Front-end <Web /> Development

Lesson 9: JS (Part 3)

University of Tehran

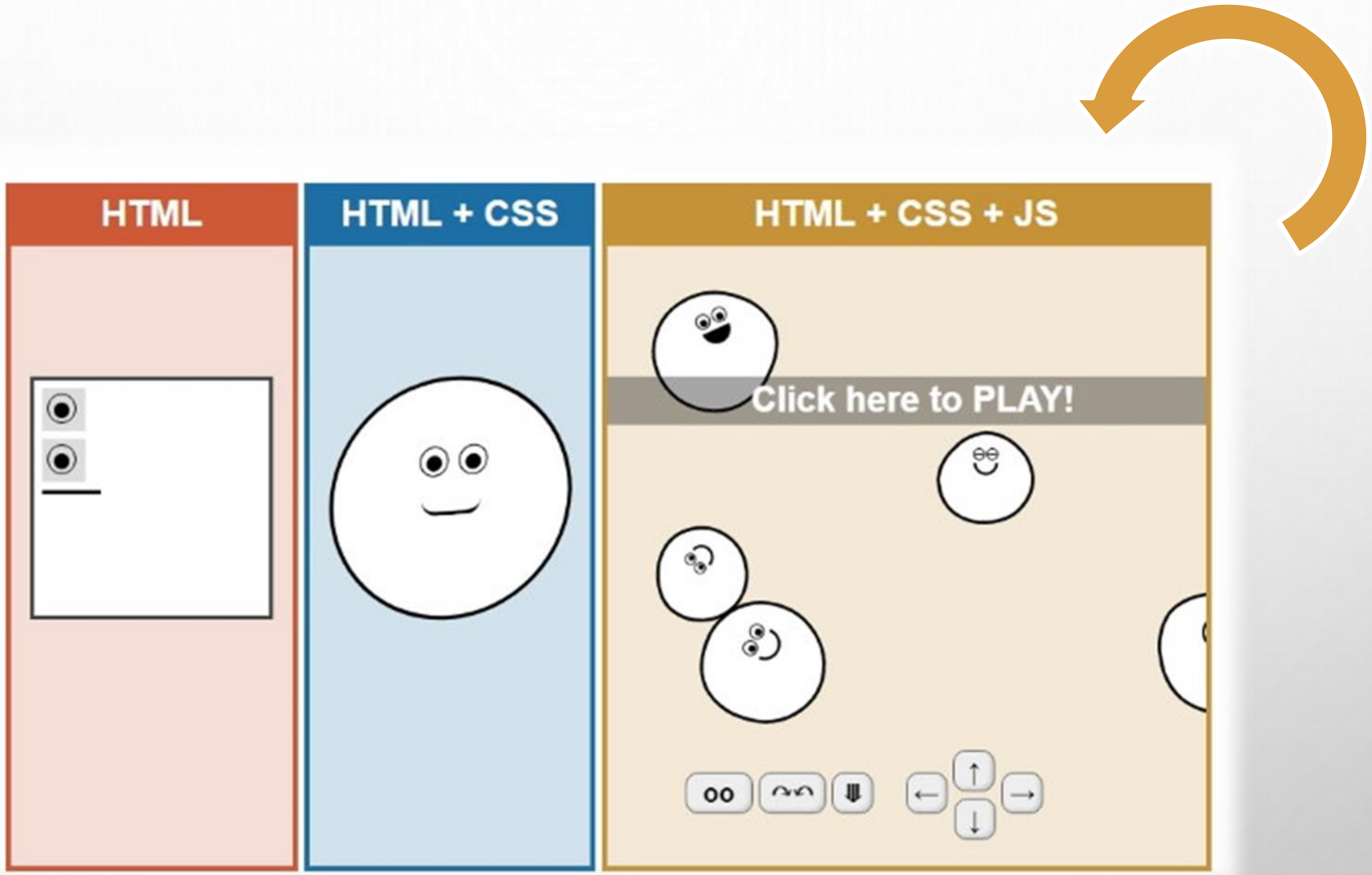
ACM
Summer of Code 2024

Lecturer:
Misagh Mohaghegh

A Short Review…

- Strings, length, and concatenation
- Template strings (multi-line, interpolation)
- Substring, slice, and toUpperCase
- Trim, padding, and repeat
- Search, replace, and split
- Number, BigInt safe rage, NaN and Infinity
- Number toString and from string
- Array and looping methods (for i, for x of arr, forEach)
- Push, pop, unshift, shift, spread operator
- Array toString and join, splice and isArray
- BOM and DOM navigation and manipulation
- Events

Continue…



JavaScript

Local Storage

Storages

There are many storages in JavaScript that can be used to store data in the browser:

- Cache Storage (used for caching)
- Cookies (key-value pairs that are sent to the server on each request)
- Indexed DB (a local NoSQL database)
- Local Storage (a key-value storage)
- Session Storage (a local storage that expires when the tab is closed)

Web Storage API

The Web Storage API provides 2 mechanisms to store key-value pairs in the browser:

- Local Storage
- Session Storage

The local storage can be accessed via the **window.localStorage** object.
The session storage is accessed via the **window.sessionStorage** object.
They both return a Storage object instance and have the same methods, but are different storages.

Local storage persists even when the browser is closed.
Session storage is limited to when the tab is open (including reloads).

The maximum size of each Storage is 5MB.

Local Storage

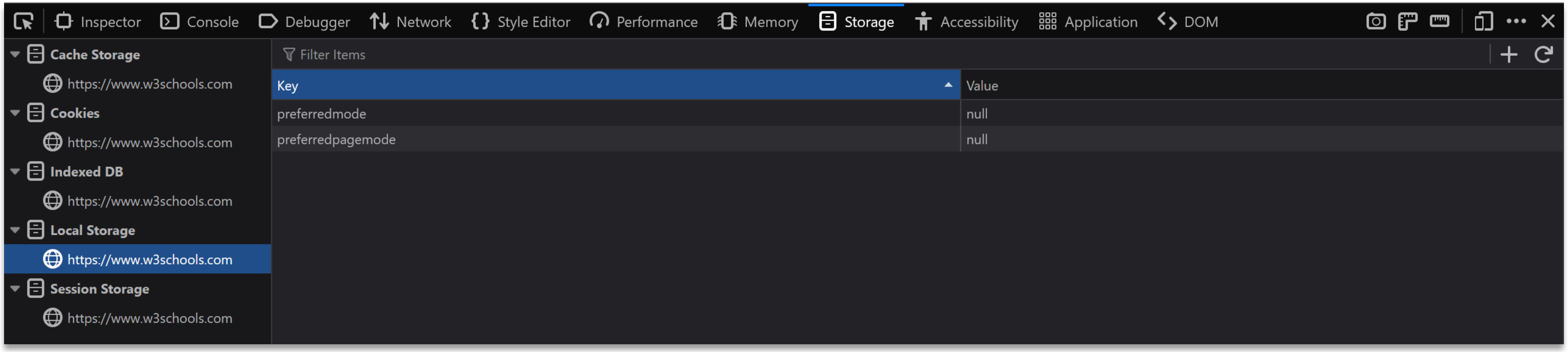
Each origin will have its own local storage.

The methods to use the Storage are as follows:

```
localStorage.setItem("name", "Misagh");
localStorage.getItem("name");
localStorage.removeItem("name");
localStorage.clear();
localStorage.length;
```

Storage in Dev Tools

We can check the contents of the storages using the browser dev tools (inspect):



Key	Value
preferredmode	null
preferredpagemode	null

Object

Objects

We have already seen how we can define and use objects in JavaScript:

```javascript
const person = {
    firstName: "Misagh",
    lastName: "Mohaghegh",
    age: 22,
    fullName: function() {
        return this.firstName + " " + this.lastName;
    }
};

console.log(person.age, person.fullName(), person["firstName"])
```

Object Properties

We use the 'this' keyword to access the object inside a function.

Object **properties** are all fields of an object (firstName, ...).

Object **methods** are just functions stored in properties (fullName).

```
const person = { name: "Misagh" };

person.country = "Iran"; // adding a new property
delete person.country; // removing a property

const copy = person; // this does not copy
copy.name = "test";
person.name == "test";
```

Nested Objects

Objects can be nested:

```
const person = {
    name: "feliks",
    wife: { name: "kate" },
    kids: [
        { name: "alex" },
        { name: "akane" }
    ]
};

console.log(person.wife.name, person["wife"]["name"], person.kids[0].name);
```

Methods

There are 2 ways to define methods in an object:

```
const person = {
    name: "feliks",

    greet: function() { return "I am " + this.name; },
    greet: () => "I am " + this.name,

    introduce() { return "I am " + this.name; },
};
```

Looping

We can loop over all object properties:

```
for (let k in person) {
    console.log(`Key: ${k} - Value: ${person[k]}`);
}

for (let v of Object.values(person)) { // there is also Object.keys(person)
    console.log("Value:", v);
}

for (let [k, v] of Object.entries(person)) {
    console.log(k, v);
}
```

Accessors (Getter and Setter)

Accessors or computed properties can be defined.

These allow us to execute some logic for getting and setting a field.

```
const person = {
    name: "feliks",
    language: "en",
    get lang() { return this.language; },
    set lang(newLang) { this.language = newLang; }
};

person.lang == "en"
person.lang = "fa";
```

Accessors vs Functions

The example below demonstrates the difference:

```
const person = {
    name: "feliks",
    get about() { return "I am: " + this.name; },
    about2: function() { return "I am: " + this.name; }
};

console.log(person.about, person.about2())
```

Object Constructor Function

Sometimes we need to create objects of the same type which have the same properties.

We can use constructor functions for that. They usually start with uppercase letter.

The 'new' keyword is used to create an object from a constructor.

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

// create Person object instances:
const me = new Person("Misagh", 22);
const you = new Person("Kyle", 24);
```

New Object Properties

After creating new objects, they are no longer related and changing or adding or deleting properties from one instance does not affect the other instances:

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

const me = new Person("Misagh", 22);
const you = new Person("Kyle", 24);
me.hairColor = "blue";
you.hairColor == undefined
```

Adding New Properties

When adding new properties to an object instance, the other instances won't have it.

To do that, we must add the property to the object constructor prototype:

```
function Person(name) { this.name = name; }

const a = new Person("test");
a.newProperty = 12; // other instances don't have this

Person.newProperty = 12; // does not work
Person.prototype.newProperty = 12;
Person.prototype.introduce = function() { return "I am " + this.name; };

a.introduce();
```

Instance of

To check if an object type variable is created from an object constructor, we use the 'instanceof' keyword:

```
const a = new Person("name");
typeof a == 'object'
if (a instanceof Person) {
    console.log("Yes");
}
a instanceof Object == true
```

Built-in Object Constructors

There are many built-in object constructors but some of them shouldn't be used:

```
new Object() // use {}
new Array() // use []
new RegExp() // use /regex/
new Function() // use () => {}

new Map()
new Set()
new Date()

// "", 0, false are primitive types and not object types
```

Built-in Primitive Constructors

The JS primitive types also have a corresponding object constructor.

They should not be used to just get a primitive value.

```
const a = new Number(10);
const b = Number(10);

typeof a == 'object'
typeof b == 'number'

a instanceof Number; // true
b instanceof Number; // false
```

Prototype

All objects have a prototype chain. Whenever we access a property, first the actual object properties are searched. If it is not found, the prototype object's properties are searched.

```javascript
function Person(name) {
    this.name = name;
    this.greet = () => "Hi!";
}

// this will create the greet method every time we make an instance of Person
const a = new Person("a");
const b = new Person("b");
// they both have their own greet method which does the same thing
// we can factor the method out into the Person prototype
```

Prototype

To add a method to the object prototype:

```
function Person(name) {
    this.name = name;
}
Person.prototype.greet = () => "Hi!";

const a = new Person("a");
a.greet();

Object.getPrototypeOf(a) // Person.prototype { greet: ... }
```

Prototype

Setting the entire prototype object:

```
const personPrototype = {
    greet: function() { return "Hi!"; }
}
function Person(name) {
    this.name = name;
}

Object.assign(Person.prototype, personPrototype);

const a = new Person("a");
a.greet();
```

Own Property

We can use the Object.hasOwn method to see if a property is for the object constructor:

```
function Person(name) {
    this.name = name;
}
Person.prototype.greet = () => "Hi!";

const a = new Person("a");
Object.hasOwn(a, "name") // true
Object.hasOwn(a, "greet") // false
```

Own Property

The for … in loop iterates the prototype as well.

```
function Person(name) {
    this.name = name;
}
Person.prototype.greet = () => "Hi!";
const a = new Person("a");

for (let x in a) { console.log(x); } // name, greet
for (let x of Object.keys(a)) { console.log(x); } // name
```

Class

Classes

Classes were introduced in ES6.

They are templates for easier creation of object constructors and prototype inheritance.

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

const a = new Person("a");
```

Methods

Classes can easily contain methods:

```
class Person {
    constructor(name) {
        this.name = name;
    }
    age() { return 22; }
}

const a = new Person("a");
console.log(a.age());
```

Accessors (Getters and Setters)

Accessors can be placed in a class just like objects:

```
class Person {
    constructor(name) {
        this.name = name;
        this._age = 22;
    }
    get age() { return this._age + 1; }
}

const a = new Person("a");
console.log(a.age);
```

Inheritance

Inheritance is easy with classes:

```
class Person {
    constructor(name) { this.name = name; }
    greet() { return "Hi!"; }
}

class Teacher extends Person {
    constructor(name, course) {
        super(name);
        this.course = course;
    }
}
const a = new Teacher("Misagh", "Frontend");
```

Static Methods

Static methods are methods that are defined on the class itself.

They cannot be accessed via a class instance.

```
class Person {
    constructor(name) { this.name = name; }
    static greet() { return "Hi!"; }
}

const a = new Person("a");
a.greet() // error
Person.greet()
```

Static Fields

Static fields are similar to static methods:

```
class Person {
    static test = "Hi!";
    constructor(name) { this.name = name; }
}

const a = new Person("a");
a.test // undefined
Person.test
```

Use Strict

"use strict"

The "use strict" directive was added in ES5.

It is a string literal and is ignored by older versions.

When strict mode is enabled, more errors are caught.

```
"use strict";
x = 12; // error, x is not declared

const obj = {
    get field() { return 10; }
}
obj.field = 12; // error, writing to read-only property
```

Thank you for your attention.