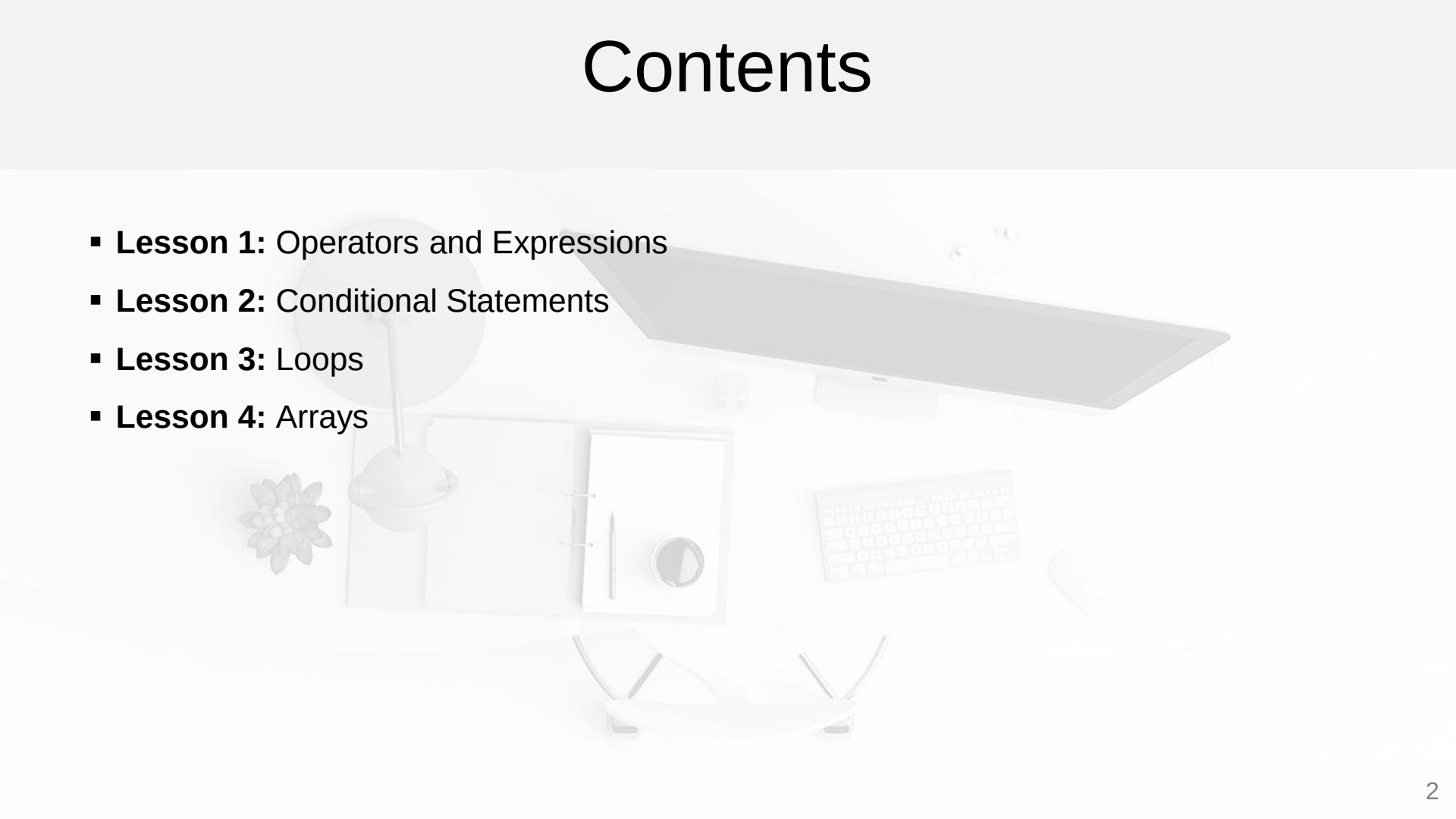
Web Programming

Lecturer: Ung Văn Giàu
Email: giau.ung@eiu.edu.vn

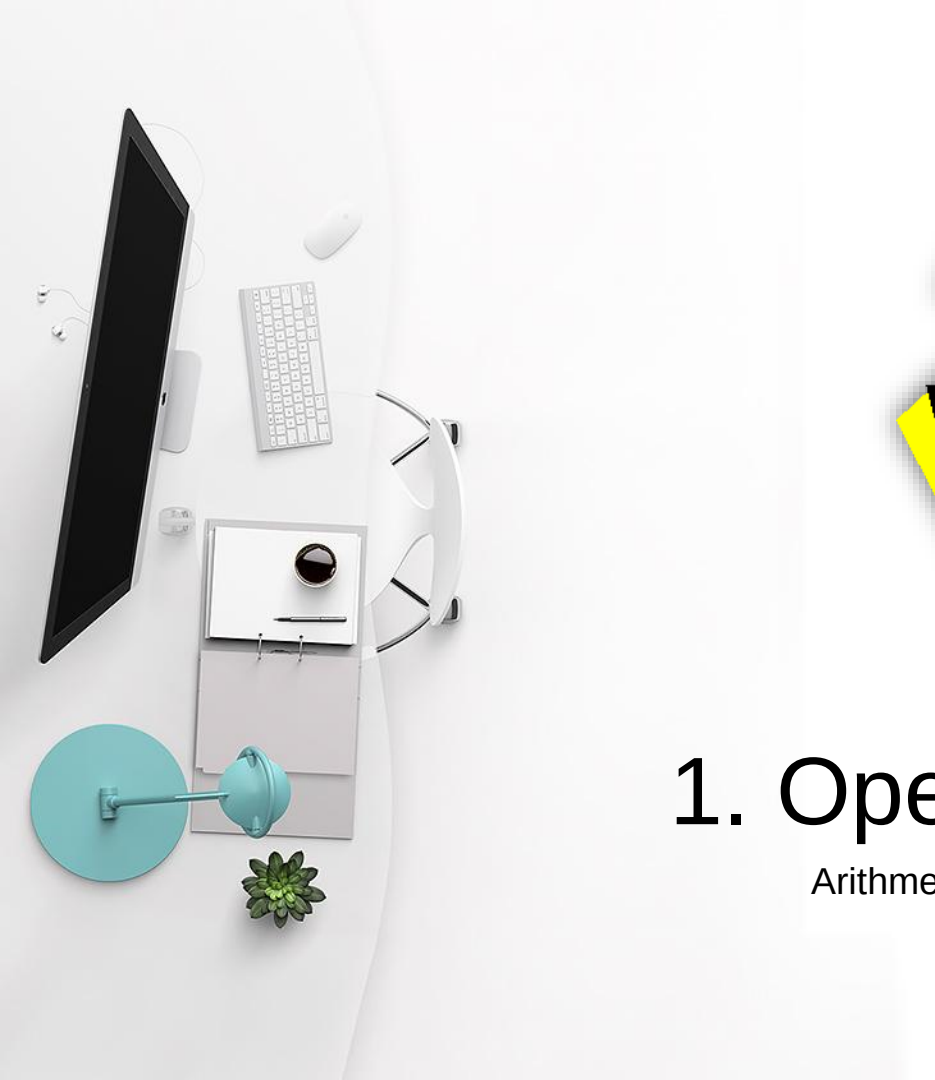
Contents



Operators and Expressions

Performing Simple Calculations with JavaScript

Contents



1. Operators in JavaScript

Arithmetic, Logical, Comparison, Assignment, Etc.

What is an Operator?

- An operator is a **symbol** that **represents** an **operation performed over data** at runtime
  - Takes **one or more arguments** (operands)
  - Produces a **new value**

- Operators have **precedence** (priority)

  Precedence defines which will be evaluated first

- **Expressions** are **sequences of operators and operands** that are **evaluated** to a **single value**

Operators in JavaScript

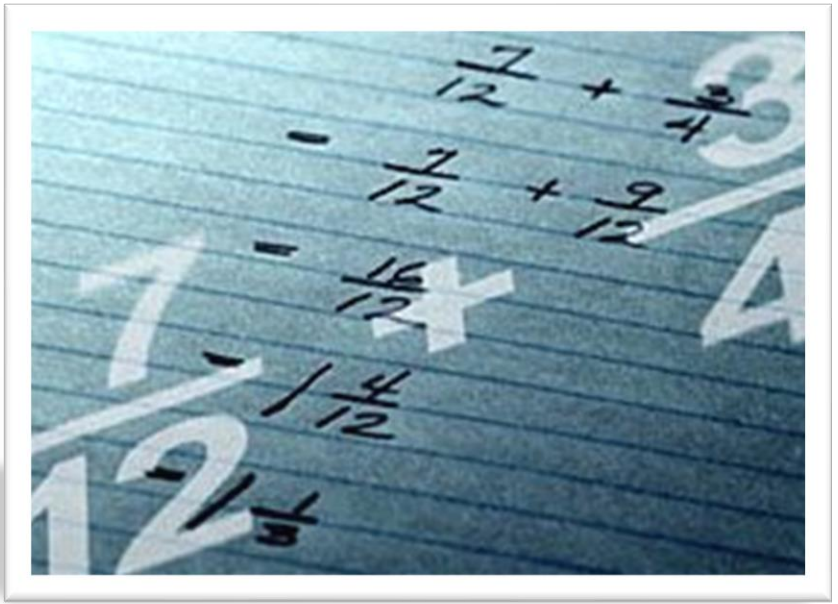
- Operators in JavaScript:
  - **Unary** – take one operand
  - **Binary** – take two operands
  - **Ternary** (?:) – takes three operands (**condition ? A : B**)

- Except for the assignment operators, all binary operators are left-associative

- The assignment operators and the conditional operator (?:) are right-associative

Operators by categories in JavaScript

Category	Operators
Arithmetic	+ – * / % ++ --
Logical	&& || ^ !
Binary	& | ^ ~ << >> >>>
Comparison	== != < > <= >= === !==
Assignment	= += -= *= /= %= |= ^= <<= >>=
Concatenation	+
Other	. [] () ?: new in , delete void typeof instanceof ...



2. Arithmetic Operators

Arithmetic Operators

- Arithmetic **operators +, –, *, /** are the same as in math

- Division **operator /** returns **number** or **Infinity** or **NaN**

  Division in JavaScript returns floating-point number (i.e. 5 / 2 = 2.5)

- Remainder **operator %** returns the **remainder** from division of numbers

  Even on real (floating-point) numbers

- The special addition **operator ++** increments (while **--** decrements) a variable's value

Arithmetic Operators

Example

```
const squarePerimeter = 17;
const squareSide = squarePerimeter / 4;
const squareArea = squareSide * squareSide;

console.log(squareSide); // 4.25
console.log(squareArea); // 18.0625

let a = 5;
let b = 4;

console.log(a + b); // 9
console.log(a + b++); // 9
console.log(a + b); // 10
console.log(a + (++b)); // 11
console.log(a + b); // 11

console.log(12 / 3); // 4
console.log(11 / 3); // 3.6666666666666665
```

Arithmetic Operators

Example

```
console.log(11 % 3);    // 2
console.log(11 % -3);   // 2
console.log(-11 % 3);   // -2

console.log(1.5 / 0.0);  // Infinity
console.log(-1.5 / 0.0); // -Infinity
console.log(0.0 / 0.0);  // NaN

const x = 0;
console.log(5 / x);
```

3. Logical Operators

Logical Operators

- Logical operators take **Boolean operands** and return **Boolean result**
- **Operator !** turns **true to false** and **false to true**
- Behavior of the **operators &&, || and ^** (1 == true, 0 == false):

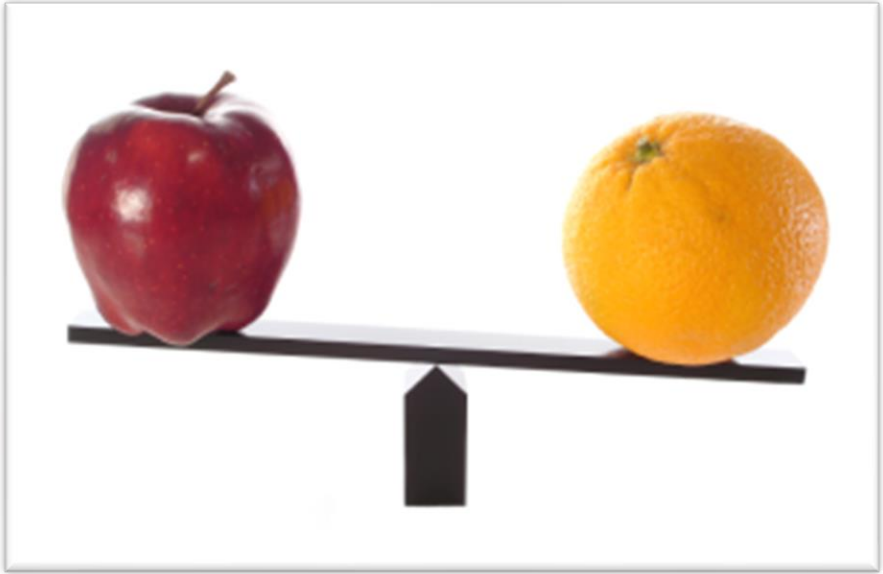
Operation	**||**				**&&**				**^**			
Operand 1	0	0	1	1	0	0	1	1	0	0	1	1
Operand 2	0	1	0	1	0	1	0	1	0	1	0	1
Result	0	1	1	1	0	0	0	1	0	1	1	0

Logical Operators

Example

Using the logical operators:

```
let a = true;
let b = false;

console.log(a && b); // False
console.log(a || b); // True
console.log(a ^ b); // True
console.log(!b); // True
console.log(b || true); // True
console.log(b && true); // False
console.log(a || true); // True
console.log(a && true); // True
console.log(!a); // False
console.log((5 > 7) ^ (a == b)); // False
```



4. Comparison and Assignment Operators

Comparison Operators

- Comparison operators are used to compare variables

  ==, <, >, >=, <=, !=, ===,!==

- For equality comparison, the use of === and !== is preferred

```
let a = 5;
let b = 4;

console.log(a >= b);    // True
console.log(a != b);    // True
console.log(a == b);    // False
console.log(0 == '');   // True
console.log(0 === ''); // False
```

Assignment Operators

Assignment operators are used to assign a value to a variable

=, +=, -=, *=, /=, ...

```
let x = 6;
let y = 4;

console.log(y *= 2); // 8

let z = y = 3;       // y=3 and z=3

console.log(z);      // 3
console.log(x |= 1); // 7
console.log(x += 3); // 10
console.log(x /= 2); // 5
```

5. Other Operators

Other Operators

- String concatenation **operator +** is used to **concatenate strings**
- If the second operand is not a string, it is converted to string automatically

```
let first = "First";
let second = "Second";

console.log(first + second); // FirstSecond

let output = "The number is : ";
let number = 5;

console.log(output + number); // The number is : 5
```

Other Operators

- Member access **operator .** is used to **access object members**
- Square brackets **[]** are used as indexers, to **access a member** with a certain name
- Parentheses **( )** are used to **override the default operator precedence** or to **invoke functions**
- Conditional **operator ?:** has the form
  - if **b** is **true** then the result is **x** else the result is **y**
  - b ? x : y
- The **new operator** is used to **create** new **objects**
- The **typeof operator** returns the **type of the value**

Other Operators

Example

Using some other operators:

```
let a = 6;
let b = 4;

console.log(a > b ? 'a > b' : 'b >= a'); // a > b

let c = b = 3; // b = 3; followed by c = 3;

console.log(c); // 3
console.log(new Number(6) instanceof Number); // true
console.log(6 instanceof Number); // false
console.log((a + b) / 2); // 4
console.log(typeof c); // number
console.log(void(3 + 4)); // undefined
```

6. Operators Precedence

Operators Precedence

- When in doubt, take a look at the MDN Precedence chart
- **Parenthesis operator** always has **highest** precedence
- It’s considered a good practice to use parentheses, even when it’s not necessary

  Improves code readability

Operators Precedence

The following table is ordered from highest (21) to lowest (1) precedence.

Precedence	Operator type	Associativity	Individual operators
21	Grouping	n/a	`( … )`
20	Member Access	left-to-right	`… . …`
	Computed Member Access	left-to-right	`… [ … ]`
	`new` (with argument list)	n/a	`new … ( … )`
	Function Call	left-to-right	`… ( … )`
	Optional chaining	left-to-right	`?.`
19	`new` (without argument list)	right-to-left	`new …`
18	Postfix Increment	n/a	`… ++`
	Postfix Decrement		`… --`

7. Expressions

Expressions

Expressions are sequences of operators, literals and variables that are evaluated to some value

```
let r = (150 - 20) / 2 + 5; // r = 70

// Expression for calculation of circle area
let surface = Math.PI * r * r;

// Expression for calculation of circle perimeter
let perimeter = 2 * Math.PI * r;
```

Expressions

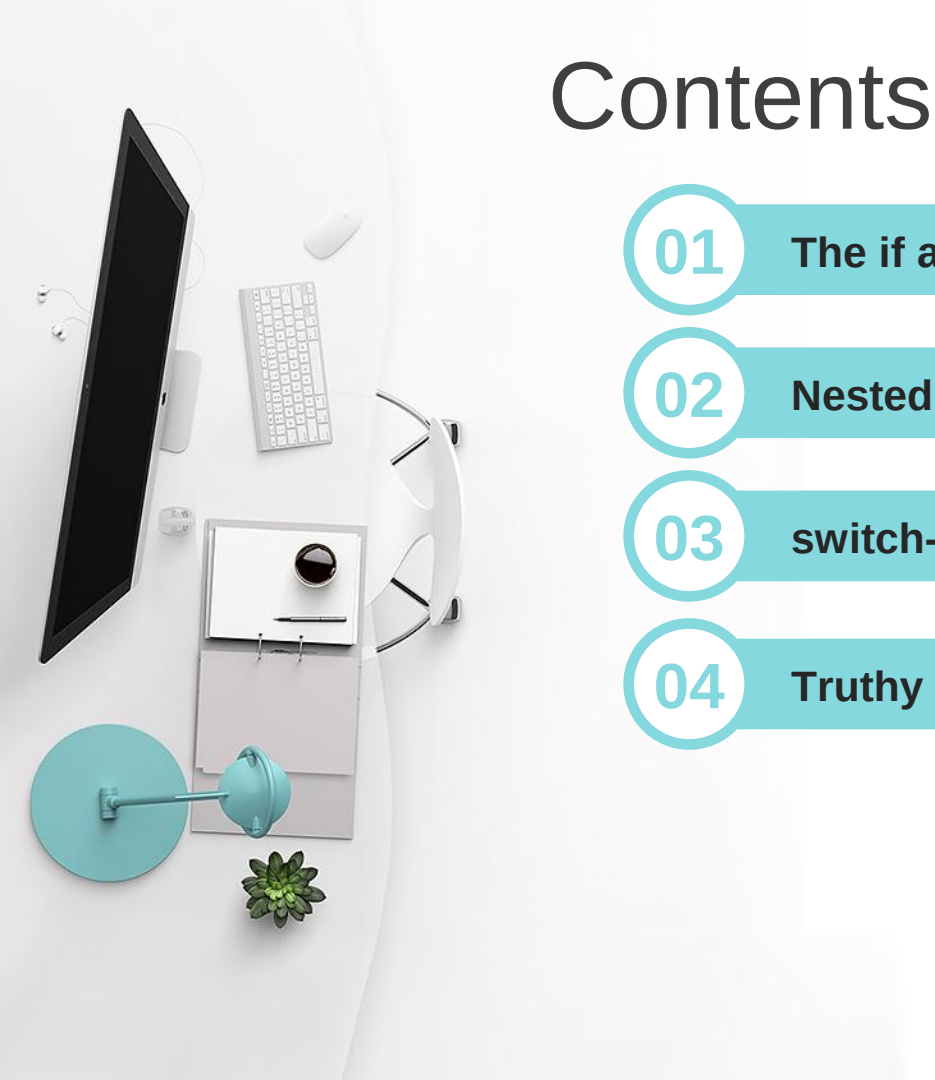
Expressions have:

- **Type** (integer, real, Boolean,...)
- **Value**

```
let a = 2 + 3; // a = 5

let b = (a + 3) * (a - 4) + (2 * a + 7) / 4;  // b = 12

let greater = (a > b) || ((a == 0) && (b == 0));
```

Conditional Statements

Implementing Control Logic in JavaScript



Contents

1. if and if-else

Implementing Conditional Logic

The if Statement

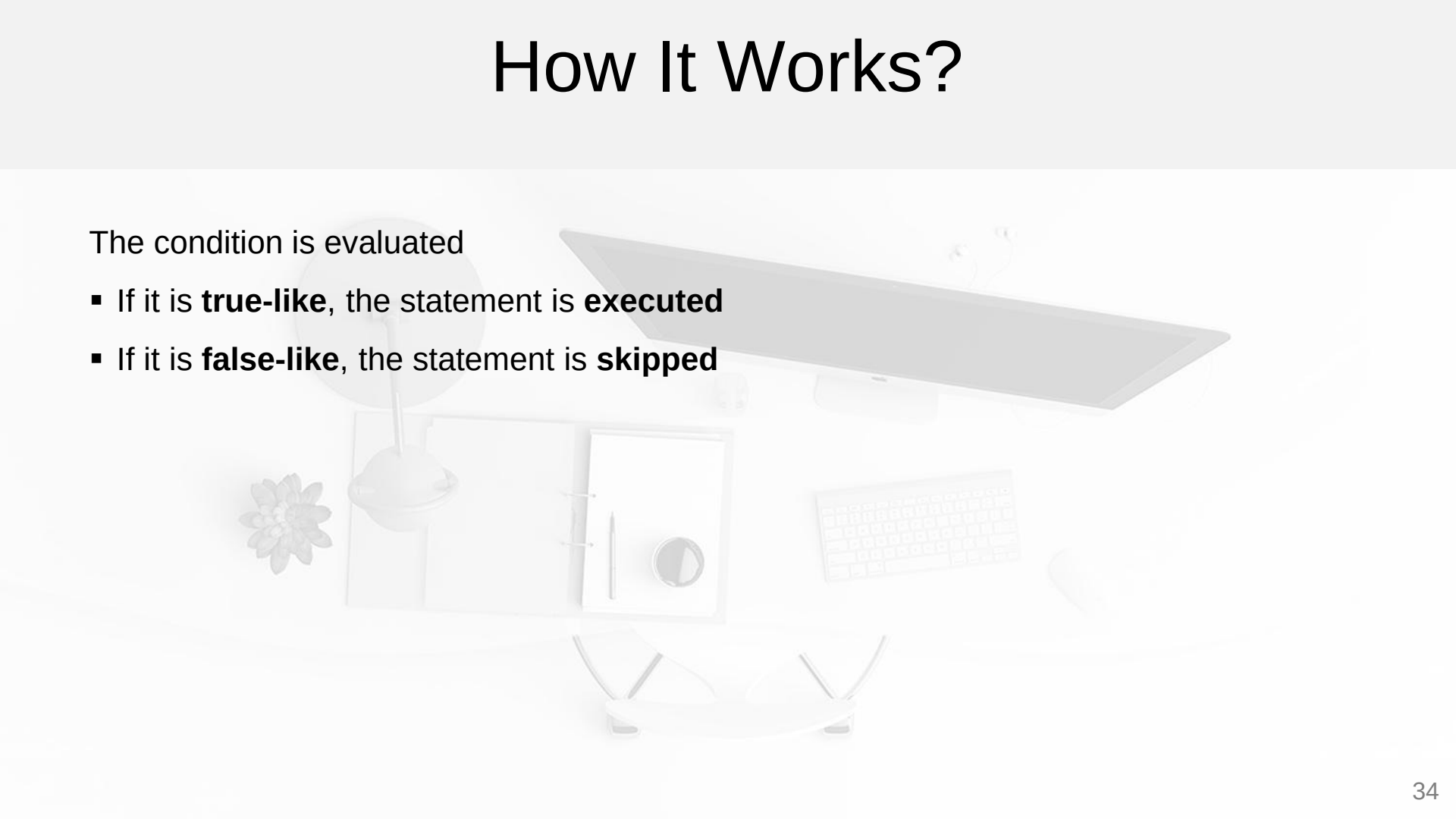
- The simplest conditional statement
- Enables you to test for a condition
- Branch to different parts of the code depending on the result
- The simplest form of an **if** statement:

```
if (condition) {
        statements;
}
```

Condition and Statement

- The condition can be:
  - Boolean variable
  - Boolean logical expression
  - Comparison expression
  - Integer, object, function… anything!
- The condition can be of any type
- The statement can be:
  - **Single** statement ending with a semicolon
  - **Block** enclosed in braces

How It Works?



The condition is evaluated

- If it is **true-like**, the statement is **executed**
- If it is **false-like**, the statement is **skipped**

The if Statement

Example

- Examples with **if** statements

```
var bigger = 123;
var smaller = 24;
if (smaller > bigger) {
    bigger = smaller;
}
console.log('The greater number is: ' + bigger);
```

- The expression evaluates for **true-like** or **false-like** values

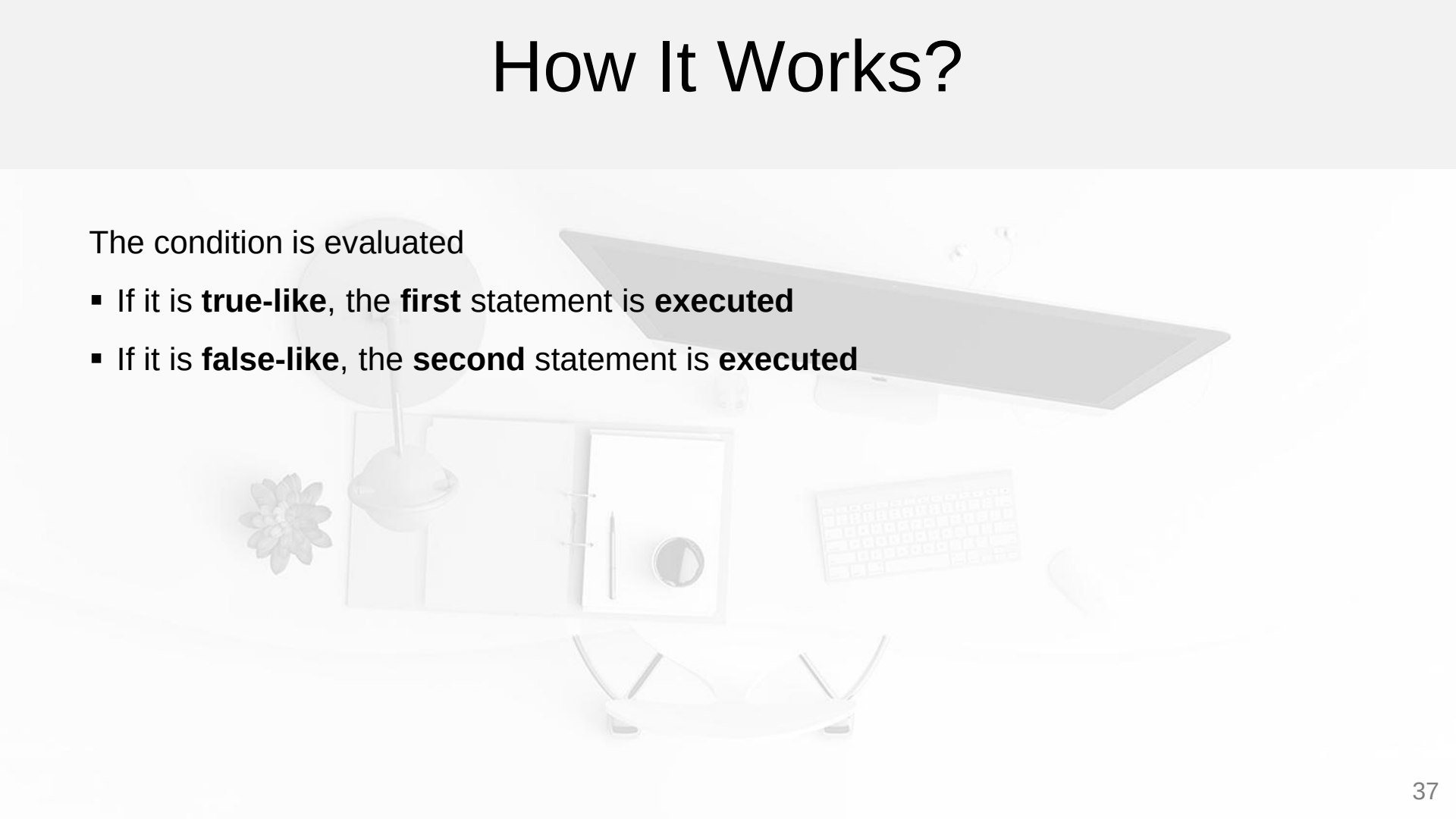
```
var str = '1c23';
if(!(+str)){ // if str is not a number, +str is NaN
    throw new Error('str is not a Number!');
}
```

The if-else Statement

- More complex and useful conditional statement
- Executes one branch if the condition is true, and another if it is false
- The simplest form of an **if-else** statement:

```
if (expression) {
        statement1;
} else {
        statement2;
}
```

How It Works?



The condition is evaluated

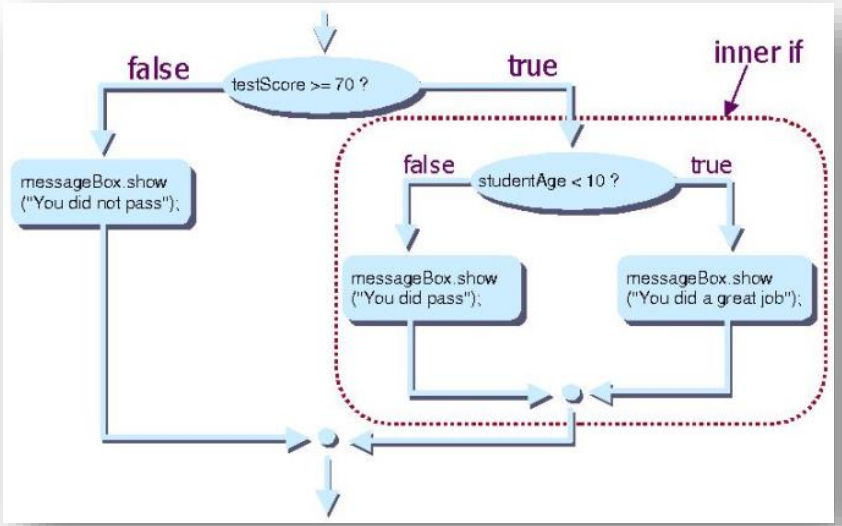
- If it is **true-like**, the **first** statement is **executed**
- If it is **false-like**, the **second** statement is **executed**

if-else Statement

Example

Checking a number if it is odd or even

```
var s = '123';
var number = +s;
if (number % 2) {
    console.log('This number is odd.');
} else {
    console.log('This number is even.');
}

if (+str) {
    console.log('The string is a Number');
} else {
    console.log('The string is not a Number');
}
```



2. Nested if Statements

Creating More Complex Logic

Nested if Statements

- if and if-else statements can be nested, i.e. used inside another if or else statement
- Every else corresponds to its closest preceding if

```
if (expression) {
        if (expression) {
                statement;
        } else {
                statement;
        }
} else {
        statement;
}
```

Nested if - Good Practices

- **Always use { … } blocks** to avoid ambiguity

  Even when a single statement follows

- **Avoid** using **more than three levels** of nested if statements

- Put the **case** you **normally expect** to process **first**, then write the unusual cases

- Arrange the code to make it more readable

Nested if Statements

Example

Examples with nested if statements

```
if (first === second) {
    console.log('These two numbers are equal.');
} else {
    if (first > second) {
        console.log('The first number is bigger.');
    } else {
        console.log('The second is bigger.');
    }
}
```

```
var n = +str;
if (n) {
    if (n % 2) {
        console.log('The number is odd');
    } else {
        console.log('The number is even');
    }
} else { //n is NaN
    console.log('This is not a number!');
}
```

Multiple if-else-if-else-…

Sometimes we need to use another if construction in the else block

Thus, **else if** can be used:

```
var ch = 'X';
if (ch === 'A' || ch === 'a') {
    console.log('Vowel [ei]');
} else if (ch === 'E' || ch === 'e') {
    console.log('Vowel [i:]');
} else if  …
else …
```

3. switch-case

Making Several Comparisons at Once

The switch-case Statement

Selects for execution a statement from a list **depending on the value** of the switch expression

```
switch (day) {
    case 1: console.log('Monday'); break;
    case 2: console.log('Tuesday'); break;
    case 3: console.log('Wednesday'); break;
    case 4: console.log('Thursday'); break;
    case 5: console.log('Friday'); break;
    case 6: console.log('Saturday'); break;
    case 7: console.log('Sunday'); break;
    default: console.log('Error!'); break;
}
```

How switch-case Works?

- The expression is evaluated
- When one of the constants specified in a case label is equal to the expression

  The statement that corresponds to that case is executed
- If no case is equal to the expression
  - If there is default case, it is executed
  - Otherwise, the control is transferred to the end point of the switch statement

The Fall-through Behavior in switch

JavaScript supports the fall-through behavior

- i.e. if a case statement **misses a break**, the code for the next cases is also executed
- Until a break is found

```
switch (day) {
    case 1:
        /* 2, 3 and 4 */
    case 5:
        console.log('Working day'); break;
    case 6:
    case 7:
        console.log('Weekend!'); break;
    default:
        console.log('Error!'); break;
}
```

4. Truthy and Falsy Values

First steps in the dynamic beauty of JavaScript

true-like and false-like values

- JavaScript, as a weakly typed language, can use **every value as true or false**
- Every value can be converted to its Boolean representation **using double not - !!**

```
console.log(
    !!'', // empty string is false-like
    !!'0', // non-empty strings are true-like
    !!0, // zero is false-like
    !!35, // non-zero numbers are true-like
    !![], // objects are true-like
    !!NaN, // NaN is false-like
    !!'true', // true
    !!'false' // true,
    !!null, // both null and undefined are false-like
    !!undefined
);
```

true-like and false-like values

Undefined	**false**
Null	**false**
Boolean	The result equals the input argument (no conversion).
Number	The result is **false** if the argument is **+0**, **−0**, or **NaN**; otherwise the result is **true**.
String	The result is **false** if the argument is the empty String (its length is zero); otherwise the result is **true**.
Object	**true**.

Truthy and Falsy Values

- Every type in JavaScript has an inherent Boolean value

  So called truthy (TRUE-like) and falsy (FALSE-like) values

- These values are **falsy**

  false, 0, "" / '', null, undefined, NaN

- All other values are **truthy**

  Info: http://www.sitepoint.com/javascript-truthy-falsy/

Summary

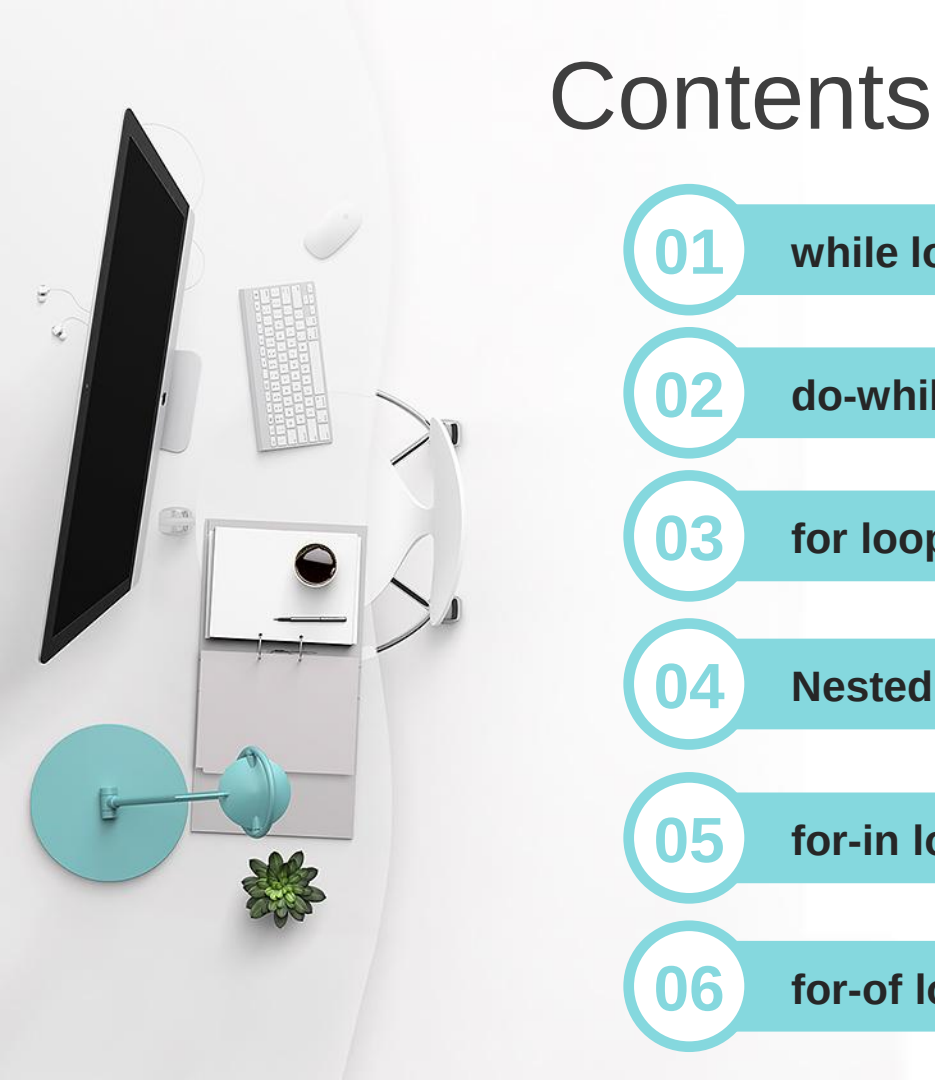
- **Comparison** and **logical operators** are used to compose **logical conditions**

- The conditional statements **if** and **if-else** provide **conditional execution** of blocks of code
  - Constantly used in computer programming
  - Conditional statements can be nested

- The **switch** statement easily and elegantly **checks an expression for a sequence of values**
  - Supports the fall-through behavior
  - Can contain expressions in the case value

Loops

Execute Blocks of Code Multiple Times



Contents

What is a loop?

- A loop is a **control statement** that allows **repeating the execution of a block of statements**
  - May execute a code block **fixed number of times**
  - May execute a code block **while given condition holds**
  - May execute a code block for **each member of a collection**

- Loops that **never end** are called **infinite loops**

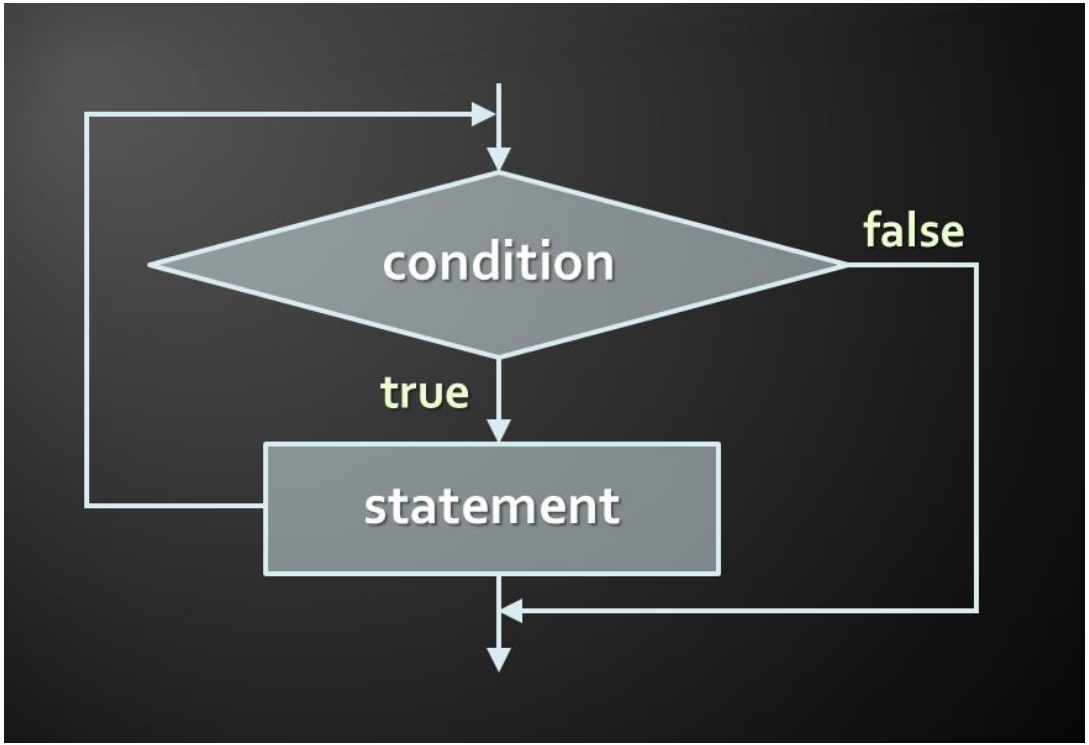
1. while loop

How to use a while loop?

- The simplest and most frequently used loop
- Has a **repeat condition**
  - Also called **loop condition**
  - Is not necessary strictly a **Boolean** value
  - Is evaluated to **true** or **false**
    - ✓5, 'non-empty', {}, etc. are evaluated as true
    - ✓0, '', null, undefined are evaluated as false

```
while (condition) {
    statements;
}
```

while loop – How It Works?



while loop

Example

```
let counter = 0;
while (counter < 10) {
    console.log('Number : ' + counter);
    counter += 1;
}
```

Exercises

1. Sum 1..N

   Calculate and print the sum of the first N natural numbers

2. Prime Number

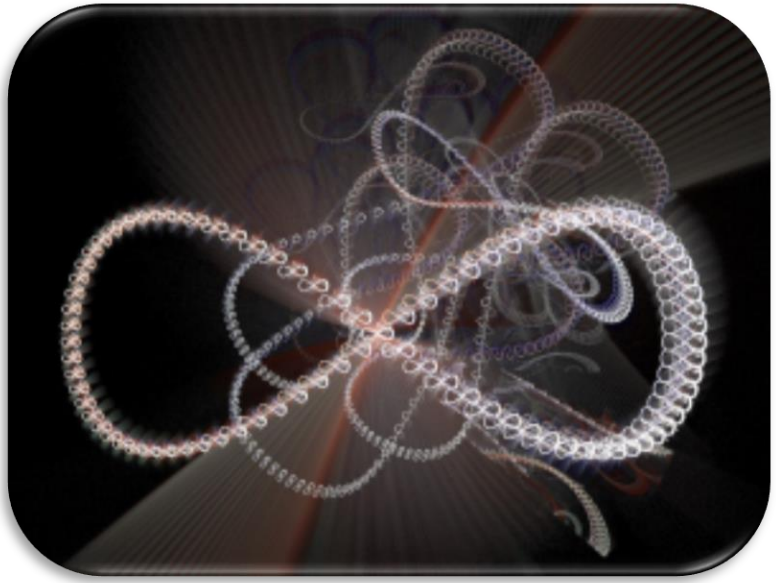
   Checking whether a number is prime or not

3. Calculating Factorial

Using break Operator

break operator exits the inner-most loop

```
let n = 10,
    fact = 1,
    factStr = 'n! = ';

while (1) { //infinite loop
    if (n === 1) {
        break;
    }

    factStr += n + '*'
    fact *= n;
    n -= 1;
}

factStr += '1 = ' + fact;
console.log(factStr);
```



2. do-while loop

Using do-while loop

Another loop structure is:

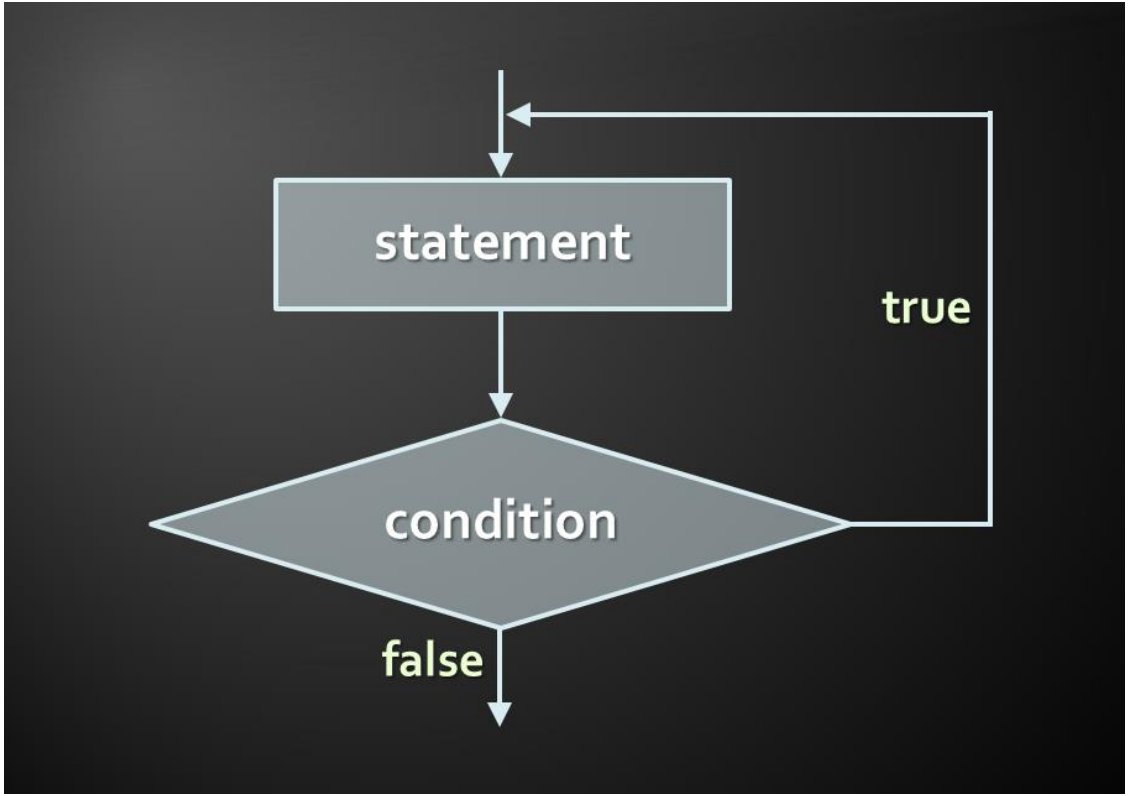
- The block of statements is repeated

  While the Boolean loop condition holds

- The loop is always **executed at least once**

```
do {
    statements;
} while (condition);
```

do-while statement



do-while

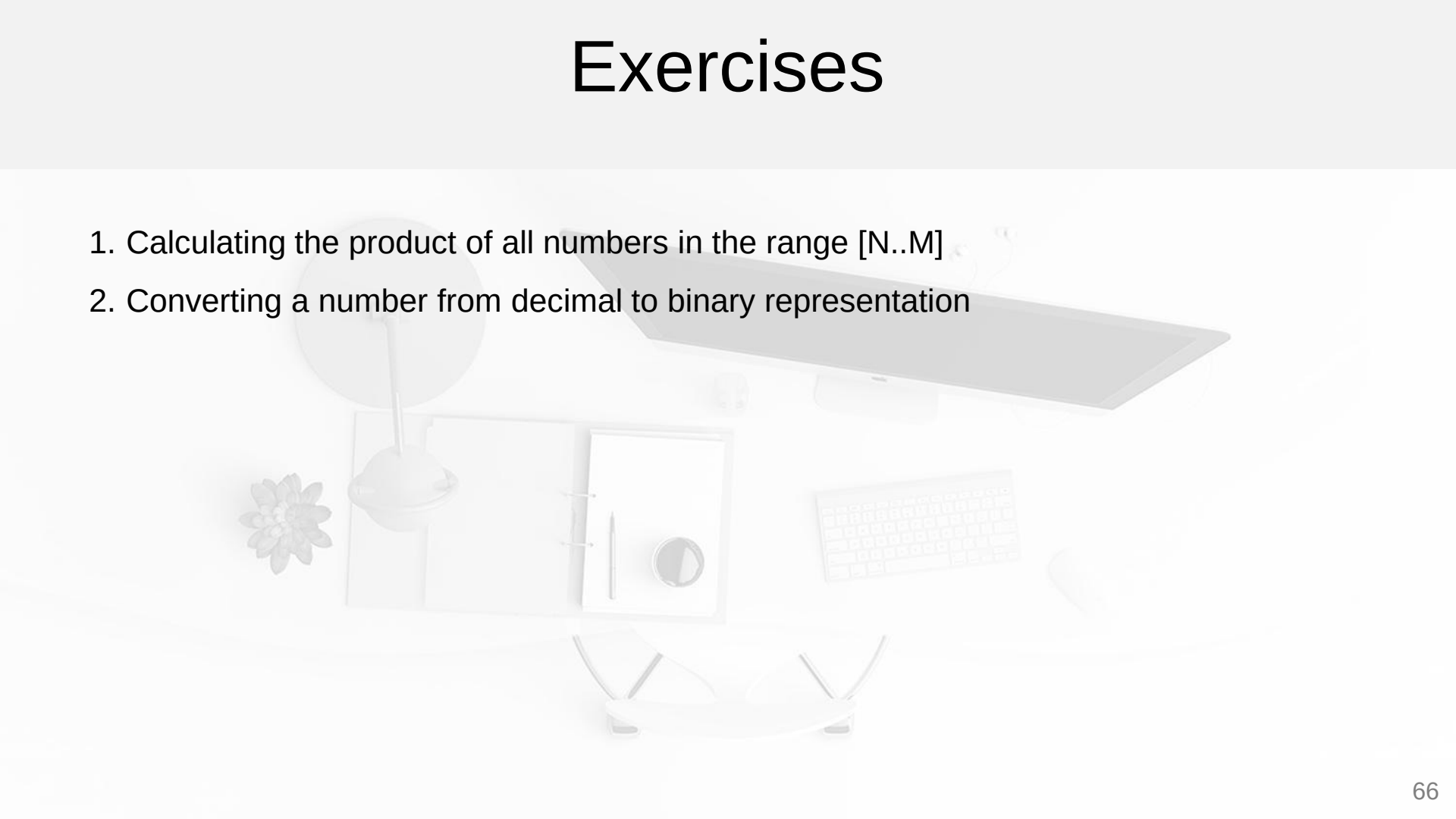
Example

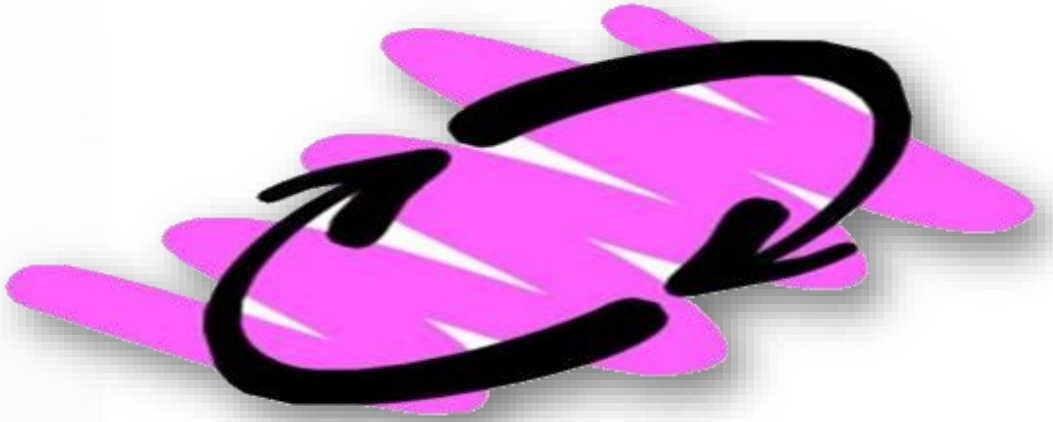
Calculating N!

```
let fact = 1,
    factStr = 'n! = ';

do {
    fact *= n;
    factStr += n + '*'
    n -= 1;
} while (n);

factStr += ' = ' + fact;
console.log(factStr)
```

Exercises

1. Calculating the product of all numbers in the range [N..M]
2. Converting a number from decimal to binary representation





3. for loops

for loops

- The typical for loop syntax is:

```
for (initialization; test; update) {
    statements;
}
```

- Consists of
  - Initialization statement
  - Test expression that is evaluated to Boolean
  - Update statement
  - Loop body block

The Initialization Expression

```
for (let number = 0; number < 10; number += 1) {
    // Can use number here
}

// Cannot use number here
```

- Executed once, just before the loop is entered
- Usually used to declare a counter variable

  Multiple variables can be declared in the initialization statement

The Test Expression

```
for (let number = 0; number < 10; number += 1) {
    // Can use number here
}

// Cannot use number here
```

- Evaluated before each iteration of the loop
  - If evaluated **true**, the loop body is executed
  - If evaluated **false**, the loop ends
- Used as a **loop condition**

The Update Expression

```
for (let number = 0; number < 10; number += 1) {
    // Can use number here
}

// Cannot use number here
```

- Executed at each iteration **after** the body of the loop is finished
- Usually used to update the counter

  for loops support multiple update statements, separated by the , (comma) operator

Simple for loop

Example

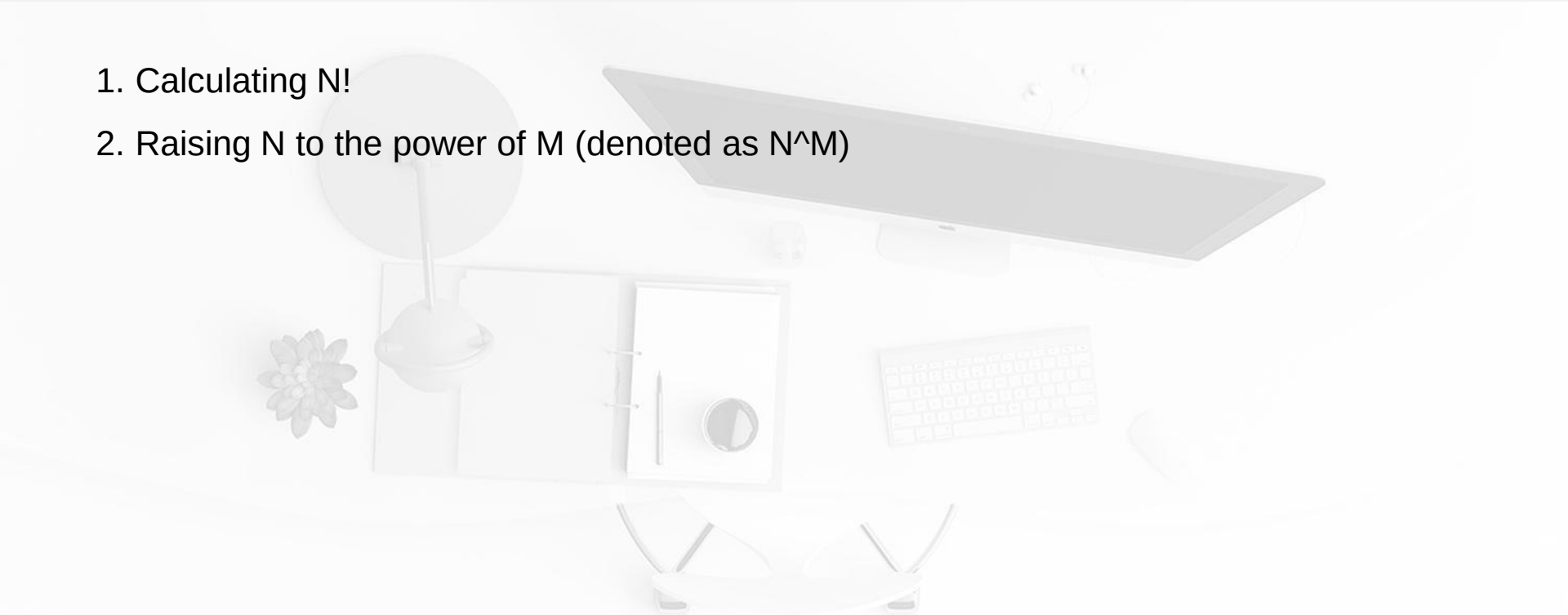
Print all natural numbers up to N

A simple for-loop to print the numbers [0..9]

```
const N = 10;

for (let number = 0; number < N; number += 1) {
    console.log(number + ' ');
}
```

Exercises

1. Calculating N!
2. Raising N to the power of M (denoted as N^M)



Complex for Loop

Example

Complex for loops could have **several counter variables**:

```
for (let i = 1, sum = 1, N = 128; i <= N; i *= 2, sum += i) {
    console.log('i=' + i + ', sum=' +sum);
}
```

4. Nested loops

What Is Nested Loop?

- A composition of loops is called a nested loop

  A loop inside another loop

- Example:

```
for (let i = 0; i < 10; i += 2) {
    for (let j = 0; j < 20; j += 1) {
        while(i !== j) {
            console.log(i);
            j += 1;
        }
    }
}
```

Nested Loops

Example

Print the following triangle:

```
1
1 2
...
1 2 3 ... N
```

```
const N = 7;
let result = '';

for(let row = 1; row <= N; row += 1) {
    for(let column = 1; column <= row; column += 1) {
        result += column + ' ';
    }

    result += '\n';
}

console.log(result);
```

Exercises

- Print all prime numbers in the range [N..M]
- Happy numbers

  Print all four-digit numbers in format ABCD such that A+B = C+D
- TOTO 6/49

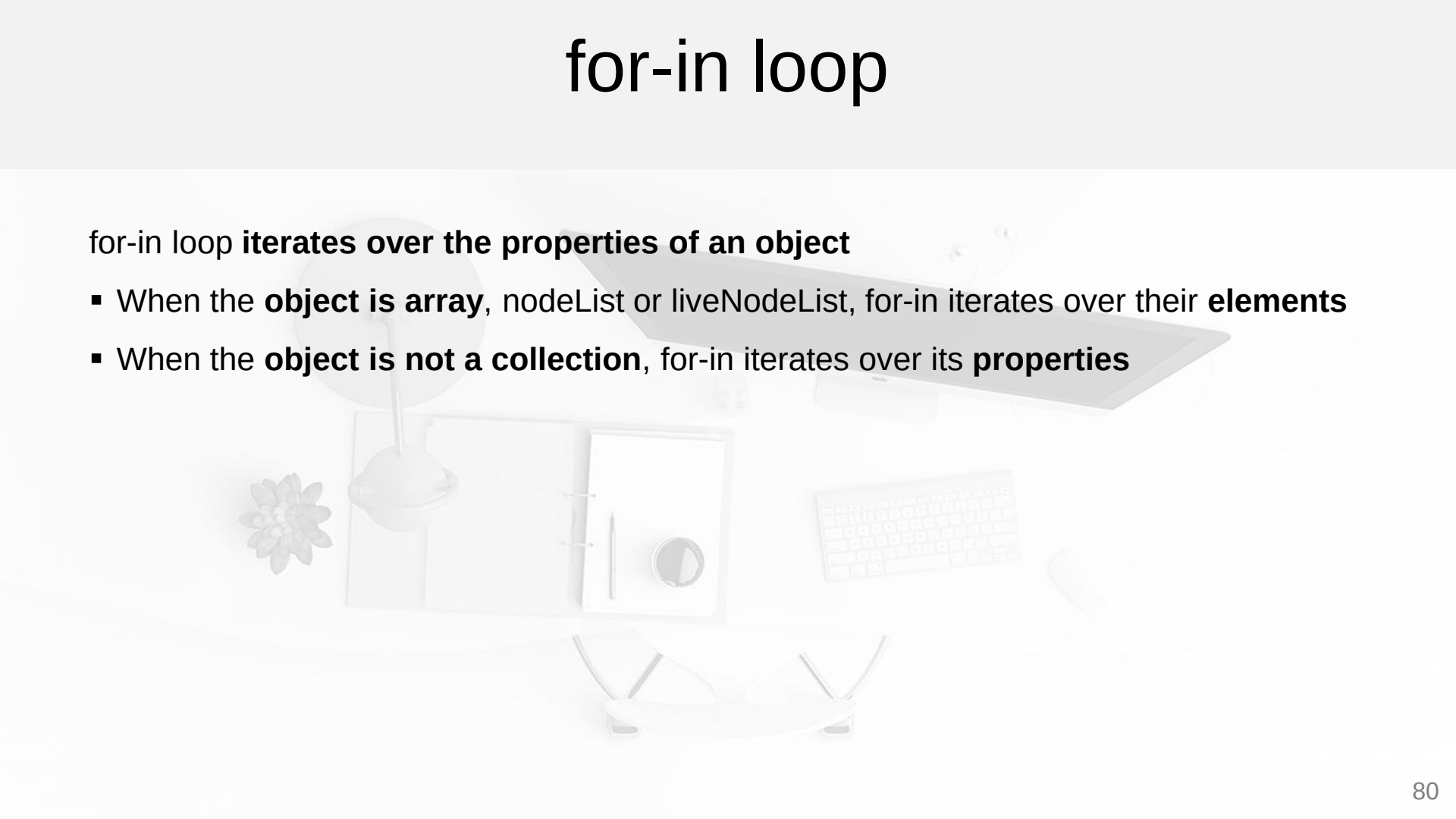
  Print all 6/49 combinations

5. for-in loop

for-in loop

for-in loop **iterates over the properties of an object**

- When the **object is array**, nodeList or liveNodeList, for-in iterates over their **elements**
- When the **object is not a collection**, for-in iterates over its **properties**



for-in

Example

- Iterating over the **properties** of **document**

```
// propName is a string - the name of the property
for (const propName in document) {
    console.log(document[propName]);
}
```

- Iterating over the **elements** of an **array**

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

for (const index in arr) {
    console.log(arr[index]);
}
```

6. for-of loop

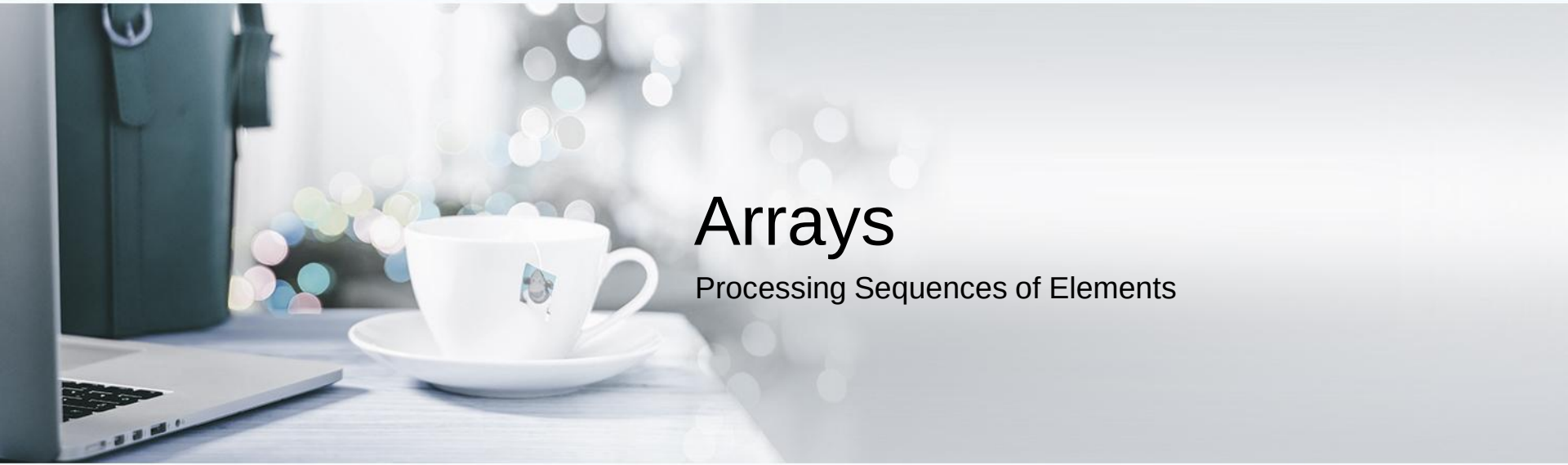
for-of loop

- for-of loop **iterates over the elements in an array**
  - Can be used only on arrays, or array-like objects
  - i.e. the arguments object

```
const arr = ['One', 'Two', 'Three', 'Four'];

for(const n of arr) {
  console.log(n);
}
```

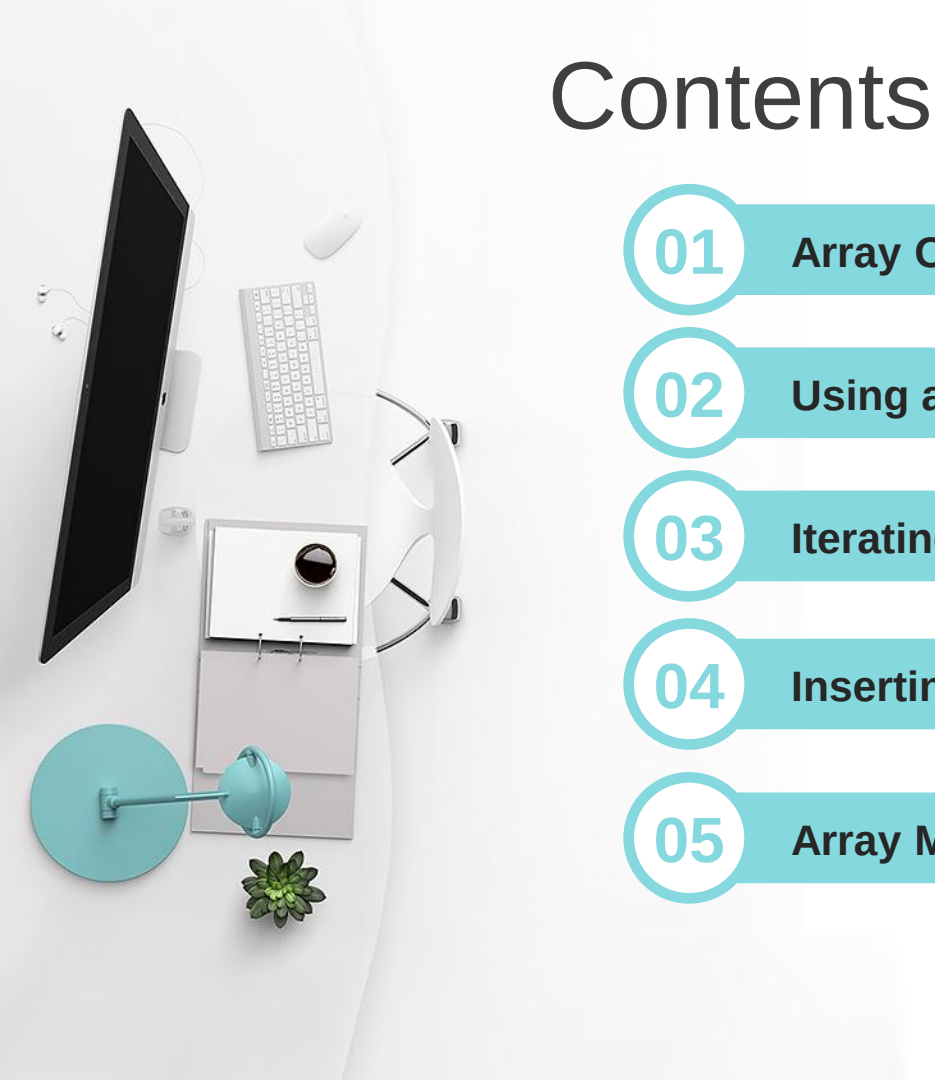
- The for-of loop is **part of the ECMAScript 6** standard

  Supported in all modern browsers



Arrays

Processing Sequences of Elements



Contents

1. Array Overview

What are arrays? How to use arrays?

What are Arrays?

An array is **a sequence of elements**

- The **order** of the elements is **fixed**
- Does **not** have **fixed size**

  Can get the current length (Array**.length**)

Declaring and Initializing Arrays

Initializing an array in JavaScript can be done in **three ways**:

- Using **new Array(elements)**:

```
var arr = new Array(1, 2, 3, 4, 5);
```

- Using **new Array(initialLength)**:

```
var arr = new Array(10);
```

- Using **array literal** (recommended):

```
var arr = [1, 2, 3, 4, 5];
```

Declaring Arrays

Declaring an array in JavaScript

```
// Array holding integers
var numbers = [1, 2, 3, 4, 5];

// Array holding strings
var weekDays = ['Monday', 'Tuesday', 'Wednesday',
  'Thursday', 'Friday', 'Saturday', 'Sunday']

// Array of different types
var mixedArr = [1, new Date(), 'hello'];

// Array of arrays (matrix)
var matrix = [
  ['0,0', '0,1', '0,2'],
  ['1,0', '1,1', '1,2'],
  ['2,0', '2,1', '2,2']];
```

2. Using arrays

Read and Modify Elements by Index

How to Access Array Element?

- Array elements are accessed using the **indexer operator**: **[]** (square brackets)
  - Array indexer takes element's index as parameter in the range **[0; length-1]**
  - The first element has index **0**
  - The last element has index **length-1**

- Array elements can be **retrieved** and **changed** by the **[]** (indexer) **operator**

Reversing an Array

Example

Reversing the elements of an array

```
//always declare var variables on the top of the scope!
var array,
    len,
    reversed,
    i,
    j;

array = [1, 2, 3, 4, 5];
reversed = [];

for (i = 0, len = array.length; i < len; i += 1) {
    j = len - i - 1;
    reversed.push(array[j]);
}
```

3. Iterating Arrays

Iterating Arrays with for

- Use **for loop** to process an array when you **need to keep track of the index**
- In the loop body use the element at the loop index (**array[index]**):

```
var i, len;
for (i = 0, len = array.length; i < len; i += 1) {
    squares[i] = array[i] * array[i];
}
```

Iterating Arrays with for

- **Example 1:**

  Printing array of numbers in reversed order

```
var arr, i, len;
arr = [1, 2, 3, 4, 5];
for (len = arr.length, i = len - 1; i >= 0; i -= 1) {
    console.log(arr[i]);
}
// Result: 5 4 3 2 1
```

- **Example 2:**

  Initialize all array elements with their corresponding index number

```
var i, len
for (i = 0, len = array.length; i < len; i += 1) {
    array[i] = i;
}
```

Iterating Arrays using for-in

- How **for-in** loop works?

  index iterates through the indexes of the array

- Used when the **indexes** are **unknown**
  - All elements are accessed one by one
  - Order is not guaranteed
  - Works for objects as well

```
var index;
for (index in array) {
  // great code
}
```

Iterating Arrays with for-in

Print all elements of an array of strings:

```
var capitals, i;
capitals = [
    'Sofia',
    'Washington',
    'London',
    'Paris'
];

for (i in capitals) {
    console.log(capitals[i]);
}
```

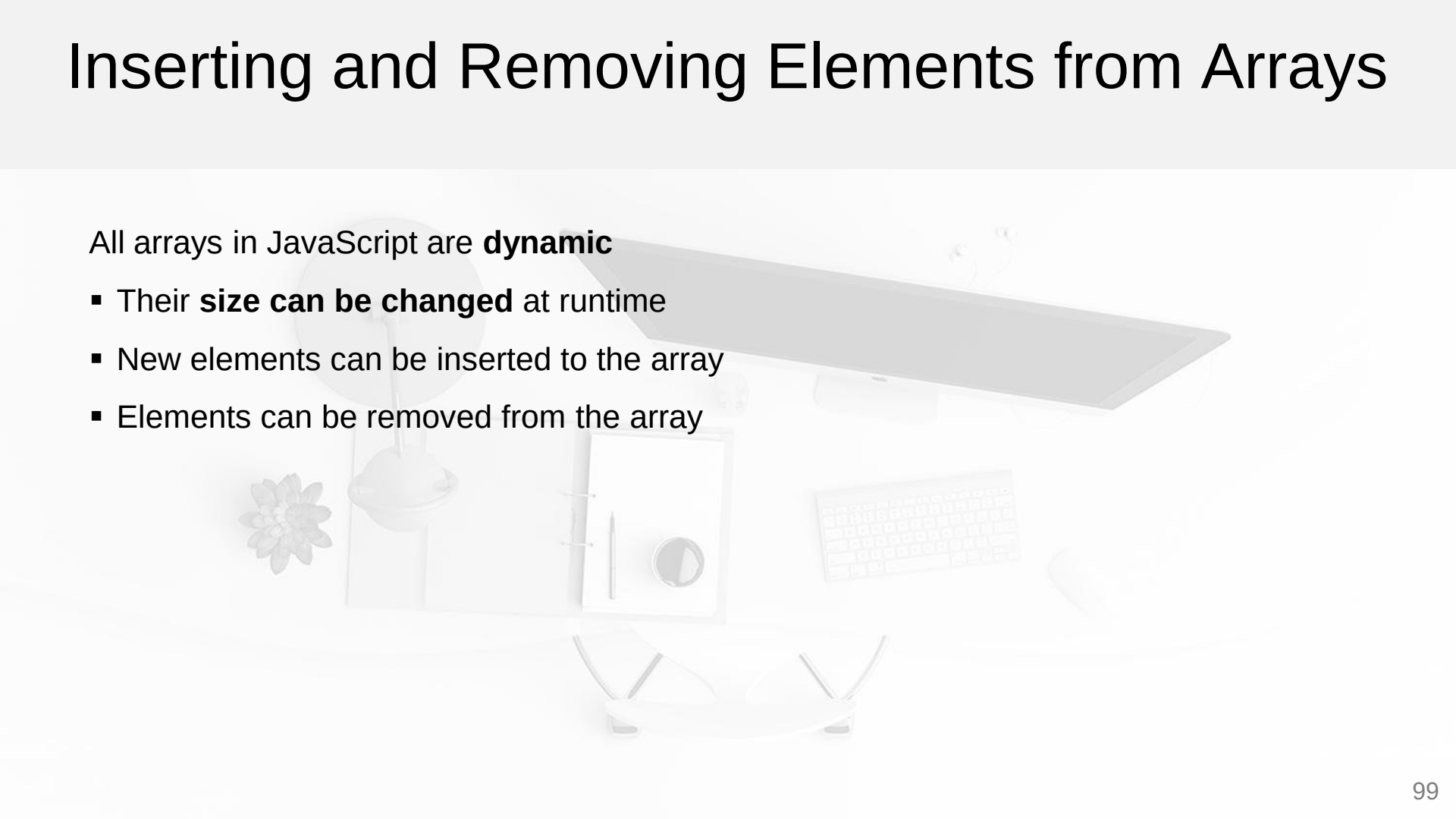
4. Inserting and Removing Elements from Arrays

push, pop, shift, unshift

Inserting and Removing Elements from Arrays

All arrays in JavaScript are **dynamic**

- Their **size can be changed** at runtime
- New elements can be inserted to the array
- Elements can be removed from the array



Inserting and Removing Elements from Arrays

Methods for array manipulation:

- **Array.push(element1, [, ...[, elementN]])**
  - **Inserts** the new element(s) at the **tail** of the array
  - **Return** the **new length** property

- **Array.pop()**
  - **Removes** the element at the **tail**
  - **Returns** the **removed element**

Inserting and Removing Elements from Arrays

Methods for array manipulation:

- **Array.unshift(element1, [, ...[, elementN]])**
  - **Inserts** the new element(s) at the **head** of the array
  - **Return** the **new length** property

- **Array.shift()**
  - **Removes** the element at the **head**
  - **Returns** the **remove element**

5. Array Methods

Reversing, joining, etc.

Array Methods

- **Array.reverse()**
  - **Reverses** the elements of the array
  - **Returns** a new arrays

```
var items = [1, 2, 3, 4, 5, 6];
var reversed = items.reverse();
//reversed = [6, 5, 4, 3, 2, 1]
```

- **Array.join(**separator**)**
  - **Concatenates** the elements with a separator
  - **Returns** a string

```
var names = ["John", "Jane", "George", "Helen"];
var namesString = names.join(", ");
//namesString = "John, Jane, George, Helen"
```

Concatenating Arrays

- **arr1.concat(**arr2**)**
  - **Inserts** the elements of **arr2** at the **end** of **arr1**
  - **Returns** a new array
  - **arr1** and **arr2** remain **unchanged**!

```
var arr1 = [1, 2, 3];
var arr2 = ["one", "two", "three"];
var result = arr1.concat(arr2);
//result = [1, 2, 3, "one", "two", "three"]
```

- **Adding the elements of an array to another array**

```
var arr1 = [1, 2, 3];
var arr2 = ["one", "two", "three"];
[].push.apply(arr1, arr2);
//arr1 = [1, 2, 3, "one", "two", "three"]
```

Getting Parts of Arrays

Array.slice(fromIndex [, toIndex]**)**

- **Returns** a new array

  A **shallow copy** of a portion of the array

- The new array contains the elements from indices **fromIndex** to **to** (excluding **toIndex**)
- Can be used to clone an array

```
var items = [1, 2, 3, 4, 5];
var part = items.slice(1, 3);
//part = [2, 3]
var clonedItems = items.slice();
```

Splicing Arrays

- **Array.splice(**index[, count[, elements]]**)**
  - **Removes count** elements, starting from **index** position
  - **Adds elements** at position **index**
  - **Returns** a new array, containing the removed elements

```
var numbers = [1, 2, 3, 4, 5, 6, 7];
var result = numbers.splice(3, 2, "four",
"five", "five.five");
```

Splicing Arrays (cont.)

- Example uses:
  - **Remove** elements from any index of the array:

```
//removes a single element at position index
items.splice(index, 1);
//removes count elements starting from position index
items.splice(index, count);
```

  - **Insert** elements at any index of the array:

```
//Inserts a single element at position index
items.splice(index, 0, element);
//Inserts many elements starting from position index
items.splice(index, 0, item1, item2, item3);
```

Searching in Arrays

- **Array.indexOf(**searchElement[, fromIndex]**)**
  - **Returns** the **index** of the **first match** in the array
  - **Returns -1** is the element is **not found**

- **Array.lastIndexOf(**searchElement, [fromIndex]**)**
  - **Returns** the **index** of the **last match** in the array
  - **Returns -1** is the element is **not found**

- Array.indexOf() and Array.lastIndexOf() do not work in all browsers

Searching in Arrays

- **Array.includes(**search-item**)**
  - Check if an element is present in an array
  - **Return** true or false.
  - Example:

    const fruits = [“Banana”, “Orange”, “Apple”, “Mango”];

    fruits.includes(“Mango”);

Searching in Arrays

- **Array.find(**callbackFn**)**
  - Returns the value of the first array element that passes a test function
  - The function takes 3 arguments:
    - The item value
    - The item index (optional)
    - The array itself (optional)

```
function testFunction(value) {
    return value > 18;
}
const numbers = [4, 9, 16, 25, 29];

let found = numbers.find(testFunction);
// let found = numbers.find((item) => item > 18);

console.log(found);
```

Sort() method

- **Array.sort**([callbackFn]): Sorts the elements of an array in place.
- The default sort order is ascending, built upon converting the elements into strings.
- Syntax:

```
// Functionless
sort()

// Arrow function
sort((a, b) => { /* … */ } )

// Compare function
sort(compareFn)
```

CompareFn(a, b) return value	Sort order
> 0	Sort a after b
< 0	Sort a before b
=== 0	Keep original order or a and b

Sort() method

Example

- Sort numbers in ascending order:

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){ return a - b });
```

- Comparing string properties is a little more complex:

```
products.sort(function(a, b){
    let x = a.Name.toLowerCase();
    let y = b.Name.toLowerCase();
    if (x < y) {return -1;}
    if (x > y) {return 1;}
    return 0;
});
```

Array Iteration Methods

- Array forEach()
- Array map()
- Array filter()
- Array reduce()
- Array Spread (...)

Array forEach()

- The method **calls a function** (a callback function) **once for each array element**.
- The function takes 3 arguments:
  - The item value
  - The item index (optional)
  - The array itself (optional)

```
const numbers = [45, 4, 9, 16, 25];
numbers.forEach((value) => {
    console.log(value);
});
```

Array map()

- The method **creates a new array** by performing a function on each array element.
- It does not change the original array.
- The function takes 3 arguments:
  - The item value
  - The item index (optional)
  - The array itself (optional)

```
const numbers1 = [45, 4, 9, 16, 25];
const numbers2 = numbers1.map((value) => value * 2);
// const numbers2 = numbers1.map((value) => {return value * 2;} );

console.log("numbers1: ", numbers1);
console.log("numbers2: ", numbers2);
```

Array filter()

- The method **creates a new array** with array **elements that pass a test**.
- The function takes 3 arguments:
  - The item value
  - The item index (optional)
  - The array itself (optional)

```
const numbers = [45, 4, 9, 16, 25];
const over18 = numbers.filter((value) => value > 18);
console.log(over18);
```

Array reduce()

- It **runs a function on each array element to produce** (reduce it to**) a single value**.
- It does not reduce the original array.
- The function takes 4 arguments:
  - The total (the initial value / previously returned value)
  - The item value
  - The item index (optional)
  - The array itself (optional)

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce((total, value) => total + value);
console.log(sum);
```

Array Spread (…)

- The ... operator expands an iterable (like an array) into more elements.

```
const q1 = ["Jan", "Feb", "Mar"];
const q2 = ["Apr", "May", "Jun"];
const year = [...q1, ...q2];
console.log(year);
```

Other Arrays Functions

- Arrays official documentation:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

- Checking for array
  - **typeof**([1, 2, 3]) → **object**

    Not working
  - **Array.isArray**([1, 2, 3]) → **true**

    Supported on all modern browsers

Exercise

1. Exercise 1

- Create an array containing product titles.
- Show all products.
- Find and delete element(s) containing a given string.
- Add a new element to the array.

2. Exercise 2

- Create an array containing image urls.
- Using Carousel to show all images.

String Methods

Operations with strings

String Methods

- string.**length**

  Returns the number of characters in the string

- string.**concat**(string2)

  Returns a new string – the concatenation of the two strings

- string.**replace**(str1, str2)

  Replaces first occurrence of str1 with str2

String Methods

- string.**indexOf**(substring [,position])
  - Returns the **left-most** occurrence of **substring** in a string, that is **after position**

    Position is optional and has default value of **0**
  - If string doesn't contain substring, returns **-1**

```
let text = "Hello world, welcome to the universe.";
let result = text.indexOf("welcome");
```

- string.**lastIndexOf**(substring [,position])
  - Returns the **right-most** occurrence of **substring** in a string, that is **before position**

    Position is optional, default value is string.**length**
  - If string doesn't contain substring, returns **-1**

String Methods

- string.**toLowerCase()**

  Returns a new string representing the calling string converted to lower case

```
let text = "Hello World!";
let result = text.toLocaleLowerCase();
```

- string.**toUpperCase()**

  Returns a new string representing the calling string converted to upper case

String Methods

- string.**includes**(searchString [, position = 0])

  Returns **true** if a string contains a specified string by performing a case-sensitive search.

  Otherwise, returning **false**.

```
let text = "Hello world, welcome to the universe.";
let result = text.includes("world");
```

String Methods

- string.**localeCompare**(compareString)

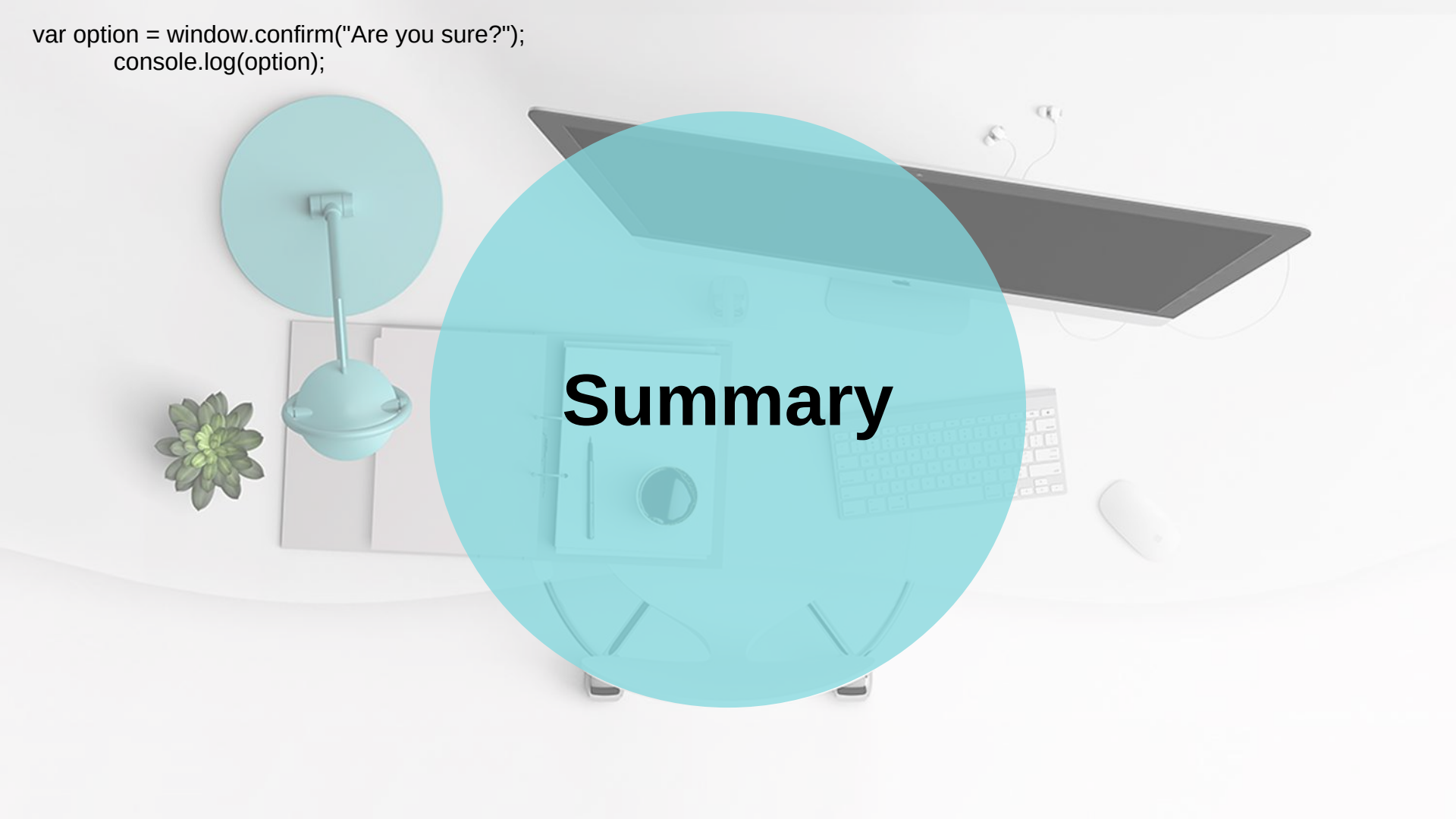
  Compares two strings in the current locale and returns sort order -1, 1, or 0 (for before, after, or equal).

  - -1 if the string is sorted before the compareString
  - 0 if the two strings are equal
  - 1 if the string is sorted after the compareString

  Note: The current locale is based on the language settings of the browser.

```
let text1 = "A";
let text2 = "a";
let result = text1.localeCompare(text2);
```

```
var option = window.confirm("Are you sure?");
        console.log(option);
```



Summary


Q&A