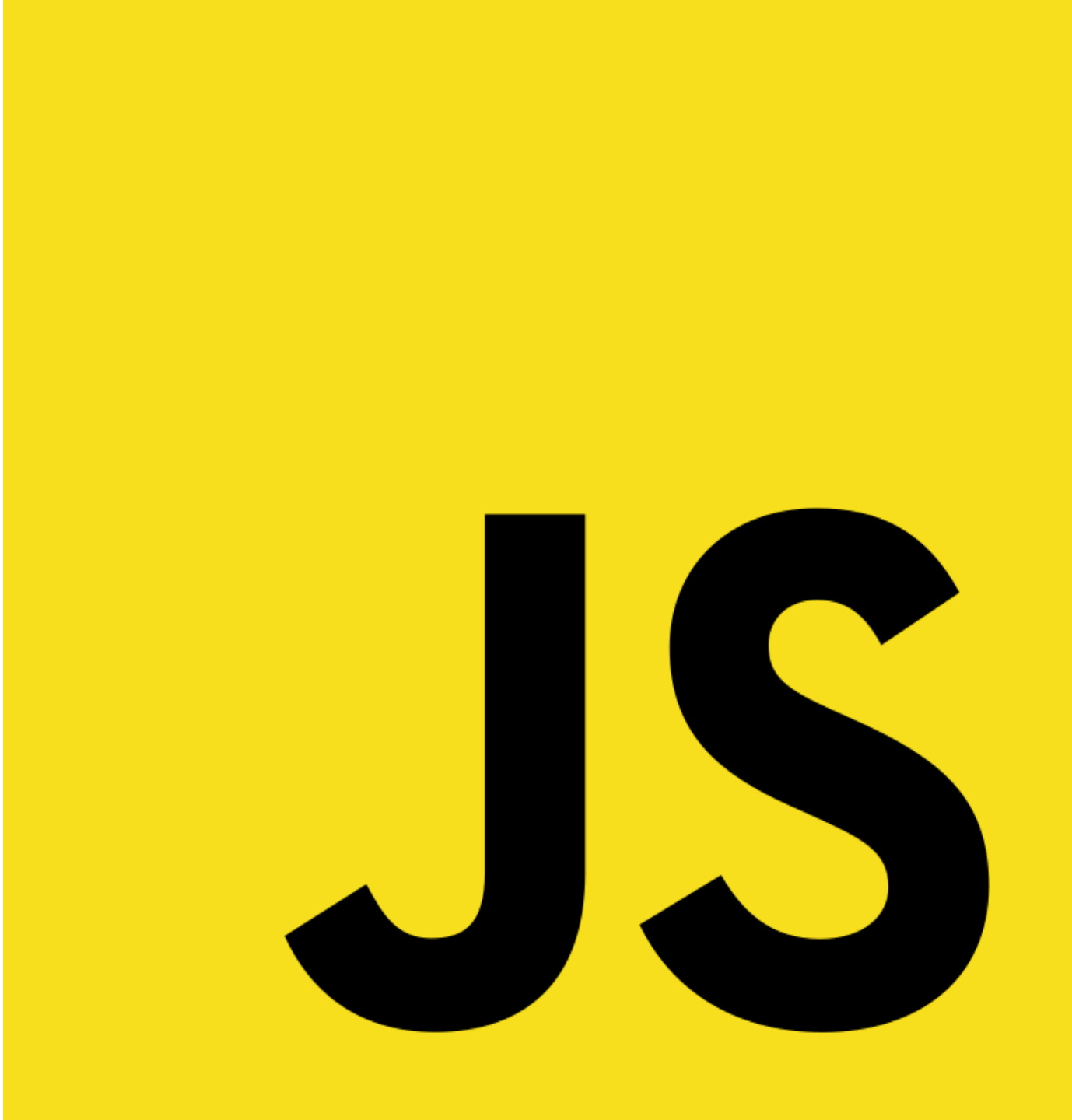
BOOTCAMP ’20

SESIONI **#17**



What will be covered?

- let & const
- Finding HTML Elements
- Error Handling
- Event Handlers
- Template literals
- Object & Array destructuring
- Object literals
- Spread operator
- Rest parameters
- Computed property names

let & const

Deklarimi i variablave:

let variable_name;
ose
let variable_name = value;

Deklarimi i konstantave:

const const_name;
ose
const const_name = value;

keyword	const	let	var
global scope	NO	NO	YES
function scope	YES	YES	YES
block scope	YES	YES	NO
can be reassigned	NO	YES	YES

Finding HTML elements

Method	Description
document.getElementById(*id*)	Find an element by element id
document.getElementsByTagName(*name*)	Find elements by tag name
document.getElementsByClassName(*name*)	Find elements by class name

Finding HTML elements

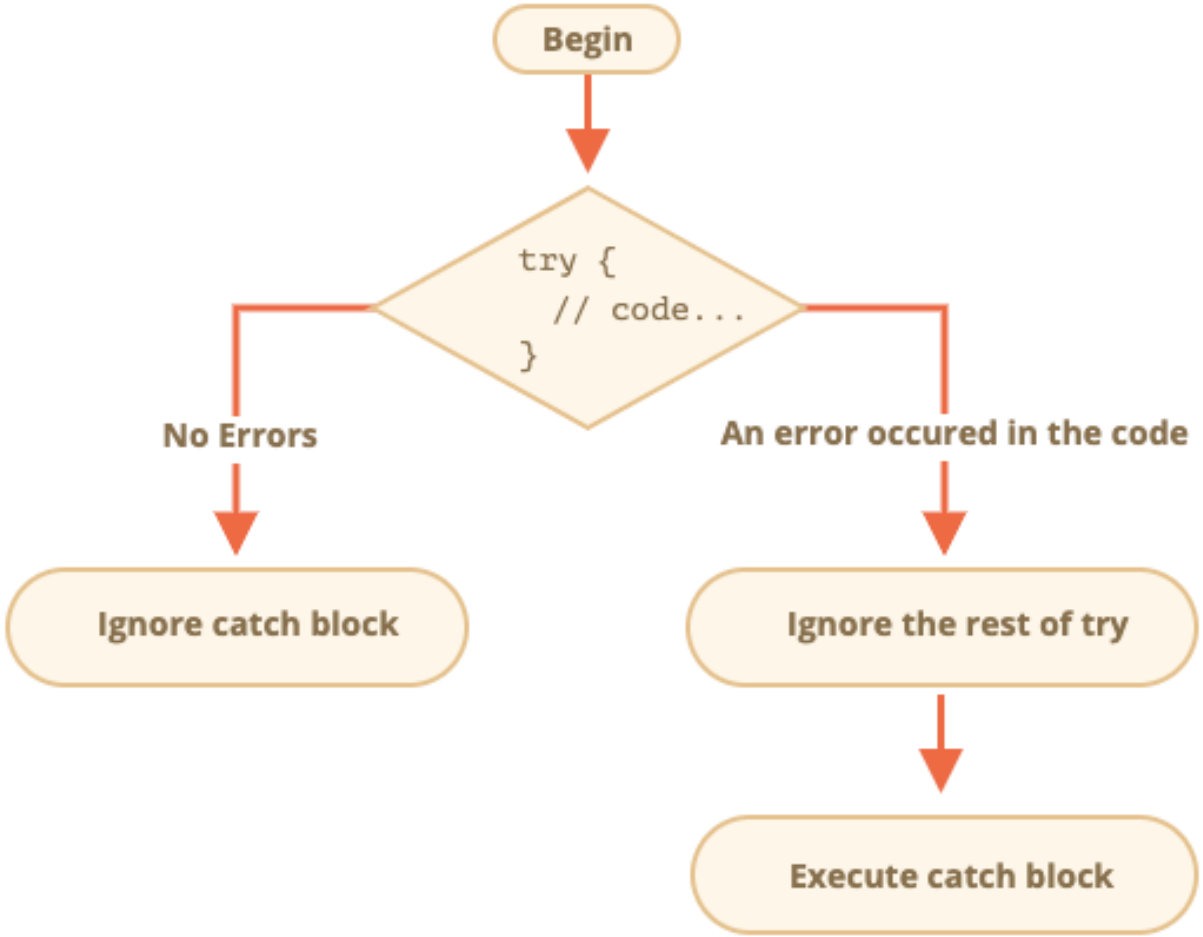
Property	Description
element.innerHTML = *new html content*	Change the inner HTML of an element
element.attribute = new value	Change the attribute value of an HTML element
element.style.*property = new style*	Change the style of an HTML element
Method	**Description**
element.setAttribute*(attribute, value)*	Change the attribute value of an HTML element

Method	Description
document.createElement(*element*)	Create an HTML element
document.removeChild(*element*)	Remove an HTML element
document.appendChild(*element*)	Add an HTML element
document.replaceChild(*new, old*)	Replace an HTML element
document.write(*text*)	Write into the HTML output stream

Error Handling

try, catch, throw, finally

```
try {
    // condition throw "..."
}
catch(err) {
    // display error message
}
finally {
    // always executed
}
```



Error Handling

try, catch, throw, finally

Six different values can be returned by the error name property:

Error Name	Description
EvalError	An error has occurred in the eval() function
RangeError	A number "out of range" has occurred
ReferenceError	An illegal reference has occurred
SyntaxError	A syntax error has occurred
TypeError	A type error has occurred
URIError	An error in encodeURI() has occurred

Event Handlers

```
element.addEventListener(event, function, useCapture);
```

The first parameter is the type of the event (like "`click`" or "`mousedown`" or any other HTML DOM Event.)

The second parameter is the function we want to call when the event occurs.

The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

Detaje rreth llojeve të ndodhive (event-eve) mund të gjeni në vegëzën në vijim:
https://developer.mozilla.org/en-US/docs/Web/Events

Event Handlers

Bubbling vs. Capturing

In *bubbling* the inner most element's event is handled first and then the outer: the <p> element's click event is handled first, then the element's click event.

In *capturing* the outer most element's event is handled first and then the inner: the element's click event will be handled first, then the <p> element's click event.

Një shembull konkret:
https://www.w3schools.com/js/tryit.asp?filename=tryjs_addeventlistener_usecapture

Template literals

- Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them.
- They were called "template strings" in prior editions of the ES2015 specification.

```
let name = "Ylber";

document.write("Hello there " + name);

document.write(`Hello there ${name}`);
```

```
let name = "Ylber";

document.write("Hello there " + name +
" some other text");

document.write(`Hello there ${name}
some other text`);
```

Array destructuring

Basic variable assignment

```
1  const foo = ['one', 'two', 'three'];
2
3  const [red, yellow, green] = foo;
4  console.log(red); // "one"
5  console.log(yellow); // "two"
6  console.log(green); // "three"
```

Array destructuring

Assignment separate from declaration

A variable can be assigned its value via destructuring separate from the variable's declaration.

```
let a, b;

[a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2
```

Array destructuring

Default values

A variable can be assigned a default, in the case that the value unpacked from the array is `undefined`.

```
let a, b;

[a=5, b=7] = [1];
console.log(a); // 1
console.log(b); // 7
```

Array destructuring

Swapping variables

Two variables values can be swapped in one destructuring expression.

Without destructuring assignment, swapping two values requires a temporary variable (or, in some low-level languages, the XOR-swap trick).

```
let a = 1;
let b = 3;

[a, b] = [b, a];
console.log(a); // 3
console.log(b); // 1

const arr = [1,2,3];
[arr[2], arr[1]] = [arr[1], arr[2]];
console.log(arr); // [1,3,2]

```

Array destructuring

Parsing an array returned from a function

It's always been possible to return an array from a function. Destructuring can make working with an array return value more concise.

In this example, `f()` returns the values `[1, 2]` as its output, which can be parsed in a single line with destructuring.

```
function f() {
  return [1, 2];
}

let a, b;
[a, b] = f();
console.log(a); // 1
console.log(b); // 2
```

Array destructuring

Ignoring some returned values

You can ignore return values that you're not interested in:

```
function f() {
  return [1, 2, 3];
}

const [a, , b] = f();
console.log(a); // 1
console.log(b); // 3

const [c] = f();
console.log(c); // 1
```

You can also ignore all returned values:

```
[,,] = f();
```

Array destructuring

Assigning the rest of an array to a variable

When destructuring an array, you can unpack and assign the remaining part of it to a variable using the rest pattern:

```
1 | const [a, ...b] = [1, 2, 3];
2 | console.log(a); // 1
3 | console.log(b); // [2, 3]
```

Object destructuring

Basic assignment

```
const user = {
    id: 42,
    is_verified: true
};

const {id, is_verified} = user;

console.log(id); // 42
console.log(is_verified); // true
```

Object destructuring

Assignment without declaration

A variable can be assigned its value with destructuring separate from its declaration.

```
let a, b;

({a, b} = {a: 1, b: 2});
```

Object destructuring

Assigning to new variable names

A property can be unpacked from an object and assigned to a variable with a different name than the object property.

```
1 const o = {p: 42, q: true};
2 const {p: foo, q: bar} = o;
3
4 console.log(foo); // 42
5 console.log(bar); // true
```

Here, for example, `const {p: foo} = o` takes from the object `o` the property named `p` and assigns it to a local variable named `foo`.

Object destructuring

Default values

A variable can be assigned a default, in the case that the value unpacked from the object is `undefined`.

```
1 | const {a = 10, b = 5} = {a: 3};
2 |
3 | console.log(a); // 3
4 | console.log(b); // 5
```

Object destructuring

Assigning to new variables names and providing default values

A property can be both

- Unpacked from an object and assigned to a variable with a different name.
- Assigned a default value in case the unpacked value is `undefined`.

```
1 const {a: aa = 10, b: bb = 5} = {a: 3};
2
3 console.log(aa); // 3
4 console.log(bb); // 5
```

Object literals

A JavaScript object literal is a comma-separated list of name-value pairs wrapped in curly braces. Object literals encapsulate data, enclosing it in a tidy package. This minimizes the use of global variables which can cause problems when combining code.

Object Literal Syntax

Object literals are defined using the following syntax rules:

- A colon separates property name[1] from value.
- A comma separates each name-value pair from the next.
- There should be no comma after the last name-value pair.[2]

Object literals

```
var Swapper = {
    // an array literal
    images: ["smile.gif", "grim.gif", "frown.gif", "bomb.gif"],
    pos: { // nested object literal
        x: 40,
        y: 300
    },
    onSwap: function() { // function
        // code here
    }
};
```

Spread operator

Spread syntax (...) allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

For array literals or strings:

```
[...iterableObj, '4', 'five', 6];
```

For function calls:

```
myFunction(...iterableObj);
```

For object literals (new in ECMAScript 2018):

```
let objClone = { ...obj };
```

Lexo më tepër

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

Rest parameters

The rest parameter syntax allows us to represent an indefinite number of arguments as an array.

Syntax

```
function f(a, b, ...theArgs) {
  // ...
}
```

```
function myFun(a,  b, ...manyMoreArgs) {
  console.log("a", a)
  console.log("b", b)
  console.log("manyMoreArgs", manyMoreArgs)
}

myFun("one", "two", "three", "four", "five", "six")

// Console Output:
// a, one
// b, two
// manyMoreArgs, [three, four, five, six]
```

Computed property names

Computed Property Names is an ES6 feature which allows the names of object properties in JavaScript object literal notation to be determined dynamically, i.e. computed.

```
const myPropertyName = 'c'

const myObject = {
  a: 5,
  b: 10,
  [myPropertyName]: 15
}

console.log(myObject.c) // prints 15
```

```
const fieldNumber = 3

const myObject = {
  field1: 5,
  field2: 10,
  ['field' + fieldNumber]: 15
}

console.log(myObject.field3) // prints 15
```

QUESTIONS