JavaScript basics summary

Variable declarations

Three keywords

- var (mostly used in legacy projects)
- **const**
- **let**

Difference between const and let:

```
let myFirstVar = 'Hello';
myFirstVar = 5; // Can be defined exactly once

const onlyOnce = 'World'; // Can be defined exactly once
onlyOnce = 'Hello'; // Throws a runtime error
```

Data types

Whole list of types: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures

Primitive types

```
// String
const s1 = 'I am a string';
const s2 = "Also a string";
const s3 = `String between backticks`; // template string
```

```
const age = 27;
const text = `I am ${age} years old`; // simple interpolation
const text2 = 'I am ' + age + 'years old';
```

```
// Number
const n1 = 5;
const n2 = 3.4343;
const n3 = Number('2');
const n4 = Number('not a number'); // NaN
// bool
let isValid = true;
isValid = false;
```

Reference types

Object and descendants

```
// Object
const o = {
   name: 'Cubix',
   favoriteNumber: 5
};
// Array
const a1 = [1, 2, 3];
const a2 = [1, 'Cubix', true];
// Function
const f = (a, b) => { return a + b; } // arrow function syntax
```

Value passing

In Javascript value passing (eg.: at a function call, or at a variable assignment) is done by a copy. For primitive types it copies the value while for reference types it copies the reference which points to the data structure. Because of this behaviour we need to be aware of the following situations:

```
let num = 5;
const foo = (n) => {
   n = n * 2; // We modify the copy, the original num variable is
untouched
   return n; // So we shall return the modified value
}

num = foo(num);
```

```
let  pet = {
   name: 'Kitty',
   type: 'cat'
};

const bar = (p) => {
   p = {
       name: 'Doggy',
       type: 'dog'
   } // We are working on a copy reference
   return p; // So we shall return the modified value
};

pet = bar(pet);
```

A másolat referencia az eredeti adatszerkezetre mutat, így, ha az adatszerkezetben módosítunk, akkor az a “pet” változó értékét is módosítja:

```
const bar2 = (p) => {
   p.name = 'Snakey';
   p.type = 'snake';
};

bar2(pet); // We are modifying the value directly on the reference, so we don't have to return
```

Null and undefined

```
let v; // Uninitialized variable's default value is undefined
console.log(v);
v = null;
console.log(v);
v = undefined;
```

Dynamic types

In JavaScript only the values have types, not the variables. This means that a variable can be reassigned multiple times with different values.

```
let d = 5;
d = '5';
d = [1, 2, 3];
d = (i) => {console.log(i)};

d('JavaScript');
```

Closure

An embedded function can reach the variables of it's parents.

```
const makeClosure = () => {
    const name = 'Closure';
    const displayName = () => {
      alert(name);
    }
    return displayName;
  }

const testFn = makeClosure();
testFn();
```

Branches and loops

Whole list of statements: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements

If-else branch

```
const isPositive = n => {
    if (n >= 0) {
        console.log('positive');
    } else {
        console.log('negative');
    }
}

isPositive(5);
isPositive(-5);
```

In JavaScript every value and expression can be used as a boolean value (that's where the concept of **truthy** and **falsy** values appears), so in this example we can use anything as a value for **v**:

```
const isTruthy = v => {
    if (v) {
        console.log('Truthy');
    } else {
        console.log('Falsy');
    }
}

isTruthy({});
isTruthy('');
```

- List of truthy values: https://developer.mozilla.org/en-US/docs/Glossary/Truthy
- List of falsy values: https://developer.mozilla.org/en-US/docs/Glossary/Falsy

Switch-case

```
const mapToStars = n => {
    let res;
    switch (n) {
        case 1: res = '*'; break;
        case 2: res = '**'; break;
        case 3: res = '***'; break;
        case 4: res = '****'; break;
        case 5: res = '*****'; break;
        default: res = '-'
    }

    return res;
}

console.log(mapToStars(2));
console.log(mapToStars(0));
```

Try-catch-throw

If we run a piece a code which can cause a runtime exception then we better wrap it inside a **try** block, then we can add a **catch** block for it, where we can handle the arose error.

```
const throwIfFalsy = v => {
    if (!v) {
        throw new Error('Falsy value');
    }
}

try {
    throwIfFalsy('Truthy');
    console.log('success');
} catch (err) {
    console.error('Caught error', err);
}

try {
    throwIfFalsy(false);
    console.log('success2');
} catch (err) {
    console.error('Caught error2', err);
}
```

For loop

```
for(let i = 0; i <= 4; i++) {
    console.log('For loop', i + 1);
}
```

While loop

```
let i = 0;
while (i <= 4) {
    console.log('While loop', i + 1);
    i++;
}
```

Operators

Whole list of operators: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators

Equation operators

There are 2 types of equation operators in JavaScript:

- **==** : doesn't require type equality, if the types are differ, then it tries to bring the operands to the same type then checks the equality
- **===** : require type equality and value equality

For reference types there's no `deep equal` operator, we can check for reference equality only

```
const a = '5';
const b = 5;
console.log(a == b);
```

```
console.log(a === b);
const p1 = {name: 'Zizi'};
const p2 = {name: 'Zizi'};
console.log(p1 === p2); // There's no deep equal, only reference equality check
const p3 = p1;
console.log(p1 === p3); // Just the reference
```

Arithmetical and value assignment operators

```
let x = 5;
let y = 4;

console.log(x + y); // 9
console.log(x - y); // 1
console.log(x * y); // 20
console.log(x / y); // 1.25
console.log(x % y); // 1

x += y; // x = x + y
console.log(x); // 9

x++ // x = x + 1
console.log(x); // 10
```

Logical operators

Because of **truthy** and **falsy** values concept they don't require a boolean input.

```
console.log(!true);

const e1 = 'Cat' && 'Dog';    // t && t returns Dog
const e2 = false && 'Cat';    // f && t returns false
const e3 = 'Cat' && false;    // t && f returns false

console.log(e1, e2, e3);

const e4 = 'Cat' || 'Dog';    // t || t returns Cat
const e5 = false || 'Cat';    // f || t returns Cat
const e6 = 'Cat' || false;    // t || f returns Cat
const e7 = false || false;    // f || f returns false

console.log(e4, e5, e6, e7);
```

Conditional (ternary) operator — ? :

The **? :** operator expects a logical expression before the **?** and if the expression is **truthy** then it'll executes (or returns with) the first part of the expression (the one between the **?** and **:**) otherwise it'll executes (or returns with) last part of the expression (the one after the **:**).

```
const isAdult = age => {
    let res;
    res = age <= 18 ? false : true;
    return res;
}

console.log(isAdult(16)); // false
console.log(isAdult(22)); // true
```

Delete operator

Delete operator can remove a key of an object, and with the key it'll also removes the value:

```
const person = {
    name: 'John',
    email: 'john@example.com'
}

delete person.email;

console.log(person);
```

Typeof and instanceof operators

typeof operator can be used to determine a type of a **primitive value**, but for reference types it will always returns with **Object** (except for functions, because there will be **Function**). For reference types we can use the **instanceof** operator, which can tell if the given value is instance of a data structure.

```
console.log(typeof 5); // number
console.log(typeof 'a string'); // string
console.log(typeof []); // object
console.log(typeof {}); // object
console.log(typeof new Date()); // object
console.log(typeof (() => {})); // function
console.log([] instanceof Array); // true
console.log([] instanceof Object); // true
console.log({} instanceof Object); // true
console.log(new Date() instanceof Date); // true
```