is "+ book.price);
book.price = -100;
book._price = -100;
window.alert("name is "+ book.name +", price is "+ book.price);
```

PROTOTYPE

- `Object.create()`: creates a new object, using an existing object as the prototype of the newly created object.

```
// Person is our portotype
let Person = {
  _name: "",
  _family: "",

  get name() {
    return this._name;
  },

  get family() {
    return this._family;
  },

  set name(name) {
    this._name = name;
  },

  set family(family) {
```

- Prototypes are the mechanism by which JavaScript objects inherit features from one another.


person1
Inherits from prototype
Person
Inherits from prototype
Object

- In our code we define the constructor, then we create an instance object from the constructor, then we add a new method to the constructor's prototype:

```
function Person(first, last, age, gender, interests) {
  // property and method definitions
}

let person1 = new Person('Tammi', 'Smith', 32, 'neutral', ['music', 'skiing', 'kic

Person.prototype.farewell = function() {
  alert(this.name.first + ' has left the building. Bye for now!');
};
```

- But the farewell() method is still available on the person1 object instance — its members have been automatically updated to include the newly defined farewell() method.

OOP: INSTANTIATING MULTIPLE OBJECTS

- To be more OOP-like, we need
    - Differentiation between class and object
    - Instantiating via constructor
- Traditional/Base/Conceptual/Common approach:
    - A function is also a type of object
    - Functions are used to define objects as the constructor
    - Inheritance is implemented by `prototype` property of the function

- Easier/New (ECMA 2015) approach:
  - No new concept, syntactical sugar over JavaScript's existing prototype-based inheritance
  - Classes are defined by `class`, which is a special function
  - Common OOP terminologies: `extends`, `super`, ...
  - *Only* for methods (properties should be deinfed using the `prototype`)

OBJECTS

- Object instantiation: `new`
- property deletion: `delete`
- Access to properties/methods: `.`
- This object: `this`
    - Needed to access properties in methods

WHAT DOES `new Foo(...)` DO?

1. A new object is created, inheriting from `Foo.prototype`
2. The constructor function `Foo` is called with the specified arguments, and with `this` bound to the newly created object.

ECMA 2015

- Class definition: `class`
- Properties declaration (optional)
- Method declaration: function definition **without** `function`

- Private/Protected properties/methods: **Underdevelopment** (can also be emulated)
- Static method: `static`
- At Constructor:
  - This class: `constructor`
  - Parent: `super`

OBJECTS EXAMPLE

```
function Student(name, id){
  window.alert("I am going to create a new student");
  this.name = name;
  this.id = id;
  this.toString = function(){
    return `${this.name}: ${this.id}`;
  }
}

let st1 = new Student("Parham Alvani", "9231058");
console.log(st1);
```

```
class Student {
  constructor(name, id){
    window.alert("I am going to create a new student");
    this.name = name;
    this.id = id;
  }
  toString() {
    return `${this.name}: ${this.id}`;
  }
}

let st1 = new Student("Parham Alvani", "9231058");
console.log(st1);
```

OBJECTS EXAMPLE

```
class Bachelor extends Student {
  constructor(name, id){
    super(name, id);
    this.average = function() {
      return 20;
    }
  }

  isPass(){
    return 'Pass';
  }

  static betterThan(a, b){
    return a.average() >= b.average();
  }
}

bc1 = new Bachelor("Parham Alvani", "9231058");
```

USE STRICT

- The `"use strict"` directive was new in ECMAScript version 5.
- It is not a statement, but a literal expression, ignored by earlier versions of JavaScript.
- Strict mode makes it easier to write "secure" JavaScript.
- Strict mode changes previously accepted "bad syntax" into real errors.
- Strict mode is declared by adding `"use strict"`; to the beginning of a **script** or a **function**.

NOT ALLOWED IN STRICT MODE

- Using a variable, without declaring it, is not allowed.
- Deleting a variable (or object) is not allowed.
- Deleting a function is not allowed.
- Duplicating a *parameter name* is not allowed.
- Octal numeric literals are not allowed.
- Writing to a get-only property is not allowed.
- Keywords reserved for future JavaScript versions can *NOT* be used as variable names in strict mode.

```
"use strict";
myFunction();

function myFunction() {
  y = 3.14;   // This will also cause an error because y is not declared
}
```

CONCURRENCY MODEL AND THE EVENT LOOP

JavaScript has a concurrency model based on an **event loop**, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks.

STACK

Function calls form a stack of frames.

HEAP

Objects are allocated in a heap which is just a name to denote a large (mostly unstructured) region of memory.

QUEUE

A JavaScript runtime uses a message queue, which is a list of messages to be processed. Each message has an associated function which gets called in order to handle the message.

EVENT LOOP

- The event loop got its name because of how it's usually implemented, which usually resembles:

```
while (queue.waitForMessage()) {
  queue.processNextMessage()
}
```

- queue.waitForMessage() waits synchronously for a message to arrive (if one is not already available and waiting to be handled).

- Each message is processed completely before any other message is processed.
- This offers some nice properties when reasoning about your program, including the fact that whenever a function runs, it cannot be pre-empted and will run entirely before any other code runs (and can modify data the function manipulates).
- A downside of this model is that if a message takes too long to complete, the web application is unable to process user interactions like click or scroll.

ZERO DELAYS

- Zero delay doesn't actually mean the call back will fire-off after zero milliseconds.
- Calling setTimeout with a delay of 0 (zero) milliseconds doesn't execute the callback function after the given interval.

```
this is the start
this is just a message
this is the end
Callback 1: this is a msg from call back
Callback 2: this is a msg from call back
```

```
(function () {
  let el = document.getElementById("event-loop");

  el.innerHTML += "this is the start<br />";

  setTimeout(function cb() {
    el.innerHTML += "Callback 1: this is a msg from call back<br />";
  }); // has a default time value of 0

  el.innerHTML += "this is just a message <br />";

  setTimeout(function cb1() {
    el.innerHTML += "Callback 2: this is a msg from call back<br />";
  }, 0);

  el.innerHTML += "this is the end<br />";
})();
```

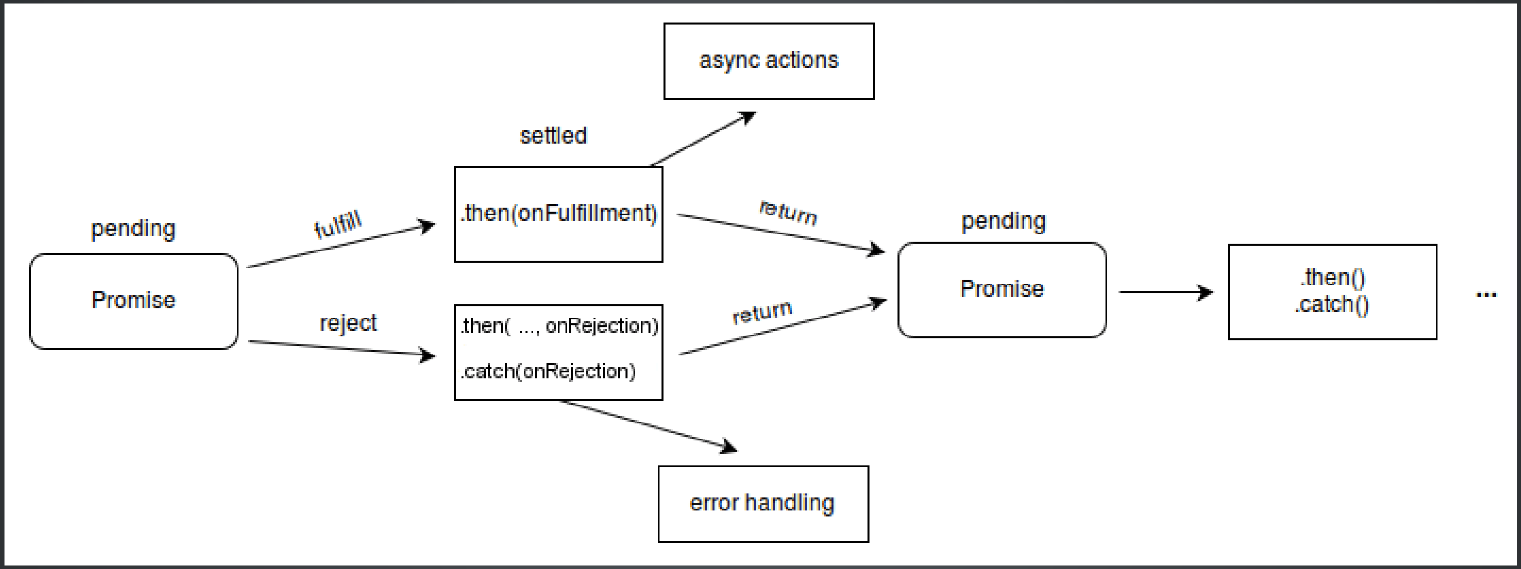
NEVER BLOCKING

- A very interesting property of the event loop model is that JavaScript, unlike a lot of other languages, never blocks.
- Handling I/O is typically performed via events and callbacks, so when the application is waiting for an IndexedDB query to return or an XHR request to return, it can still process other things like user input.

PROMISE

- The **Promise** object represents the eventual completion (or failure) of an *asynchronous* operation and its resulting value.
- A `Promise` is in one of these states:
  - *pending*: initial state, neither fulfilled nor rejected.
  - *fulfilled*: meaning that the operation was completed successfully.
  - *rejected*: meaning that the operation failed.

- A pending promise can either be fulfilled with a value or rejected with a reason (error).
- When either of these options happens, the associated handlers queued up by a promise's then method are called.
- The `.then()` method takes up to two arguments; the first argument is a callback function for the *resolved case* of the promise, and the second argument is a callback function for the *rejected case*.


async actions
settled
pending
fulfill
.then(onFulfillment)
return
pending
Promise
Promise
reject
.then(..., onRejection)
.catch(onRejection)
return
.then()
.catch()
...
error handling

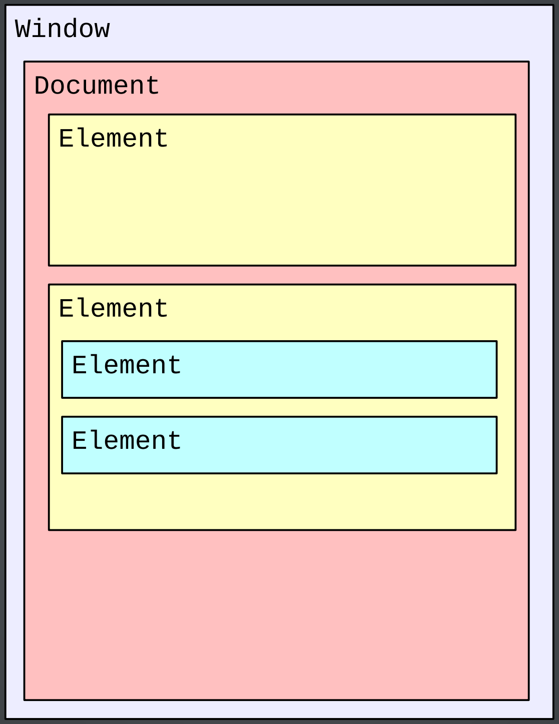
```
// timeout is 300 millisecond
const timeout = 300;

function longCalculation(id) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`resolved-${id}`);
    }, timeout);
  });
}

longCalculation(1)
  .then((result) => {
    console.log(`1st promise: ${result}`);
    return longCalculation(2);
  })
  .then((result) => {
    console.log(`2nd promise: ${result}`);
```

JAVASCRIPT IN WEB

- Basic functionality of JS in Web: *Dynamic pages*
    - Create and delete HTML elements
    - Modify HTML element contents
    - Modify element's styles
- Question: How to access HTML elements from JavaScript?
- Answer: *DOM*
    - Web browser parses HTML document
    - Output is a data-structure called "DOM"
    - Browser provides *API* to access the DOM

DOM

- Document Object Model (DOM) is the output of parsing HTML file by browser
    - HTML document is represented by DOM in browser
- Each HTML element is represented by an *object*
    - However, there are other objects (called BOM) that are not corresponding to HTML elements; e.g., `window` is the browser window
- A tree of object corresponding to HTML tag hierarchy
- Each object has
    - Properties corresponding to *HTML properties* (not all properties)
    - Methods corresponding to *events* and *actions*


Window
Document
Element
Element
Element
Element

DOM & JAVASCRIPT

- JavaScript is powerful in web since it accesses to DOM
  - JavaScript + DOM = Dynamic HTML
- *Read* access in order to
  - Check properties (conditional reaction)
  - Validate inputs
  - Handle events
- *Write* access in order to
  - Modify content
  - Modify styles
  - Add/Remove objects to/from DOM

DOM OBJECT ACCESS BY NAME

```
<form name="nametest">
<input name="output" type="text" value="Default Text">
</form>
```

```
document.nametest.output.value='New Value';
```

- We need to assign a name to all parents & children
- Is not really useful and practical

DOM OBJECT ACCESS

- `element.getElementById("id")`
  - returns the object with given "id" in *sub-tree* rooted at element
- `element.getElementsByClassName("classname")`
  - returns an array of object whose class is "classname" in *sub-tree* rooted at element
- `element.getElementsByTagName("tagname")`
  - returns an array of objects whose tag is "tagname" in *sub-tree* rooted at element
- `element.getElementsByName("name")`
  - returns an array of objects whose name is "name" in *sub-tree* rooted at element

- `element.querySelector("CSS Selector")`
    - returns the first element matches the CSS selector in *sub-tree* rooted at element
- Element *must be* exists (browser reads HTML file line-by-line)

ACCESS TO DOM OBJECTS EXAMPLE

```
<div id="box1">
```

```
<div class="testbox">
```

```
document.getElementById("box1").innerHTML="I am the new message"
```

Run

```
let da = document.getElementsByClassName("testbox");
da[0].innerHTML="I am another new message";
```

Run

SAMPLES OF DOM OBJECT'S PROPERTIES

- Element.innerHTML
  - HTML or XML markup contained within the element
  - Setting the value of innerHTML **removes** all of the element's descendants and replaces them with nodes constructed by parsing the HTML given in the value.
- Element.className
  - Is a String representing the class of the element.

```
document.getElementById("My_Element").className += " My_Class";
```

- ParentNode.children
  - Array of children elements
- ChildNode.parentNode
  - The parent of element

- HTMLElement.`style`
    - The styling rules of element
- HTMLElement.`value`
    - The value of input elements

CSS & JAVASCRIPT

- (Most) objects (document's children) have `style` property
- CSS style properties of each object are the properties of `style`
- CSS and DOM use different name (syntax) for the same style. E.g., Background color:
  - In CSS: `background-color`
  - In DOM: `backgroundColor`
- To access a CSS style property in JavaScript: `object.style.PropertyNameInDOM`

CSS & JAVASCRIPT EXAMPLE

```
let color;
index = prompt("Enter 1 for blue, 2 for red, 3 for green");
switch(parseInt(index)) {
  case 1:
    color = "blue"; break;
  case 2:
    color = "red"; break;
  case 3:
    color = "green"; break;
  default:
    color = "black"; break;
}

document.getElementById("colorBox").style.backgroundColor = color;
```

Run

- Introduction
- JavaScript Basic
- JavaScript & DOM & CSS
- Event Handling
- Web Applications
- Summary

EVENTS

- Okay, JS can make page dynamic
- But, why would a page be dynamic?
- Because some *events* happened in the page
    - Client does something; e.g., click on button
    - There is a periodical update; e.g., check for new emails
    - New content is received from server; e.g., chat message
- Who does know what happened? **Browser**
- Browser runs a JS code for each event (which is usually an empty function)
- How does browser know which function should be called?
    - JS has to *register* itself for the event!

EVENTS & THEIR HANDLING (IN MORE DETAILS)

- There are many predefined events in browsers
    - Mouse click
    - Key press
    - ...
- These events occur on an *element*: e.g., mouse click on a button
    - Not all elements have all events

- DOM Objects have event handler methods
    - Event handler is called by browser when event occur on object
        - E.g., `onclick` is called when object (html element) is clicked
    - Most event handlers are empty by default

EVENT HANDLING IN JAVASCRIPTS

- To do somethings when an event occurs
    - The event handling
- Override the corresponding event handler
- In HTML, using element attribute

```
<tag onclick="\\ JavaScript code or function()">
```

- In JavaScript, using DOM object

```
object.onclick=function
```

SAMPLE EVENTS

Event	Occurs when...
onabort	a user aborts page loading
onblur	a user leaves an object
onchange	a user changes the value of an object
onclick	a user clicks on an object
ondblclick	a user double-clicks on an object
onfocus	a user makes an object active
onkeydown	a keyboard key is on its way down
onkeypress	a keyboard key is pressed
onkeyup	a keyboard key is released
onload	a page is finished loading.
onmousedown	a user presses a mouse-button
onmousemove	a cursor moves on an object
onmouseover	a cursor moves over an object
onmouseout	a cursor moves off an object
onmouseup	a user releases a mouse-button
onreset	a user resets a form
onselect	a user selects content on a page
onsubmit	a user submits a form
onunload	a user closes a page

JAVASCRIPT EVENT HANDLING EXAMPLE

```
<input
  id="clickbtn"
  type="button"
  value="click"
  onmouseover="document.getElementById('msg').innerHTML='Click Here'"
  onmouseout="document.getElementById('msg').innerHTML='Outside! Clicks are ignor
  onmousedown="mouseDown()"
/>
<div id="msg" style="width: 50%"></div>
<script type="text/javascript">
  let counter = 0;
  function mouseDown() {
    counter++;
    document.getElementById("msg").innerHTML = "A new click";
  }

  function mouseUp() {
    window.alert("Total # of clicks =  " + counter);
```

click

HOW TO PASS INPUTS TO EVENT HANDLERS?

- It is easy when the handler is registered in HTML

```
function setBgColor(color) {
  document.getElementById('change-my-color').style.backgroundColor =
}
```

```
<button onclick="setBgColor('red')">Red</button>
<button onclick="setBgColor('green')">Green</button>
```

Red Green

- *Find out* the required information (specially when the handler is registered in JS)

```
color: <input type="text" id="color" />
<button id="set">set</button>
```

```
function setBgColor2() {
  document.getElementById('change-my-color-2').style.backgroundColor =
  document.getElementById("color").value;
}
document.getElementById("set").onclick = setBgColor2;
```

color: set

HOW TO PASS INPUTS TO EVENT HANDLERS (CONT.)?

- Use `this` only when the handler is registered in JS

```
function setBgColor3() {
  window.alert("this = " + this);
  this.style.backgroundColor = this.innerHTML.toLowerCase();
}

document.getElementById("r").onclick = setBgColor3;
document.getElementById("b").onclick = setBgColor3;
```

```
<button id="r">Red</button>

<button id="b">Blue</button>

<button onclick="this.style.backgroundColor =
this.innerHTML;">Green</button>

<!-- how we can fix this? -->
<button onclick="setBgColor3();">Black</button>
<button onclick="setBgColor3.bind(this)();">Black</button>
```

Red Blue Green Black Black

DYNAMIC EVENT HANDLER REGISTRATION

- Remark, event handlers can be registered by JavaScript

```
object.onclick = function
```

- Other methods to dynamically register event handlers
  - Assign multiple event handlers
  - Reset event handler to default
- Add event handler

```
object.addEventListener(eventName, function)
```

- Remove event handler

```
object.removeEventListener(eventName, function)
```

DYNAMIC EVENT HANDLER EXAMPLE

```
<button onclick="AddEventHandler();">
  Add a 'click' event listener to the blue button
</button>

<button onclick="RemoveEventHandler();">Remove the event listener</button>

<button id="blueButton" style="background-color: #0077ff">Big Blue Button</button

<script type="text/javascript">
  function blueClick1() {
    alert("You have clicked on me!!!");
  }

  function blueClick2() {
    alert("Yahoooo!!!!");
  }

  function AddEventHandler() {
```

Add a 'click' event listener to the blue button | Remove the event listener | Big Blue Button

EVENT PROPAGATION & THE `event` OBJECT

- What happen if both parent and child handle the same event?!
    - The event is said to propagate outward, from the node where it happened to that node's parent node and on to the root of the document.
- By default, the event object is passed to all event handler
    - At any point, an event handler can call the `stopPropagation` method on the event object to prevent handlers further up from receiving the event.

```html
<div id="parent" style="border: solid">
  I am parent <br />
  <button id="click-btn-1">click me</button>
</div>
```

```javascript
// counter have been declared before
counter = 0;
function childHandler(event) {
  window.alert("child handler");
  counter++;
  // add message to the event
  event.message = `You have clicked me ${counter} times`;
}

document
  .getElementById("click-btn-1")
  .addEventListener("click", childHandler);

document.getElementById("parent").addEventListener("click", (event) => {
  window.alert(
    `child message that is added into event: ${
      event.message ?? "use clickme button"
    }`
```

I am parent
click me

HTML ATTRIBUTE

```
<button onclick="alert('Hello world!')">
```

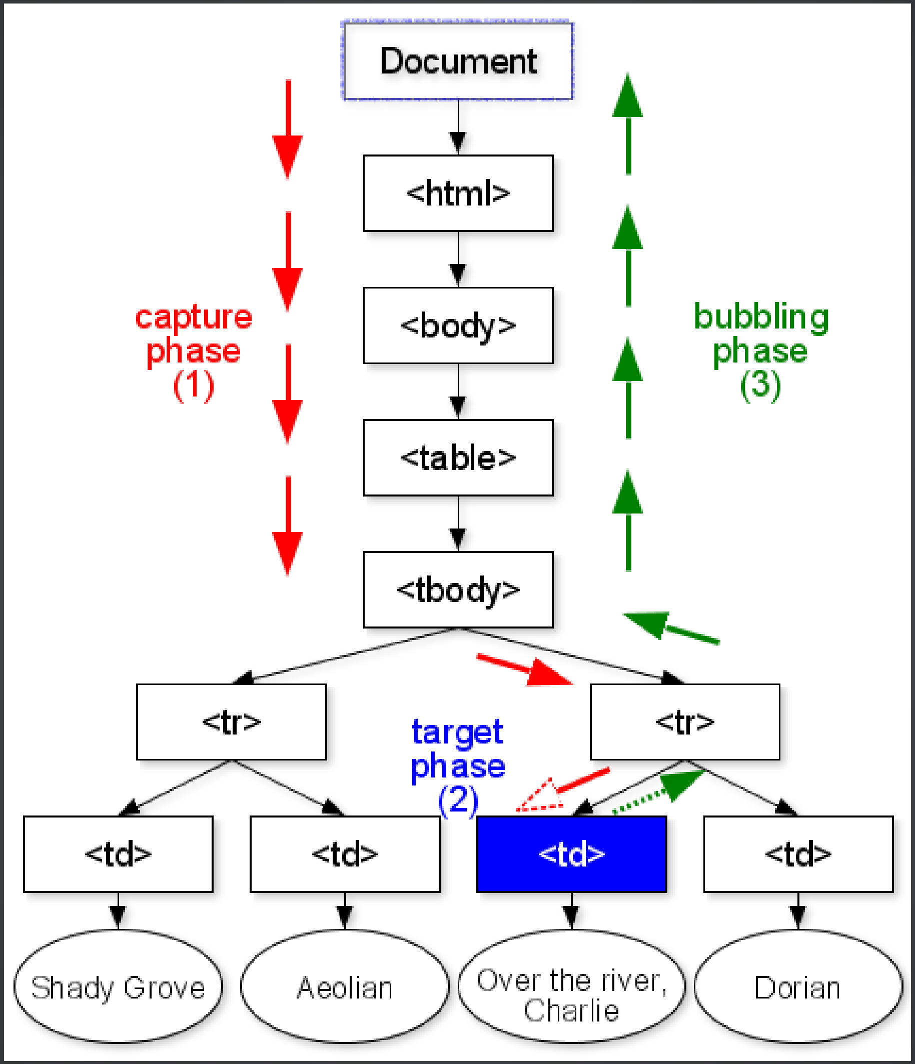
The JavaScript code in the attribute is passed the Event object via the `event` parameter.

DOM ELEMENT PROPERTIES

```
// Assuming myButton is a button element
myButton.onclick = function(event){alert('Hello world')}
```

The function can be defined to take an `event` parameter.

- The Event interface represents an event which takes place in the DOM.
- `Event.target` A reference to the target to which the event was originally dispatched.
- `Event.type` The name of the event. Case-insensitive.


Document
<html>
<body>
<table>
<tbody>
<tr>
<tr>
<td>
<td>
<td>
<td>
Shady Grove
Aeolian
Over the river, Charlie
Dorian
capture phase (1)
bubbling phase (3)
target phase (2)

event.target VS. this

- There is a difference between this and event.target, and quite a **significant** one.
- this always refers to the DOM element the listener was attached to
- event.target is the actual DOM element that was clicked

- Introduction
- JavaScript Basic
- JavaScript & DOM & CSS
- Event Handling
- Web Applications
- Summary

APPLICATION 1: DYNAMIC ELEMENT GENERATION

- Add/Remove objects to/from DOM using JavaScript
- Create an element

```
document.createElement(tagName)
```

- Set or Get attribute

```
element.getAttribute(name)
element.setAttribute(name, value)
```

- Append & Insert the new element as a child of parent

```
parent.appendChild(child)
parent.insertBefore(newChild, existingChild)
```

- To delete or replace child

```
parent.removeChild(child)
parent.replaceChild(newChild, oldChild)
```

CONTENT GENERATION EXAMPLE

Add New Paragraph Replace New Paragraph

```
let id = 0;
function addNewP() {
  id++;
  let parent = document.getElementById("main");
  let newp = document.createElement("p");
  newp.id = "newp" + id;
  newp.innerHTML = "I am new paragraph";
  parent.appendChild(newp);
}
```

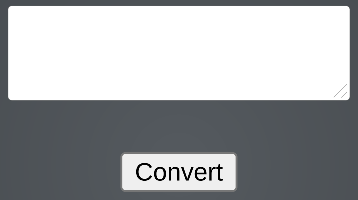
```
let replace = 0;
function replaceNewP() {
  replace++;
  if (replace <= id) {
    let parent = document.getElementById("main");
    let newp = document.getElementById("newp" + replace);
    let newer = document.createElement("span");
    newer.style.borderStyle = "solid";
    newer.innerHTML = "I replace the new paragraph";
    parent.replaceChild(newer, newp);
  } else {
    replace = id;
  }
}
```

APPLICATION 2: FORM VALIDATION

- One of the major applications of JavaScript: *Form Validation*
  - To check input data correctness before submitting to server
  - To save bandwidth, time, and server load
- *Note*: Data **cannot** be validated completely in client-side
  - In client side we check *format* (pattern/syntax)
    - Checking format, e.g., Date or Time pattern
    - Checking length, e.g. Password length
- Steps:
  1. Read the input
  2. Check the format/pattern (by regular expression)
  3. Don't allow to submit if there is an error

FORM VALIDATION IN JAVASCRIPT

- At first, we should access to input (form) data, then validate
- For `text`, `password`, and `textarea`
    - The input text is accessible via `.value` of the corresponding object
- For `select`
    - The value of selected `option` is given by `SelectObject.value`
- For `checkbox` and `radio`
    - Enumerate all options (children)
    - `.checked == true` is selected
    - The value of the selected child is given by its `.value`



Output will be here:

```
function upperCaseArea(textareaID, outputID) {
  return () => {
    let textareaObject = document.getElementById(textareaID);
    let content = textareaObject.value.toUpperCase();
    let outputObject = document.getElementById(outputID);
    let outputMessage = content;
    outputObject.innerHTML = `<code class="hl-orange">${outputMessage}</code>`;
  };
}

document.getElementById("btnID").onclick = upperCaseArea("txtID", "outID");
```

JAVASCRIPT FORM VALIDATION EXMAPLE

☐ Linux, ☐ Windows, ☐ Mac, ☐ Unix

Find OS

```
function findOS(inputCheckBoxs, Outputdiv) {
  let boxes = document.getElementsByName(inputCheckBoxs);
  let outputMessage = "Ok, you are master in ";
  for (let i = 0; i < boxes.length; i++) {
    if (boxes[i].checked) outputMessage += " " + boxes[i].value + ", ";
  }
  document.getElementById(Outputdiv).innerHTML = outputMessage;
}
```

FORM VALIDATION BY REGULAR EXPRESSION

```
reg_expr = /expression/;

// The test() method executes a search for a match between a regular expression ar
// Returns true or false.
reg_expr.test(string)

// The match method retrieves the matches when matching a string against a regular
string.match(reg_expr)
```

`c*`	≥ 0 of c	`c+`	≥ 1 of c
`c?`	0 or 1 of c	`c{x}`	x times of c
`.`	A char (no new line)	`c1|c2`	c1 or c2
`[]`	Any combination of given characters	`[^]`	Any string without the given characters
`\d`	A digit	`\D`	Every thing except digits
`^c`	Beginning match	`c$`	End match

```
let m = /^ab*c+d{3}z$/;
// "acz": False
// "abbccdddz": True
// "ffabbccdddz": False

let m = /ab*c+d{3}z/;
// "abbccdddz": True
// "ffabbccdddz": True
// "ffabbccdddzggg": True

// To match Date format:
let dateRegex = /^\d{4}\/\d{1,2}\/\d{1,2}$/
```

VALID FORM SUBMISSION

Password Manager

User name:	
New Password :	
Reenter New Password:	

Change Password

```javascript
function checkpassword(event) {
  let p1 = document.passwordform.password.value;
  let p2 = document.passwordform.repassword.value;

  if (p1.length < 6) {
    alert("To short password, re-enter");
    event.preventDefault();
  } else if (p1 == p2) {
    alert("Password will be changed");
  } else {
    alert("Incorrect password");
    event.preventDefault();
  }
}
document.getElementsByName("passwordform")[0].onsubmit = checkpassword;
```

`Event.preventDefault`

The Event interface's preventDefault() method tells the user agent that if the event does not get explicitly handled, its default action should not be taken as it normally would be.

Toggling a checkbox is the default action of clicking on a checkbox.

```
document.querySelector("#id-checkbox").addEventListener("click", function(event) 
  document.getElementById("output-box").innerHTML += "Sorry! <code>preventDefault
  event.preventDefault();
});
```

```
<p>Please click on the checkbox control.</p>

<form>
  <label for="id-checkbox">Checkbox:</label>
  <input type="checkbox" id="id-checkbox"/>
</form>

<div id="output-box"></div>
```

Please click on the checkbox control.

Checkbox: ☐

HTML VALIDATOR

age:

email:

APPLICATION 3: WORKING WITH DOCUMENT

- `document` object is created by browser for each HTML page (document) that is viewed
- Its properties provide useful information to read/write about the HTML document

`anchors`, `applets`, `forms`, `images`, `links`	Array of different types of HTML elements
`body`	The object corresponding to `<body>`
`dir`	Document direction
`title`	Title of the document
`cookie`, `location`, `domain`, ...	Information about HTTP

DOCUMENT OBJECT EXAMPLE

`referrer`	https://1995parham-teaching.github.io/ie-lecture/
`URL`	https://1995parham-teaching.github.io/ie-lecture/lectures/lecture-5/?print-pdf
`location.protocol:`	https:
`domain:`	1995parham-teaching.github.io
`location.pathname:`	/ie-lecture/lectures/lecture-5/

Change Sides

من یک متن فارسی هستم

APPLICATION 4: WORKING WITH BROWSER

- The `window` objects provide useful properties and methods to work with browser window
    - Properties to access browser window size
    - Methods & properties to work browser history
    - Method to open/close/change browser window
    - Method to run periodic functions

WINDOW OBJECT

- The `window` object is created for each window/tab that appears on the screen
- Major properties

`document`	This is the `document` object (that we have seen)
`history`	Provides information on the browser history of the current window; method to go forward and backward in the history
`*Height` `*Width`	Height & width of window or screen
`location`	URL of the window

WINDOW PROPERTIES

- Window Size (in pixel)
- Please resize to see what happens

`screen.height:`	1080	`screen.width:`	19
`screen.availHeight:`	1053	`screen.availWidth:`	18
`outerHeight:`	0	`outerWidth:`	0
`innerHeight:`	948	`innerWidth:`	18

- Zoom in/out decreases/increases height/width because it makes pixels bigger/smaller
- History length: 1
- Location: https://1995parham-teaching.github.io/ie-lecture/lectures/lecture-5/ print-pdf

```
let tooSmall = 0;
let content;

window.onresize = function () {
  console.log(`new window ${window.innerWidth} x ${window.innerHeight}`);
  console.log(`small: ${tooSmall}`);

  if (
    tooSmall == 0 &&
    (window.innerHeight < 250 || window.innerWidth < 500)
  ) {
    tooSmall = 1;
    let el = document.getElementsByName("window-properties")[0];
    if (el != null) {
      content = el.innerHTML;
      el.innerHTML = "";

      msg = document.createElement("h1");
```

WINDOW OBJECT METHODS

`forward(), back()`	One time forward or back in history
`stop(), close()`	Stop page loading or close it
`open()`	Create new window

Alert "Hello" every 3 seconds (3000 milliseconds)

```
let hello = setInterval(() => alert("Hello"), 3000);
clearInterval(hello)
```

Set Clear

Alert "Hello" after 3 seconds (3000 milliseconds)

```
let hello = setTimeout(() => alert("Hello"), 3000);
clearTimeout(hello)
```

Set Clear

NAVIGATOR OBJECT

- Checking browser software
- `navigator`
  - Properties to read browser characteristic

`appCodeName:`	Mozilla
`appName:`	Netscape
`appVersion:`	5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36
`language:`	en-US
`platform:`	Linux x86_64
`userAgent:`	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36

APPLICATION 5: JAVASCRIPT ACCESS TO COOKIES

- Saving client preferences
- Remark: Cookie is a solution for HTTP statelessness problem
    - Server sets cookies on first connection: name=value
    - Client returns back the cookies in subsequent communications
    - Server knows who the client is and provides its state
- Cookies can be used by JavaScript as small storage in *client side!*

- because browser
  - stores & loads them automatically
  - provides API to access them by JS
- *Note*: disrespect of originator, cookies are sent to server automatically!

JAVASCRIPT ACCESS TO COOKIES

- All cookies (set by server or JS itself) are accessible by JavaScript
    - Except the cookies with `httpOnly` attribute
- Extremely easy approach to read or write cookies
    - Cookies are saved in `document.cookie` as a string
    - To add a cookie, set "name=value" string in `document.cookie`
    - To read a cookie, parse `document.cookie`
- *Note*: Chrome by default ignores cookies set by *file* schema.

LOCAL STORAGE

- The `window.localStorage` object stores the data with no expiration date.
- Name/value pairs are always stored as strings. Remember to convert them to another format when needed

```
// store
localStorage.setItem("lastname", "Smith");

// retrieve
document.getElementById("result").innerHTML = localStorage.getItem("lastname");

// remove
localStorage.removeItem("lastname");
```

SESSION STORAGE

- The `window.sessionStorage` object is equal to the localStorage object, except that it stores the data for only one session.
- The data is deleted when the user closes the specific browser tab.

APPLICATION 6: FETCH

- The `Fetch API` provides a JavaScript interface for accessing and manipulating parts of the `HTTP pipeline`, such as requests and responses.
- It also provides a global `fetch()` method that provides an easy, logical way to fetch resources **asynchronously** across the network.

- 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, whether it is successful or not.

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

- Here we are fetching a JSON file across the network and printing it to the console.
- The simplest use of fetch() takes one argument — the path to the resource you want to fetch — and returns a promise containing the response (a Response object).
    - This is just an HTTP response, not the actual JSON.
    - To extract the JSON body content from the response, we use the json() method (defined on the Body mixin, which is implemented by both the Request and Response objects.)

```
fetch('flowers.jpg')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.blob();
  })
  .then(myBlob => {
    myImage.src = URL.createObjectURL(myBlob);
  })
  .catch(error => {
    console.error('There has been a problem with your fetch operation:', error);
  });
```

- A fetch() promise will reject with a TypeError when
  - a network error is encountered
  - CORS is misconfigured
- The ok read-only property of the Response interface contains a Boolean stating whether the response was successful (status in the range 200-299) or not.

```
function swapiExec() {
  fetch("https://swapi.dev/api/people/1/")
    .then((resp) => resp.json())
    .then(
      (data) =>
        (document.getElementById(
          "swapi-result"
        ).textContent = JSON.stringify(data))
    );
}
```

```
Waiting...
```

Fetch

```
function wikiSWLogo() {
  fetch(
    "https://upload.wikimedia.org/wikipedia/commons/thumb/6/6c/Star_Wars_Logo.svg/
  )
    .then((resp) => resp.blob())
    .then((content) => {
      let img = document.createElement("img");
      img.src = URL.createObjectURL(content);
      document
        .querySelector("section.present > section.present")
        .appendChild(img);
    });
}
```

Logo!

```
function fetchWithErr(url) {
  fetch(url)
    .then((response) => {
      if (!response.ok) {
        throw new Error(
          "Network response was not ok, " + response.status
        );
      }
      return response.text();
    })
    .then((content) => {
      document.getElementById(
        "errMessage"
      ).innerHTML += `Success: <code class="hl-orange">${content}</code><br />`;
    })
    .catch((error) => {
      document.getElementById(
        "errMessage"
```

403 UUID CORS

JAVASCRIPT IN ACTION WITH AN EMAIL CLIENT

- Add new item per new received email
  - `document.createElement()`, *`parent`*`.appendChild()`
- Check email format when composing
  - *`regex`*`.test(`*`input`*`.value())`
- Information about HTTP (e.g., HTTP or HTTPS?)
  - `document.`*`property corresponding to HTTP headers`*
- Hide some elements when browser window is small
  - `window.*.height/width`

- Do something periodically (e.g., checking emails)
  - `window.setInterval()`
- Redirect to other pages
  - `window.location`
- Check browser software (e.g., to recommend better one)
  - `navigator.userAgent/appVersion/...`
- Save client preferences (e.g., remember me, theme, ...)
  - `document.cookie`

- Introduction
- JavaScript Basic
- JavaScript & DOM & CSS
- Event Handling
- Web Applications
- Summary

DEBUGGING 🐛

- Use the DOM tab to find methods & properties
- Basic debugging via console, `console.log()/info()/warn()/error()`
- Advanced debugging via real debuggers; e.g., FireFox debugger
- Best Practices
    - Avoid Global Variables
    - Always Declare Local Variables
    - Beware of Automatic Type Conversions
    - Use === Comparison
- Use powerful IDEs for development

WARNINGS ⚠️

- JavaScript is a big, complex language
  - It's easy to get started in JavaScript, but if you need to use it heavily, must invest more time in learning it well
- JavaScript is not totally platform independent
  - Expect different browsers to behave differently
- this is a bit confusing!
  - this referes to the owner of the executing function (in most cases it the is "document" or "window" object!!!, not the element)

WHAT NEXT?!

- JavaScript Libraries
    - To make life easier
    - jQuery, Modernizr, MooTools, ...
- JavaScript Frameworks
    - To make life easier even more
    - Angular JS, Vue.js, React.js, Ember.js, ...
- JavaScript server-side programming!!!
    - Node.js, Hapi.js, Socket.io, Meteor.js, ...

WHAT NEXT?!

- Other languages: TypeScript
    - Object oriented
    - Static typing
    - Supports modules and interfaces
    - Transpiled into JavaScript
    - Safe in complex large applications
- Other languages: C, C++, ... 😱
    - Set of tool-chains (compiler, library, VM, ...) to run other languages in client side
    - asm.js and WebAssembly
    - High performance

REFERENCES 📚

- Introduction to the DOM
- JavaScript: Wrapper objects
- Prof. Bahador Bakhshi's Internet Eng. Course's Slides
- What's in an Interpretation?