JAVASCRIPT

JavaScript

Numbers

In JavaScript, all numbers belong to the `number` type, and they can contain both integer and decimal values. Let's explore the operations we can perform with numbers, along with their precedence.

Operations

As you know, numbers can be added subtracted, multiplied, divided, raised to a power, and divided to get a remainder. Here are a few examples:

```
let addition = 10 + 5; // addition

console.log(addition); // 15


let subtraction = 10 - 5; // subtraction

console.log(subtraction); // 5


let multiplication = 10 * 5; // multiplication

console.log(multiplication); // 50


let division = 10 / 5; // division

console.log(division); // 2


let exponentiation = 10 ** 5; // exponentiation

console.log(exponentiation); // 100000


let remainder = 16 % 7; // division remainder

console.log(remainder); // 2 - subtract 7 until the result is <7
```

Using the `%` operator, you can get the last digit of a number:

```
let number = 345 % 10;
```

```
console.log(number); // 5
```

Or check if a number is even:

```
let number = 345 % 2;

console.log(number); // remainder 1 — an odd number
```

Precedence

Operations are usually executed in sequence from left to right, with the exception of multiplication and division. They have a **higher precedence,** so they are performed before addition and subtraction:

```
console.log(

  5 + 1 * 10 // 15, not 60

);
```

To modify this order of action, use `()`. Whatever expression you'll enclose in parenthesis, **it will take precedence** and be executed first. Here's an example:

```
console.log(

  (5 + 1) * 10 // 60

);
```

If there's more than one `()`, expressions are executed from left to right.

Strings

A string (`string`) is a sequence of characters: letters, numbers, special symbols, etc., used to represent textual data in programs. They must be **enclosed in quotes,** 'single` or "double", with identical quote types on **both ends** of the string.

String Length

You can find out the length of a string (the number of characters in it) using the length property:

```
const name = 'John';


console.log(name.length); // 4
```

Character in a String

You can access an individual character in a string by its index, but remember that **string indices start from `0`** (not `1`!). Here's an example:

```
const name = 'John';


console.log(
  name[0], // 'J'
  name[1], // 'o'
  name[2], // 'h'
  // the index of the last character is 1 less than the string length
  name[name.length - 1], // 'n'
);
```

Also, in square brackets, you can write an expression and calculate the index. For example, the index of the last character is 1 less than the string's length:

```
const name = 'John';
const lastIndex = name.length - 1;


console.log(
  name[lastIndex], // 'n'
  name[name.length - 1], // 'n' - calculation directly in []
);
```

Concatenation

The + lets us do more then just add numbers. We can use it to concatenate (glue together) **strings with strings:**

```
const name = 'John';
const lastName = 'Stewart';


const initials = name[0] + lastName[0];
console.log(initials);// 'JS'


const fullName = 'Mr.' + name + ' ' + lastName;
console.log(fullName);// 'Mr. John Stewart'
```

...and strings with numbers:

```
console.log(
  '12' + 34, // '1234'
  12 + '34', // '1234'
);
```

💡 Numbers encolsed in quotes are treated as a string.

Interpolation

Interpolation is the insertion of values into a string. We perform it with the `${...}` syntax:

```
const name = 'John';
const lastName = 'Stewart';

const fullName = `Mr. ${name} ${lastName}`;

console.log(fullName); // 'Mr. John Stewart'
```

Here's another example to show the difference between string concatenation and interpolation:

```
const name = 'John';
const age = 30;

// concatenation
const concatenationGreeting = 'Hello! My name is ' + name + ' and I am ' + age + '.';
console.log(concatenationGreeting); // 'Hello! My name is John and I am 30.'

// interploation
const interpolationGreeting = `Hello! My name is ${name} and I am ${age}.`;
console.log(interpolationGreeting); // 'Hello! My name is John and I am 30.'
```

In most cases, interpolation is more intuitive and therefore much more popular.

Boolean Type

The Boolean type can take one of only two values, `true` and `false`. We use it to state truths and falsehoods, respectively:

```
const isAdult = true;

const hasEnoughMoney = false;
```

Number Comparison

Boolean values can also be obtained as a result of comparisons, e.g., of numbers. JavaScript lets us compare them with the operators sourced straight from math lessons:

- `>` — greater than
- `<` — less than
- `>=` — greater than or equal to
- `<=` — less than or equal to

...and two more, **not to be confused** with the assignment `=` operator:

- `===` — strict equality
- `!==` — strict inequality

Here's an example of their use:

```
const a = 2;


console.log(
  a === 2, // true
  a !== 2, // false
  a > 2, // false
  a < 2, // false
  a >= 2, // true
  a <= 2, // true
);
```

And another one, for good measure:

```
const age = 25;


let isAdult = age >= 18;

console.log (isAdult); // true
```

```
const isNotAdult = age < 18;

console.log (isNotAdult); // false
```

Negation Operator (!)

To get the opposite value of a variable, we can use the logical negation operator `!`:

```
const age = 25;

let isAdult = age >= 18;


console.log(isAdult); // true

console.log(!isAdult); // false
```

Logical AND (&&)

To check whether two conditions are true simultaneously, we can use the && operator (logical AND). Its result will be `true` only if both conditions are `true`:

```
let cash = 50;

let price = 40;

let hasEnoughCash = cash >= price;


let age = 25;

let isAdult = age >= 18;


let canBuy = isAdult && hasEnoughCash;


console.log(canBuy); // true
```

Logical OR (||)

To check whether at least one condition is true, we can use the || operator (logical OR). Its result will be `true` if at least one of the conditions is `true`:

```
let cash = 50;

let creditCard = 20;

let price = 40;
```

```
let hasEnoughCash = cash >= price;

let hasEnoughCredit = creditCard >= price;


let canBuy = hasEnoughCash || hasEnoughCredit;

console.log(canBuy); // true
```

Useful Keyboard Shortcuts

- `F2` to rename a variable
- `Shift + Alt + ↑` (on Windows) or `Shift + Option + ↑` (on macOS) to duplicate the line up
- `Shift Alt + ↓` (on Windows) or`Shift + Option + ↓` (on macOS) to duplicate the line down

Functions

A function is a block of code that performs a specific task. Functions are the fundamental "building blocks" of a program, and they allow you to reuse the same code multiple times with different data, avoiding duplication.

Function Declaration

In JavaScript, there are several ways to create (synonyms: "declare", "define") a function. For instance, we can do so with a `function` keyword, similar to how `let` lets us declare variables. Here's an exemplary function that calculates the area of a rectangle and prints it to the console:

```
function calculateArea(length, width) {

  let area = length * width;


  console.log(area);

}
```

Let's break the code apart:

- `function` is the keyword that declares the function
- `calculateArea` is the name of the function, followed by parentheses `()`
- `length` and `width` are parameters of the function; they are variables that will receive values when the function is used

All commands inside curly braces `{}` are called the function **body.** Functions can have one parameter, multiple parameters, or no parameters at all.

Function Call

To execute the commands inside a function, we need to **call** the function itself. We do so by writing the function's name and succeeding it with **arguments** in parentheses, which are specific values for the function's parameters. Let's call the `calculateArea` function with arguments `5` and `7`:

```
function calculateArea(length, width) {

  let area = length * width;



  console.log(area);

}



calculateArea(5, 7); // It will print 35 in the console
```

Here, `5` will be assigned to the `length` parameter, and `7` — to the width parameter. But sometimes, **parameters might already have a predefined or default value.** To demonstrate, let's calcualte the area of a fabric roll with a standard width of `1.5` meters:

```
function calculateArea(length, width = 1.5) {

  let area = length * width;



  console.log(area);

}



calculateArea(5); // the `width` value will be 1.5, so the result is 7.5
```

Default values are similar to the default settings on a phone set by the manufacturer. They are predefined, but **you can always change them** by passing new values as arguments when calling the function:

```
function calculateArea(length,  width = 1.5) {

  let area = length * width;



  console.log(area);

}



calculateArea(5); // The `width` is 1.5 because we didn't pass another value, so the result is 7.5

calculateArea(5, 7); // We provided new values for `width`, so the result is 35
```

Returning a Value From a Function

Most often, we don't want to just print the results of calculations to the console, but use them in the program for further actions. Such a function **should return a value.** In the function body, we use the `return` keyword, followed by an expression or value that we want to return:

```
function calculateArea(length, width) {

  let area = length * width;



  return area;

}
```

Previously, we called the function in a separate line, but to use the result of the function in the program, we call it directly — **inside of an expression.** For instance, to calculate the total area of `10` tables:

```
function calculateArea(length, width) {

  let area = length * width;



  return area;

}



let totalArea = 10 * calculateArea(5, 7);
```

If a function doesn't have the `return` keyword or if there's no value to the right of `return`, the result is `undefined`:

```
function calculateArea(length,  width) {

  let area = length * width;



  return;

}



let tableArea = calculateArea(5,7);



console.log(tableArea); // undefined
```

For simplicity, we can remove the area variable declaration and place the calculation expression after the `return` keyword:

```
function calculateArea(length, width) {
  return length * width;
}
```

Any commands following the `return` keyword won't be executed:

```
function calculateArea(length, width) {
  return length * width;

  console.log(area); // this command will not be executed
}
```

Function Naming

The name of a function should convey its purpose. It can consist of multiple words, but the first word should always be a verb that describes what the function does. Similar to variable names, the first word is written in lowercase, while all succeeding words are written in uppercase:

```
// good name
function sayHello(personName) { // explains the purpose of the function
  console.log(`Hello, ${personName}!`);
}

// bad names
function func(personName) { // it is not clear from the name what the function does
  console.log(`Hello, ${personName}!`);
}

function sayhello(personName) { // the second word is written with a lowercase letter
  console.log(`Hello, ${personName}!`);
}
```

Useful Keyboard Shortcuts

- `F2` to rename a variable
- `Shift + Alt + ↑` (on Windows) or `Shift + Option + ↑` (on macOS) to duplicate the line up

- `Shift Alt + ↓` (on Windows) or`Shift + Option + ↓` (on macOS) to duplicate the line down

Conditional Statements

- In this lesson, we're going to explore conditional statements: `if`, `else`, and `else if`. They allow us to execute certain commands only when specific conditions are met.

The `if` Statement

- The `if` statement executes a block of code only if a condition is true, like so:

```
let age = 25;

if (age >= 18) {
  console.log('Hello!');
}
```

- After the `if` keyword, we put a space and parentheses `()`, in which we write the condition. Next, we use curly braces `{}` with one or more commands (code block), executed only if the condition is `true`. In our example, the condition `age >= 18` is `true` because `age = 25`, so the command inside the block is executed and the console displays:

```
Hello!
```

- When using the `if` statement within a function, we don't have to declare the variable `age`. It's a parameter that will receive a new value with each function call, for example:

```
function greetIfAdult(age) {
  if (age >= 18){
    console.log('Hello!');
  }
}

greetIfAdult(15); // nothing will appear in the console
greetIfAdult(25); // the console will display Hello!
```

Checking Multiple Conditions

- To check multiple conditions, we can use `if` within another `if`:

```
if (age >= 18) {
  if (isPresent == true) {
    console.log('Hello!');
  }
}
```

- However, it's hard to track such a nested structure, for instance, where the brackets open and close. **We're better off using logical operators && and || to combine conditions.** Above example can be rewritten as follows:

```
if (age >= 18 && isPresent) {
  console.log('Hello!');
}
```

- A greeting will appear only if both conditions are `true`. The `||` operator is used where it's sufficient for at least one of two conditions to be met:

```
if (age >= 18 || isPresent) {
  console.log('Hello!');
}
```

The `else` Statement

The `else` statement can be added after the `if` code block to specify what actions should occur if the condition isn't met (i.e., it's `false`). For example:

```
let age = 25;


if (age >= 18) {
  console.log('Hello!');
} else {
  console.log('Hi!');
}
```

Here's an example of using `if` and `else` statements in a function:

```
function greet(age) {
  if (age >= 18) {
    console.log('Hello!');
  } else if (age > 7) {
    console.log('Hi!');
  } else if (age > 3) {
    console.log('Hi, kid!');
  } else {
    console.log('Hi, toddler!');
  }
}


greetIfAdult(25); // the condition is true, so `Hello!` will appear in the console
greetIfAdult(15); // the condition is NOT true, so `else` will trigger and `Hi!` will appear in the console
```

The `else if` Statement (Checking Multiple Conditions)

When we have more than two options to choose from, we can use the `else if` statement to check additional conditions. Let's consider an example:

```
if (age >= 18) {
  console.log('Hello!');
} else if (age > 7) {
```

```
  console.log('Hi!');
} else if (age > 3) {
  console.log('Hi, kid!');
} else {
  console.log('Hi, toddler!');
}
```

Note that it's sufficient to add `else` at the end, which will execute if none of the above conditions are met. Here's an example of using `else if` statements in a function:

```
function greet(age) {
  if (age >= 18) {
    console.log('Hello!');
  } else if (age > 7) {
    console.log('Hi!');
  } else if (age > 3) {
    console.log('Hi, kid!');
  } else {
    console.log('Hi, toddler!');
  }
}
```

```
greet(25); // the first condition is true, so `Hello!` appears in the console
greet(15); // the second condition is true (the first is false), so `Hi!` appears
greet(6);  // the third condition executes (first and second are false), so `Hi, kid!` appears
greet(2);  // all previous conditions are false, so the `else` block executes, resulting in `Hi, toddler!`
```

The `return` Keyword Inside `if`

So far, we've looked at examples where conditional statements in functions print messages to the console depending on conditions. However, it's more common for a function to **return a value** with the `return` keyword; a value to be used later in the program. Like here:

```
function getGreeting(age) {
```

```
  if (age >= 18) {
    return 'Hello!';
  } else if (age > 7) {
    return 'Hi!';
  } else if (age > 3) {
    return 'Hi, kid!';
  } else {
    return 'Hi, toddler!';
  }
}

let greeting = getGreeting(6);

console.log(greeting); // Hi, kid! will appear in the console
```

We can simplify this code with separate `if` statements for each option, instead of `else if`, because the function ends immediately after executing the line with the `return` keyword. Each `if` block executes only if all previous conditions were `false`:

```
function getGreeting(age) {
  if (age >= 18) {
    return 'Hello!';
  }

  if (age > 7) {
    return 'Hi!';
  }

  if (age > 3) {
    return 'Hi, kid!';
  }
```

Arrays

Up until now, we've dealt with variables holding strings, numbers, or boolean values (`true` and `false`). But sometimes, we need to group multiple values, such as a list of cities, a compilation of tasks, or any other collection of related items. Arrays let us do exactly that: They're a type of data structure that **holds multiple values in a single variable.**

Creating an Array

We create an array using square brackets `[]`, within which we place the values, separated by commas:

```
let cities = ['Kyiv', 'London', 'Paris'];
```

When we display an array in the console, the values appear as a list, wrapped in square brackets and separated by commas:

```
let cities = ['Kyiv', 'London', 'Paris'];


console.log(cities); // the screen will display ['Kyiv', 'London', 'Paris']
```

Tuple

Arrays typically consist of homogeneous elements. For instance, an array of distances to nearby cities might be represented as:

```
let distances = [10, 30, 50];
```

However, there are times when an array holds related values of various types, such as details about a person or a product. We call such an array with mixed types a tuple. Here's an example:

```
let person = ['John', 25, false]; // tuple
```

Array Length

An array's length refers to the total number of elements it contains. To find out the length of an array, we use the `length` property. For example:

```
let cities = ['Kyiv', 'London', 'Paris'];


console.log(cities.length); // the console will display 3
```

Accessing an Element by Index

You can access an array element by using its index within square brackets. Index counting **starts from zero.** For instance:

```
let cities = ['Kyiv', 'London', 'Paris'];


console.log(cities[0]); // the console will display 'Kyiv'

console.log(cities[1]); // the console will display 'London'
```

The index of the last element is always *one less* than the array's length:

```
let cities = ['Kyiv', 'London', 'Paris'];


console.log(cities[cities.length - 1]); // the console will display 'Paris'
```

Replacing an Element by Index

We can modify an array element through its index by simply assigning it a new value. For example:

```
let cities = ['Kyiv', 'London', 'Paris', 'Tokyo'];


cities[2] = 'New York';


console.log(cities); // the console will display ['Kyiv', 'London', 'New York', 'Tokyo']
```

Adding an Element at the End of the Array

To append a new element to the array's end, we can assign it to an index **equal to the array's current length:**

```
let cities = ['Kyiv', 'London', 'Paris', 'Tokyo'];


cities[cities.length] = 'New York';


console.log(cities); // the console will display ['Kyiv', 'London', 'Paris', 'Tokyo', 'New York']
```

Alternatively, we can use the `push()` method to add one or more new elements to the array's end. These elements will be placed after the last existing one:

```
let cities = ['Kyiv', 'London', 'Paris', 'Tokyo'];


cities.push('Amsterdam', 'New York');
```

```
console.log(cities); // the console will display ['Kyiv', 'London', 'Paris', 'Tokyo', 'Amsterdam', 'New York']
```

Loops

In your programs, you'll often need to execute one command set more then once. **Loops serve this exact purpose,** and today we'll focus on a single loop type — the `for` loop. Here's an example of a loop that triggers `console.log` five times, a.k.a, a loop with five iterations:

```
for (let age = 1; age <= 5; age = age + 1) {

  console.log(`I am ${age}`);

}
```

Let's break the loop's building blocks:

- The `for` keyword initiates the loop
- Inside the `parentheses ()`, there are three components:
    - The initial command `let age = 1`, executed only once before the loop starts
    - The condition `age <= 5`, checked before each loop iteration
    - The command `age = age + 1`, executed after each loop iteration
- The command `console.log(I am ${age})`, placed inside the curly braces `{}`, forms the loop body and is executed on each iteration

...and how they come to work together:

1. The command `let age = 1` is executed.
2. The condition `age <= 5` is checked.
3. Since the condition is `true` (`1 <= 5`), the loop body is executed.
4. The text `I am 1` is displayed in the console (since `age = 1`).
5. The command `age = age + 1` is executed, and `age` becomes `2`.

Steps second through fith are repeated as long as the condition is met. When age reaches `6`, the condition becomes `false`, and the loop ends. In the console, we'll see:

```
I am 1

I am 2

I am 3

I am 4

I am 5
```

Shortcut Assignment

We can write the `age = age + 1` command with the shortcut assignment operation `age += 1`. This lets us increase the value by any number, for example:

```
age += 1; // equivalent to age = age + 1;

age += 10; // equivalent to age = age + 10;
```

Shortcut assignment can also be used for other arithmetic operations:

```
age -= 5; // equivalent to age = age - 5;

age *= 5; // equivalent to age = age * 5;

age /= 5; // equivalent to age = age / 5;

age %= 5; // equivalent to age = age % 5;
```

Most often, the number in a loop is incremented or decremented by `1`, and thus we can write these operations as:

```
age++; // equivalent to age += 1;

age--; // equivalent to age -= 1;
```

💡 The operation `++` is called an **increment,** and `--` is a **decrement.**

Reverse Iteration

Sometimes, instead of increasing, we need to decrease the loop variable. Consider this example:

```
for (let i = 3; i > 0; i--) {

  console.log(i);

}
```

Here, we store `3` in the loop variable `let i = 3`, and we decrease its value by `1` with each iteration. In the console, we'll see:

```
3

2

1
```

Sum of Numbers

A loop can be used to sum numbers, like so:

```
function sumFromTo(min, max) {

  let sum = 0;
```

```
  for (let n = min; n <= max; n++) {
    sum += n;
  }

  return sum;
}

console.log(
  sumFromTo(1, 5) // will output 15
);
```

Loop with Different Steps

In JavaScript, we can create loops with different steps. This means the loop variable can be increased or decreased by any specified value, not just 1. Below is an example where the loop step is passed as a third parameter when calling the function:

```
function sumFromTo(min, max, step) {
  let sum = 0;

  for (let n = min; n <= max; n += step) {
    sum += n;
  }

  return sum;
}

console.log(
  sumFromTo(1, 5, 2) // will output 9
);
```

Factorial (Product of Numbers from 1 to N)

Factorial of a natural number `N` is the **product of all natural numbers from 1 to `N`.** We can calculate it with a `for` loop:

```
function factorial(N) {
  let result = 1; // start with 1

  for (let n = 1; n <= N; n++) {
    result *= n; // multiply the product of previous numbers by the current number n
  }

  return result; // return the calculated factorial
}

const result = factorial(5);

console.log(result); // will output 120
```

💡 Natural numbers are the numbers that appear when counting objects or their order. These are numbers like 1, 2, 3, 4, … The set of natural numbers is commonly denoted by N.

Iterating Over String Characters

In this lesson, we'll learn how to iterate through a string and create a new string based on an existing one.

Iteration with Index

To iterate over the characters in a string, we can use a `for` loop and iterate over indices from `0` to the last one. Usually, the loop variable is named `i`, short for `index`. For example:

```
const title = 'String';

for (let i = 0; i < title.length; i++) {
  console.log(title[i]);
}
```

The console will output:

```
S

t

r

i

n

g
```

If we don't need all characters, we can both start the iteration from a later and conclude the iteration prematurely. For instance:

```
const title = 'String';


for (let i = 2; i < 5; i++) {

  console.log(title[i]);

}
```

The console will output:

```
r

i

n
```

Reverse Order Iteration

To iterate over the characters in reverse order, start from the last index and decrease it in the loop:

```
const title = 'String';


for (let i = title.length - 1; i >= 0; i--) {

  console.log(title[i]);

}
```

The console will output:

```
g
```

```
n

i

r

t

S
```

Iterating without an Index

We can also iterate over the characters of a string using a `for` of loop, which doesn't require indices:

```
const title = 'String';


for (let char of title) {

  console.log(char);

}
```

Counting Characters

Using a loop, we can count the number of specific characters in a string, such as spaces:

```
const text = 'Working with Strings';

let spacesCount = 0;


for (let char of text) {

  // counting the number of spaces in the string

  if (char === ' ') {

    spacesCount++;

  }

}


console.log(spacesCount); // 2
```

Removing Characters from a String

We can't modify an existing string, but we *can* create a new string and fill it with the desired characters. Here's a loop that copies all characters to a new string **except spaces:**

```
const text = 'Working with Strings';

let result = '';


for (let char of text) {

  if (char !== ' ') {

    // adding everything except spaces

    result += char;

  }

}


console.log(result); // 'WorkingwithStrings'
```

Creating a Reversed String

To create a string with letters in reverse order, we can iterate through the input string by indices from the end:

```
const name = 'String';

let reversed = '';


for (let i = name.length - 1; i >= 0; i--) {

  reversed += name[i];

}


console.log(reversed); // 'gnirtS'
```

Alternatively, we can iterate through the letters in normal order **but add them to the beginning** instead of the end:

```
const name = 'String';
```

```
let reversed = '';


for (let char of name) {
  // adding each character before those already added
  reversed = char + reversed;
}


console.log(reversed); // 'gnirtS'
```

Replacing Characters Using a Loop

Sometimes while working with strings, we may need to replace spaces with other characters, like hyphens. This can be useful when creating URL slugs. A *for* loop will be perfect for this purpose:

```
let title = 'Working with Strings';
let result = '';


for (let i = 0; i < title.length; i++) {
  if (title[i] === ' ') {
    result += '-';
  } else {
    result += title[i];
  }
}


console.log(result); // 'Working-with-Strings'
```

The same task can be accomplished with a `for of` loop:

```
function replaceAll(input, char, replacement) {
  let result = '';


  for (let ch of input) {
```

```
    if (ch === char) {
      result += replacement;
    } else {
      result += ch;
    }
  }

  return result;
}

let title = 'Working with Strings';

console.log(
  replaceAll(title, ' ', '-')
); // 'Working-with-Strings'
```

String Methods

In this lesson, we'll dive into methods that make handling strings easier and more efficient.

Text Case

The methods `toUpperCase()` and `toLowerCase()` convert all characters of a string to upper and lower case, respectively, and return the modified string. Here's how they work:

```
let title = 'Working with Strings';

console.log(
  title.toLowerCase(), // outputs 'working with strings'
  title.toUpperCase() // outputs 'WORKING WITH STRINGS'
  title, // 'Working with Strings'
```

```
);
```

Importantly, these methods are **applicable to letters only.** To check, if a character is a letter, use below feature:

```
function isLetter(ch) {
  return ch.toLowerCase() !== ch.toUpperCase();
}

console.log(
  isLetter('a'), // true
  isLetter('B'), // true
  isLetter('1'), // false
  isLetter(','), // false
  isLetter(' '), // false
);
```

Replacing All Occurrences

The `replaceAll` method replaces every instance of a specified substring in a string with another given substring. It takes two arguments, first — the substring to find, second — the replacement string. Here's an example:

```
let title = 'Working with Strings';

console.log(
  title.replaceAll(' ', '-'), // 'working-with-strings'
  title.replaceAll('with', '**'), // 'working-**-strings'
);
```

`replaceAll()` is **case-sensitive,** so it only replaces instances that exactly match the case of the search string.

Substring Search

The `includes()` method checks if a string contains a certain character or substring. We specify what we're looking for in the method's parameters, like so:

```
let title = 'Working with Strings';


console.log(
  title.includes(' '), //true
  title.includes('with'), //true
  title.includes('With'), //false
);
```

`includes()` is **also case-sensitive,** meaning it treats uppercase and lowercase letters as distinct.

Searching for an Index

To find the first appearance of a character or substring within a string, you can use the `indexOf()` method. If the character or substring is present, `indexOf()` returns the index of its first occurrence. If not, it returns `-1`.

```
let title = 'Working with Strings';


console.log(
  title.indexOf(' '), //7
  title.indexOf('with'), //8
  title.indexOf('Strings'), //13
);
```

Working with Arrays

In real life, we often encounter large amounts of homogeneous data: words in a dictionary, grades in a journal, products in a store, books on a shelf, and so on. As we've previously discussed, arrays in JavaScript are used to store such data in a single variable, and today, we'll consider a few of their applications.

Sum of Numbers in an Array

A common operation with arrays is summing numbers. For example, if an array contains the prices of items in a shopping cart, we can calculate the total cost. Let's calculate the sum of numbers in an array using a for loop:

```
const prices = [10, 20, 30, 40, 50];

const totalPrice = getSum(prices);


console.log(totalPrice);


function getSum(values) {
  let sum = 0;


  for (let i = 0; i < values.length; i++) {
    sum += values[i];
  }


  return sum;
}
```

Here's how above code works:

1. We call the `getSum` function and pass our `prices` array to it.
2. Inside the function, we declare a `sum` variable where the total of the numbers will accumulate.
3. Using a `for` loop, we iterate over the array elements and add the current `values[i]` to the sum.
4. Once the loop concludes, we `return` the value of `sum`.

The same can be done with a `for of` loop:

```
const prices = [10, 20, 30, 40, 50];

const totalPrice = getSum(prices);


console.log(totalPrice);


function getSum(values) {
  let sum = 0;
```

```
  for (const value of values) {
    sum += value;
  }

  return sum;
}
```

Average Value

To find the average value of numbers in an array, we need to compute the sum of all elements and divide it by the number of elements. For example:

```
function getAverage(values) {
  let sum = 0;

  for (const value of values) {
    sum += value;
  }

  return sum / values.length;
}

const prices = [10, 20, 30, 40, 50];
const averagePrice = getAverage(prices);

console.log(averagePrice);
```

However, if the array is empty, dividing by `0` will result in a computational error `NaN`. This case should be handled separately at the beginning of the function, like so:

```
function getAverage(values) {
  if (values.length === 0) {
    return 0;
```

```
  }

  // the rest of the code is the same as before
}
```

Minimum and Maximum Value

Sometimes, we need to find the smallest or largest value in an array. Let's find the smallest value, e.g.:

```
function getMin(values) {
  if (values.length === 0) {
    return 0;
  }

  let min = values[0];

  for (const
value of values) {
if (value < min) {
min = value;
}
}

return min;
}

const prices = [10, 20, 30, 40, 50];
const minPrice = getMin(prices);

console.log(minPrice); // Outputs the minimum value in the array
```

Working with Arrays of Strings

Performing operations on arrays of strings can be useful when dealing with lists of names, products, or any other textual information.

Converting a String into an Array (`split`)

The `split` method "slices" a string into separate parts based on a specified delimiter and returns an array of these parts. Said delimiter is provided as an argument to the `split` method. If the delimiter is an empty string, we get an array of individual characters:

```
const text = 'one two three four five';

const chars = text.split('');


console.log(chars); // ['o', 'n', 'e', ..., 'f', 'i', 'v', 'e']
```

If the delimiter is a space ' ', the string is split into separate words:

```
const text = 'one two three four five';

const words = text.split(' ');


console.log(words); // ['one', 'two', 'three', 'four', 'five']
```

A delimiter can also be a string of multiple characters:

```
const text = 'one two three four five';

const words = text.split(' f');


console.log(words); // ['one two three', 'our', 'ive']
```

Filtering Array Elements

Sometimes, we must create a new array with elements from the original array which meet a certain condition. For example, to filter out short words and compile them into a new array:

```
function getShortWords(words, maxLength) {

  let results = [];


  for (const word of words) {

    if (word.length <= maxLength) {

      results.push(word);

    }
```

```
  }

  return results;
}

const text = 'one two three four five';
const words = text.split(' ');

console.log(
  getShortWords(words, 4) // Filters words with 4 or fewer characters
);
```

Creating a New Array from an Existing One

Using a loop, we can receive an array of data (input array) and create a new array (resulting array). To exemplify, here's a function that takes an array of words and returns an array of their lengths:

```
function getWordsLengths(words) {
  const result = [];

  for (const word of words) {
    result.push(word.length);
  }

  return result;
}

// Returns [3, 3, 5, 4, 4]
getWordsLengths(['one', 'two', 'three', 'four', 'five']);
```

Joining Array Elements into a String (join)

We use the `join` method to concatenate all elements of an array into a single string. This method takes one argument, a delimiter (symbol or string), to be inserted between the joined array elements:

```
const fruits = ['apple', 'banana', 'cherry', 'date'];

const joinedString = fruits.join(', '); // The delimiter is a comma and a space


console.log(joinedString); // Outputs 'apple, banana, cherry, date' to the console
```

Searching for Elements in an Array

To check if an array contains a specific element, you can use the `includes` and `indexOf` methods. They function similarly to their string method counterparts:

```
const words = ['apple', 'banana', 'cherry', 'date'];


console.log(words.includes('apple')); // Use `includes` to search for an element in the array, result — `true`

console.log(words.indexOf('apple')); // Use `indexOf` to find the index where the desired element starts, result — `0`
```

💡 These methods are useful when we need to find an element in an array. For more complex search criteria, we can use a loop and check conditions inside it using `if`.

Code Style

Great code not only works correctly, but it's also ***readable***, so it'll be easy to understand and expand when needed. Have you ever set had to set aside an interesting book or TV series, and had a hard time following the plot when you came back to it a couple weeks later?
This happens in programming too — which is why it's important to establish and keep to ***coding standard***. It's a set of rules on how code should be written, so it's easier to go back to your own code, or read someone else's. These rules can vary across different projects. Here are the rules we use at Mate Academy.

Writing Guidelines

- Our code should **always be understandable**.
- Our code should be shortened only **if we can keep it uncomplicated**.
- When commenting our code, comments should explain **why** a decision was made, not how it works (which should be clear from the code itself).
- We should always remove redundant code (don't leave code that's commented out).

Commands

- We will write each command on a new line, so that its boundaries are easier to find.
- To avoid ambiguity in the code, we will end each command with a semicolon `;`.
- We should strive for line lengths that don't exceed `80` characters for comfortable readability.

Code Blocks

Always enclose the bodies of loops, functions, and `if` blocks in curly braces `{ }`:

```
// good
if (condition) {
  result += 123;
}

for (let i = 0 ; i < text.length; i++) {
  result = text[i] + result;
}
// bad
if (condition) result += 123;;

for (let i = 0 ; i < text.length; i++)
  result = text[i] + result;
```

- Place the opening bracket { on the same line as the start of the block.
- Place the closing bracket } on the line following the last command of the block.

```
// good
if (condition) {
  result = 123;
} else {
  result = 456;
}

function sum(a, b) {
  return a + b;
}
// bad
if (condition) { result = 123; }
else { result = 456; }

function sum(a, b)
```

```
{
  return a + b;
}
```

INDENTATION

Nested code should be indented with **two** spaces more than the outer code:

```
// good
if (condition) {
  console.log(456)
  console.log(123)
}
// bad
if (condition) {
    console.log(456)
console.log(123)
}
```

We should separate logical blocks at the same nesting level with an empty line. This includes functions, loops, `if` blocks, `return` statements, variable blocks, and groups of related commands:

```
// good
function test() {
  let x = 10;

  while (x > 5) {
    x--;
    console.log(x);
  }

  return x;
}
// bad
function test() {
```

```
  let x = 10;

  while (x > 5) {


    x--;


    console.log(x);


  }

  return x;


}
```

VARIABLES

We will be declaring variables using `let` or `const`, **but not** `var`, which is an outdated method. Each variable should be declared separately.

```
// good

let x = 1;

let y = 2;

// bad

let x = 1, y = 2;
```

We should write variable and function names in camelCase for variable and function names. Avoid using underscores _, and make sure all words after the first start with a capital letter:

```
// good

let userName = '';

// bad

let user_name = '';

let UserName = ''; // can be used in some situations
```

Single-letter variables are permissible only where their meaning is clear e.g.:

- As an index in a loop;
- For mathematical variables (x, y, a, b, c);
- If it significantly shortens and simplifies the code, making it more understandable.

Variable names should be written in plain English and be ***descriptive***. This means it should be clear from the name what the variable is storing. For boolean values, use verbs in the third form, passive voice, or modal verbs (`isLoaded`, `loaded`, `hasEnoughMoney`, `canBuy`). For arrays, use nouns in the plural. Do not use prefixes like `arr`, `obj`, etc. — the name should be clear without them. Function names should start with a verb and describe what they do.

💡 Event handlers are the exception to these rules, as they have specific names like `onClick` or `submitHandler` – but we will discuss them later.

What NOT To Do: Examples of Poor Coding Style

- DON'T use abstract variable and function names (`obj`, `data`, `value`, `item`, `elem`, etc.) if you can help it.
- DON'T use similar variable names (`num1` and `num2`), as they can easily lead to typos.
- DON'T reuse variables and parameters for new values, making it unclear what they currently store.
- DON'T create a function that's doing something more or something else than its name would suggest (causing a side effect).

Conditional (Ternary) Operator

Sometimes it's necessary to assign different values to a variable based on a condition. This can be done using `if` and `else`:

```
let age = 23;

let result = '';



// condition

if (age >= 18) {

  // value1

  result = 'Adult';

} else {

  // value2

  result = 'Not Adult';

}



console.log(result); // 'Adult'
```

However, this may be problematic when our programs gets larger. The construct can become fairly large, and it prevents us from using use `const` for the `result` variable, as we need to assign a new value inside the `if - else` block.

A shorter and more convenient option is the conditional (ternary) operator. Let's look at an example:

```
let age = 23;
```

```
const result = age >= 18 ? 'Adult' : 'Not Adult';


console.log(result); // 'Adult'
```

If the condition `age >= 18` is met (as in our example), then the value `'Adult'` following the `?` is returned. Otherwise, the statements returns `'Not Adult'`.
For clarity, we can enclose the expression's condition in parentheses:

```
const result = (age >= 18) ? 'Adult' : 'Not Adult';
```

If the expression is lengthy, we can break the values into multiple lines:

```
const result = (age >= 18)

  ? 'Adult'

  : 'Not Adult';
```

This operator is often called ***ternary***, as it's the only operator that uses three parts.

JSDoc

Starting from this lesson, to improve auto-complete features of our IDE, we will be adding comments in the JSDoc format to each function:

```
/**

 * @param {number} a

 * @param {number} b

 *

 * @returns {string}

 */

function getSumText(a, b) {

  return `${a} + ${b} = ${a + b}`;

}
```

The `@param {type} name` comment tells the editor that the parameter named name should be of the type specified in curly brackets `{}`. This can be `number`, `string`, `boolean`, `null`, `undefined` for primitive values, or `number[]` for an array of numbers, `string[]` for an array of strings, etc.
You can also specify two or more types using the pipe character `|`, indicating that the parameter can be one of those types. For example:

```
/**

 * @param {number|string} id

 */
```

```
function check(id) {}
```

Similarly, the type after `@returns {type}` suggests the type of value that the function should return.

Working with Numbers

As software developers, we often need to perform operations on numbers: round them, convert user inputs into numbers, generate random numbers, and so on. In this lesson, we'll practice executing these routine tasks.

Numeric Literals

In JavaScript, numbers can be either integers or decimals, and can be represented in various number systems or formats. Here's how you can represent the decimal number `123` in different systems:

```
let dec = 123; // typical decimal (Base-10) system, requires no prefix

let bin = 0b1111011; // `0b` is the prefix for binary (Base-2). The actual number value is '1111011`

let oct = 0o173; // `0o` is the prefix for octal (Base-8). The actual number value is `173`

let hex = 0x7b; // `0x` is the prefix for hexadecimal (Base-16). The actual number value is `7b`
```

We will be mostly working with decimal numbers in our projects. They can also be expressed in several ways:

```
let x = 15 // An integer decimal number

let y = 0.52 // A decimal number with a fractional part

let z = 3.14e6 // `e6` means multiply by `10` to the `6th` power, so the number is `3140000`
```

Binary and Hexadecimal Numbers

Our familiar decimal system is positional, so each digit is multiplied by 10 to the corresponding power (position from the right, starting from 0). Consider the number 345:

```
345 === 3 * (10 ** 2) + 4 * (10 ** 1) + 5 * (10 ** 0)

345 === (3 * 100) + (4 * 10) + (5 * 1)
```

In the binary system, there are only digits `0` and `1`, which need to be multiplied by `2` to the corresponding power:

```
0b101101 === 1 * (2 ** 5) + 0 * (2 ** 4) + 1 * (2 ** 3) + 1 * (2 ** 2) + 0 * (2 ** 1) +
1 * (2 ** 0)

0b101101 === (1 * 32) + (0 * 16) + (1 * 8) + (1 * 4) + (0 * 2) + (1 * 1)

0b101101 === 32 + 8 + 4 + 1 === 45
```

In the hexadecimal system, digits from `0` to `9` and letters `A` to `F` are used, where `A = 10`, `B = 11`, `C = 12`, `D = 13`, `E = 14`, and `F = 15`:

```
0x9F2C = 9 * (16 ** 3) + 15 * (16 ** 2) + 2 * (16 ** 1) + 12 * (16 ** 0)
```

```
0x9F2C = (9 * 4096) + (15 * 256) + (2 * 16) + (12 * 1)

0x9F2C = 36864 + 3840 + 32 + 12 === 40748
```

Infinity and NaN

JavaScript has two special numeric values:

`Infinity` is the mathematical infinity. It appears when dividing by `0` and is larger than any other number. There is also the opposite value `-Infinity`, which is smaller than any other number. The function `isFinite()` converts its argument to a number and returns `true` if it's a regular number, not `NaN` or `Infinity`.

`NaN` (Not a Number) is a special numerical value that's returned if the program is trying to calculate something that logically cannot be calculated. It may pop up, for example, when we divide `0` by `0`, or divide a number by a string that can't be converted to a number (`346 / 'apple'`). The `NaN` value is unique in that it doesn't equal anything else – not even itself:

```
console.log(NaN === NaN); //false
```

There's also a function `isNaN`, which converts its argument to a number and determines whether it is `NaN` (not a number) or not:

```
isNaN(NaN); //true

isNaN(45.2); //false

isNaN(Infinity); //false

isNaN('123'); //false

isNaN('0x123'); //false

isNaN('12 hello'); //true
```

Rounding Values

Very often, calculations result in fractional numbers, not integers. We can round these up or down. To do so, we use the following commands:

1. `Math.round(x)` — rounds x to the nearest integer (up or down):

```
console.log(

  Math.round(10.1), //10

  Math.round(10.5), //11

  Math.round(10.8), //11

  Math.round(-10.1), //-10

  Math.round(-10.5), //-10

  Math.round(-10.8), //-11

);
```

2. `Math.floor(x)` — rounds x down (to the nearest smaller integer):

```
console.log(
```

```
  Math.floor(10.1), //10
  Math.floor(10.5), //10
  Math.floor(10.8), //10
  Math.floor(-10.1), //-11
  Math.floor(-10.5), //-11
  Math.floor(-10.8), //-11
);
```

3. `Math.ceil(x` — rounds x up (to the nearest larger integer):

```
console.log(
  Math.ceil(10.1), //11
  Math.ceil(10.5), //11
  Math.ceil(10.8), //11
  Math.ceil(-10.1), //-10
  Math.ceil(-10.5), //-10
  Math.ceil(-10.8), //-10
);
```

4. `Math.trunc(x)` — removes the fractional part:

```
console.log(
  Math.trunc(10.1), //10
  Math.trunc(10.5), //10
  Math.trunc(10.8), //10
  Math.trunc(-10.1), //-10
  Math.trunc(-10.5), //-10
  Math.trunc(-10.8), //-10
);
```

5. `x.toFixed(numberOfDigits)` – converts a number to a string with a specified number of digits after the decimal point:

```
console.log(
  (3.14159).toFixed(0), //'3'
  (3.14159).toFixed(2), //'3.14'
  (3.14159).toFixed(3), //'3.142'
```

```
  (3.14159).toFixed(4), // '3.1416'
  (3.14).toFixed(4), // '3.1400'
  (3).toFixed(4), // '3.0000'
  (-3.14159).toFixed(4), // '-3.1416'
);
```

Generating Random Numbers

The `Math.random` method returns a random number from `0` (inclusive) to `1` (exclusive):

```
console.log(
  Math.random(), // 0.6460395007021751
  Math.random(), // 0.511311069945108
  Math.random(), // 0.36258333704200685
);
```

To generate random integer numbers from `0` to `n`, we can multiply the result by `n + 1` and round down (to the smaller integer):

```
Math.floor(Math.random() * 11); // Random integer from 0 to 10 inclusive
```

For random integers, ranging from `min` to `max` inclusive, we can generate a number from `0` to `max - min`, and then add `min`:

```
Math.floor(Math.random() * (max - min + 1)) + min;
Math.floor(Math.random() * (11 - 7 + 1)) + 7; // Random integer from 7 to 11 inclusive
Math.floor(Math.random() * (20 - 10 + 1)) + 10; // Random integer from 10 to 20 inclusive
```

Converting a String to a Number

The standard `Number` function works similarly to the `+` operator for converting strings to numbers:

```
console.log(
  Number(text), // 3.14159
  Number('27.5'), // 27.5
  Number('2.4e3'), // 2400 - Exponential format
  Number('0xABC'), // 2748 - Hexadecimal number
  Number('0b1001'), // 9 - Binary number
);
```

We can also convert other value types into numbers:

```
console.log(+true); // 1
```

```
console.log(+false); // 0

console.log(+null); // 0

console.log(+undefined); // NaN
```

If a full conversion of the string to a number isn't possible, the result of the conversion will be `NaN`:

```
const text = 'Hello';


console.log(

  +text, // NaN - text isn't a number

  +'35$', // NaN - $ isn't a valid number symbol

  +'2.4.3', // NaN - a second decimal point isn't allowed

  +'0xABCZ', // NaN - Z isn't a valid symbol in hexadecimal

  +'0b10012', // NaN - 2 isn't a valid symbol in binary

);
```

For partial conversion of a numeric string, we can use the standard functions `parseInt` and `parseFloat`.
`parseInt` takes two parameters: the value to convert and the number system (optional), and returns an integer or `NaN` if the string doesn't start with a number:

```
parseInt('256$'); // 256 - stops at the first character that isn't part of an integer

parseInt('123.456'); // 123 - the fractional part is discarded

parseInt('256', 2); // NaN - because such digits don't exist in binary

parseInt('1A'); // 1 - the letter `A` is ignored

parseInt('1A', 16); // 26 — hexadecimal number

parseInt('abc'); // NaN - the string doesn't start with a number in decimal
```

`parseFloat` takes a value to convert and returns a number or `NaN` if conversion is impossible, for example:

```
parseFloat('3.14'); // 3.14

parseFloat('2.718 is the base of natural logarithm'); // 2.718

parseFloat('abc'); // NaN, because the string doesn't start with a number
```

This wraps up our overview of working with numbers in JavaScript. With these tools and techniques, you'll be well-equipped to handle various numerical operations, which are commonly required in programming.

Loops in Detail

Loops allow us to execute a block of code **repeatedly**, often until a certain condition is met. They're incredibly useful for tasks that require repetitive actions, like processing items in a list or executing a set of instructions multiple times.

Infinite Loop

A loop **is infinite if its condition is always** **`true`**, which means it will execute indefinitely until forcibly closed or the user's computer is turned off. Most browsers can detect these "runaway" conditions, and will offer to stop the execution of an infinitely looping script.
Example of an infinite loop:

```
for (let i = 2; i > 1; i++) {

  console.log(i);

}
```

BREAK

The `break` statement allows us to **exit a loop at any moment**, even if its condition is still `true`. When this command is invoked, the execution of the loop body is interrupted, and the code following the loop begins to execute. For example:

```
for (let i = 2; i > 1; i++) {

  if (i === 9) {

    break;

  }



  console.log(i);

}
```

Now, as soon as `i` becomes 9, the loop will end. The value 9 won't even be printed to the console.

CONTINUE

Sometimes it's necessary to move to the next iteration of the loop without executing the commands the loop body below. This is where the `continue` command comes in handy:

```
for (let i = 1; i <= 10; i++) {

  if (i === 5) {

    continue;

  }



  console.log(i);

}
```

Here, for `i === 5`, the program won't execute the command `console.log(i);`. Of course, we can also achieve the same outcome by changing the condition to the opposite and moving the command inside an `if` statement:

```
for (let i = 1; i <= 10; i++) {

  if (i !== 5) {

    console.log(i);

  }

}
```

However, as the code becomes more complex, we might inadvertently begin creating nested conditions:

```
for (let i = 1; i <= 10; i++) {

  if (i % 2 === 0) {

    console.log('Even number');


    if (i === 4) {

      console.log(i);

    }

  }

}
```

Which can make things **both hard to read and hard to manage**. The `continue` command lets us avoid this mess of nestedness:

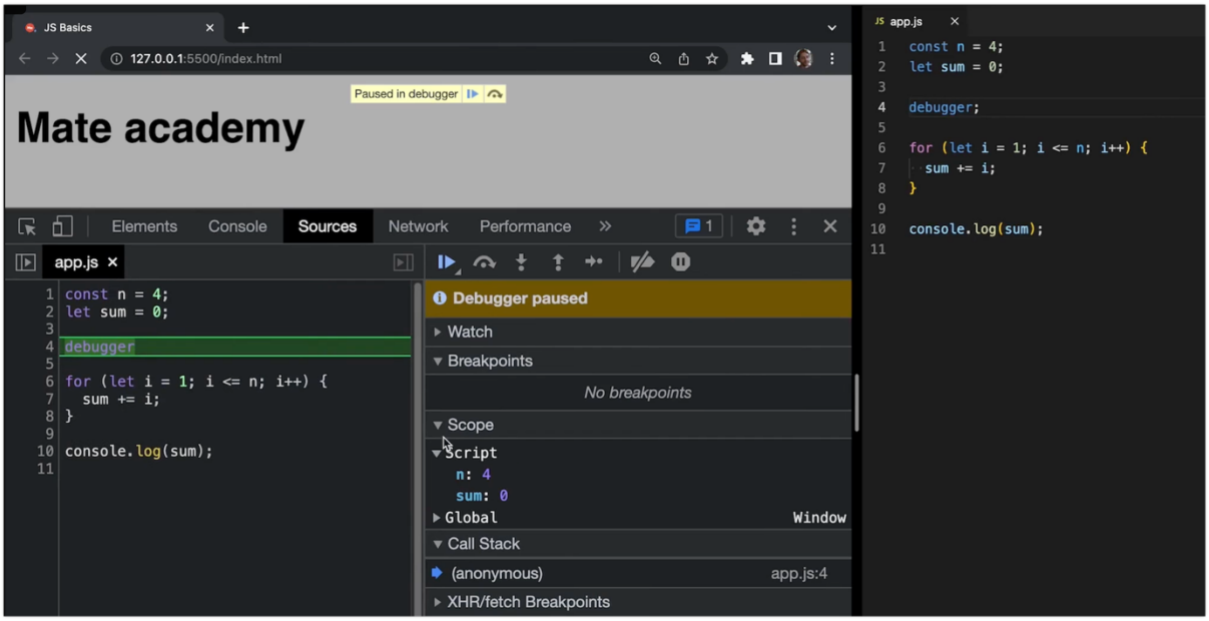
```
for (let i = 1; i <= 10; i++) {

  if (i % 2 !== 0) {

    continue;

  }


  console.log('Even number');


  if (i === 4) {

    console.log(i);

  }

}
```

DEBUGGER

Sometimes our code may not work as expected. To understand why, we could print variable values or messages for debugging purposes using the command `console.log`.

However, browsers give us the option to **halt our program and continue it step-by-step**. To take advantage of this, we need to add the `debugger` command to our code. Afterwards, it's just a matter of opening `Developer Tools` in the browser by pressing `F12` on the keyboard, and reloading the page.

The program will stop at the line with the `debugger` command and wait for further instructions. On the right, in the `Scope` section, we can see the values of the variables.



To continue execution, simply click the arrow buttons at the top, or press the following keys:

- `F10` to move to the next command
- `F11` to enter a function
- `Shift+F11` to exit a function
- `F8` to execute all commands to the next debugger or the end of the program

Try writing a simple `for` loop (as in the screenshot) and executing it step-by-step to gain a better understanding on how the debugger works.

WHILE LOOP

We won't always know in advance how many times a particular command needs to be executed. Say we need to factor a number or find a remainder. This is where the `while` loop comes into play, which executes as long as its condition is `true`.

Take this code, for example:

```
let value = 22;


while (value >= 5) {

  value -= 5; // 17, 12, 7, 2

}


console.log(value); // 2
```

Here, if the condition `value >= 5` is `true`, the body of the loop will execute.

DO WHILE LOOP

If we need to check the condition after the first iteration (not before!), we could use a `do while` loop. It's similar to a `while` loop, but it checks the condition **after executing the loop body**. Here's an exmaple:

```
let i = 0;


do {
  console.log(i);
  i++;
} while (i < 3);
```

In the console, we'll see:

```
0
1
2
```

Let's take a look at another example:

```
let i = 7;


do {
  console.log(i);
  i++;
} while (i < 3);
```

Unlike the loop while `(i < 3)`, this code will print `7` and then terminate, despite the condition being `false` initially. But don't mull over it too much – the `do while` loop is rarely used in practice.

WHILE(TRUE)

Sometimes the same commands need to be executed before the start of the loop, and on each iteration. For example, to print random numbers to the console as long as they are `< 0.9`, we can write the following code:

```
let n = Math.random();


while (n < 0.9) {
  console.log(n);
  n = Math.random();
```

```
}
```

As you can see, we had to write the line to generate a random number twice. A loop with a post-condition won't help here, because it will always print the first number **regardless of the condition**:

```
do {

  let n = Math.random();



  console.log(n);

} while (n < 0.9)
```

Here, the `break` command can help:

```
do {

  let n = Math.random();



  if (n < 0.9) {

    break;

  }



  console.log(n);

} while (n < 0.9)
```

But we're checking the same condition twice. In fact, the condition for `while` is no longer needed. We can replace it with `true`:

```
do {

  let n = Math.random();



  if (n < 0.9) {

    break;

  }



  console.log(n);

} while (true)
```

And now, it doesn't matter where `while` is located. It can be either at the beginning or at the end, and the exit from the loop will occur only **if `break` is triggered**:

```
while (true) {
```

```
  let n = Math.random();


  if (n < 0.9) {
    break;
  }


  console.log(n);
}
```

Working with Strings

When working on our projects, we'll often need to perform various operations on strings, such as retrieving their length or finding a character at a specific position. In this lesson, we'll cover the most common string operations.

Getting Characters in a String

Each character in a string has a specified position, called an ***index***. There are three ways to get a character by its index:

- `str[index]` — returns the character at the index, or `undefined` if `index < 0` or `index >= str.length`.
- `str.charAt(index)` — returns the character, or '' if `index < 0` or `index >= str.length`.
- `str.at(index)` works just like `[index]`, but negative indices mean positions from the end.

```
const fullName = 'Alice Smith';


console.log(
  fullName[0], // 'A'
  fullName[20], // undefined
  fullName[-1], // undefined
  fullName[fullName.length - 1], // 'h'


  fullName.charAt(0), // 'A'
  fullName.charAt(20), // ''
  fullName.charAt(-1), // ''
```

```
  fullName.at(0), // 'A'
  fullName.at(20), // undefined
  fullName.at(-1), // 'h' — the last character
  fullName.at(-2), // 't' — the penultimate character
  fullName.at(-20), // undefined
);
```

Text Case

The methods `toUpperCase()` and `toLowerCase()` return a new string in which all characters of the input string are converted to upper or lower case, respectively.
For example:

```
const name = 'Misha';

console.log(
  name.toLowerCase(), // 'misha'
  name.toUpperCase(), // 'MISHA'

  name[0].toLowerCase(), // 'm'
  name[0].toUpperCase(), // 'M',
);
```

Note: the case can be changed **only** for letters:

```
console.log(
  '1'.toLowerCase(), // '1'
  '1'.toUpperCase(), // '1'
  ','.toUpperCase(), // ','
  ' '.toUpperCase(), // ' '
  ''.toUpperCase(), // ''
);
```

This feature allows you to check if a character is a letter:

```
function isLetter(ch) {
  return ch.toLowerCase() !== ch.toUpperCase();
```

```
}


console.log(
  isLetter('a'), // true
  isLetter('B'), // true
  isLetter('1'), // false
  isLetter(','), // false
  isLetter(' '), // false
);
```

The code below checks if a letter is uppercase:

```
function isBigLetter(ch) {
  return ch !== ch.toLowerCase();
}


console.log(
  isBigLetter('A'), // true
  isBigLetter('a'), // false
  isBigLetter('1'), // false
  isBigLetter(','), // false
  isBigLetter(' '), // false
);
```

Finding a Substring

We can use the `includes()` to check if a string contains a certain character or substring, for example:

```
const name = 'Alice';


console.log(
  name.includes('A'), // true
  name.includes('x'), // false
```

```
  // the string only has uppercase 'J'
  name.includes('a'), // false

  // searching for a substring
  name.includes('Al'), // true
  name.includes('lic'), // true

  // the substring is searched in its entirety, not individual letters
  name.includes('Ai'), // false
);
```

We don't have to search from the beginning of the string. Instead, we can use the second argument to **specify the starting point** (index) for our operation:

```
const word = 'abracadabra';

console.log(
  word.includes('c'), // true

  // 'k' is at the 4th index, but we start checking from the 5th
  word.includes('c', 5), // false

  // if the index is greater than the string length, we return false
  word.includes('b', 15), // false
);
```

Case-Insensitive Search

If we want to check whether a string contains a certain substring regardless of case, we need to convert both the string and the substring **to the same case** (e.g., lowercase):

```
function search(text, part) {
  const normalizedText = text.toLowerCase();
  const normalizedPart = part.toLowerCase();
```

```
  return normalizedText.includes(normalizedPart);
}

console.log(
  search('Alice', 'a'), // true
  search('Alice', 'ICE'), // true
);
```

Searching at the Start and End of a String

We can use the `startsWith()` method to determine if a string **begins** with a specific substring. Similarly, the `endsWith()` lets us check if a string **ends** with a specific substring.
For example:

```
const name = 'Alice';

console.log(
  name.startsWith('Ali'), // true
  name.startsWith('Alice'), // true
  name.startsWith('A'), // true

  name.startsWith('ic'), // false
  name.startsWith('e'), // false
);

console.log(
  name.endsWith('e'), // true
  name.endsWith('ice'), // true
  name.endsWith('Alice'), // true

  name.endsWith('ic'), // false
  name.endsWith('Al'), // false
```

```
);
```

Finding an Index

The `indexOf()` method helps us find out where a character or substring first appears in a string. If the character or substring is found, **the index of the first occurrence is returned.** Otherwise, `-1` is returned. For example:

```
const name = 'My name is Alice';


console.log(
  name.indexOf('y'), // 1
  name.indexOf('n'), // 3
  name.indexOf('My'), // 0

  // character not found
  name.indexOf('x'), // -1

  // case matters
  name.indexOf('my'), // -1

  // the index of the first occurrence
  name.indexOf(' '), // 2
  name.indexOf('e'), // 6

  // uppercase 'A'
  name.indexOf('A'), // 11
);
```

To start the position search from a certain index, we can pass it as the second argument:

```
const name = 'My name is Alice';


console.log(
```

```
  name.indexOf(' '), // 0

  name.indexOf(' ', 4), // 7


  // ' ' doesn't occur starting from index 12

  name.indexOf(' ', 12), // -1

);
```

There's also a method for searching in the reverse direction — `lastIndexOf()`. It returns the index of the last occurrence of a character or substring:

```
const name = 'My name is Alice';


console.log(

  name.lastIndexOf('y'), // 1

  name.lastIndexOf('i'), // 13


  // you can specify the index where the reverse search starts

  name.lastIndexOf('i', 10), // 8

);
```

Removing Whitespace

To remove all spaces at the beginning and end of the string, we can use the `trim()`, `trimLeft()`, and `trimRight()` methods.

```
const name = '       Alice';

const fullName = 'Alice Smith      ';

const word = '     abracadabra     ';


console.log
```

Extracting a Substring

We'll often need to obtain a part of a string (a substring) in our projects. In these cases, the `slice()` method is a convenient tool. It accepts two arguments:

- The index from which to start copying characters.
- The index at which to end copying characters. The character at this index won't be included.

For example:

```
const text = '0123456789';
```

```
console.log(
  // The character at index 5 isn't included
  text.slice(1, 5), // '1234'

  // Taking the first 8 characters
  text.slice(0, 8), // '01234567'

  // If the start is greater than the end, we get an empty string
  text.slice(5, 1), // ''

  // Without arguments, we get the entire string
  text.slice(), // '0123456789'

  // If the second argument isn't passed, we take everything to the end
  text.slice(2), // '23456789'

  // If the start index is too large, we get an empty string
  text.slice(15), // ''

  // Negative indices are counted from the end
  text.slice(-5, -2), // '567'

  // Taking the last 3 characters
  text.slice(-3), // '789'

  // All characters except the first and last
  text.slice(1, -1), // '12345678'
);
```

This method can be used to assemble a new string from parts:

```
const text = 'I have 4 dogs';


// 'I have five dogs'
const text2 = text.slice(0, 7) + 'five' + text.slice(-5);


// 'We don't have dogs'
const text3 = `We don't ${text.slice(2, 6)} ${text.slice(-4)}`;
```

PADSTART AND PADEND

The `padStart` method adds a new string to the beginning of an existing string. The syntax of this method is `padStart(targetLength, padString)`, where `targetLength` is the length of the resulting string, and `padString` is the string to be added to the existing one. The default value for `padString` is a single space:

```
const word = 'fruit';


console.log(word.padStart(10, '*')); // '*****fruit'
console.log(word.padStart(10, ' ')); // '     fruit'
console.log(word.padStart(10)); // '     fruit'
console.log(word.padStart(19, '0123456789')); // '01234567890123fruit'
```

The `padEnd()` method adds a new string to the end of an existing string:

```
const word = 'fruit';


console.log(word.padEnd(10)); // 'fruit     '
console.log(word.padEnd(10, '3')); // 'fruit33333'
console.log(word.padEnd(17, ' yummy')); // 'fruit yummy yummy'
```

REPEAT

The `repeat` method repeats a string a specified number of times:

```
const newString = 'Alice'.repeat(2);


console.log(newString); // 'AliceAlice'
```

CONVERTING TO STRING – AND BACK

Sometimes it's necessary to convert a number to a string. We can do this in several ways:

```
const n = -123;


console.log(
  String(n), //'-123'
  n.toString(), //'-123'
  `${n}`, //'-123'
  '' + n, //'-123'
);
```

This method can be used to obtain an individual digit of a number:

```
function getFirstDigit(n) {
  return String(n)[0];
}


console.log(
  getFirstDigit(123), //'1'
  getFirstDigit(76543), //'7'
  getFirstDigit(0), //'0'
);
```

However, when we're working with characters, the result will always be returned as a string. If a number is needed afterwards, we need to convert it explicitly:

```
const digits = '123';


console.log(
  Number(digits), //123
  +digits, //123
);
```

The + operator is most commonly used in practice.

Special Characters

Some special characters, such as single quotes or line breaks, cannot be simply inserted into a string, as they have a functional purpose in code. Hence, they can be misinterpreted as e.g. the end of a string.

We can "escape" these character using the backslash \, making sure they'll be interpreted as regular characters.
For example:

```
const str = 'first \' line\

second \\ line';



/*

first ' line

second \ line

*/
```

Here, using a backslash \, we escaped the second quotation mark in the word `'first'`. As a result, JavaScript won't interpret it as the end of the string. If we didn't escape it, all subsequent text would be analyzed as commands, not as a string literal.
The \ character has the following meanings:

1. \n adds a line break when placed in single and double quotes:

```
const greeting = 'Hello! \n I am Alex.'



console.log(greeting);

/*

Hello!

I am Alex.

*/
```

2. \\ adds a backslash to the string — the first \ is an escape character, and the second \ goes directly into the string.
3. \t adds a tab to the string.

Character Code and String Comparison

Strings are composed of characters. We already know that each character has an index — its position in the string. Besides the index, each character also has a ***code***.
A character can be added to a string by its code. For example, the © symbol can be added in these ways:

```
console.log('\xA9');

console.log('\u00A9');

console.log('\u{A9}');
```

To obtain a character's code, we can use `charCodeAt` method. This method takes the character's index in the string as a parameter:

```
'Apple'.charCodeAt(0) // `65` — the code for `A`
```

We can either obtain a character's code from the character or get a character from its code. For this, we can use the `String.fromCharCode()` method:

```
String.fromCharCode(65); // 'A'
```

Strings in JavaScript are compared character by character, or more precisely, by the their **character codes**. When comparing strings, remember the following rules:

- lowercase letters are larger than uppercase;
- special characters and diacritics, such as Ö, are ordered after the base Latin alphabet;

To compare strings in a way that respects the peculiarities of a specific alphabet and case, we can use the `localeCompare method`. It has the syntax `str1.localeCompare(str2)` and returns a number:

- Negative, if the string `str1` is smaller than `str2`
- Positive, if the string `str1` is larger than `str2`
- `0` if the strings are equal.

```
console.log('Ö' < 'Z'); // false


// but

console.log('Ö'.localeCompare('Z')); // -1
```

Functions Revisited

In this lesson, we're going to learn how to handle handling missing arguments in functions, and what arrow functions are all about.

Missing Arguments and Default Values

We already know that a ***parameter*** is a variable written between the parentheses of a function during its declaration. In contrast, an ***argument*** is the value we pass to the function when calling it. The arguments **become the values of the corresponding parameters** in order: the first argument becomes the value of the first parameter, the second for the second, and so on. For example:

```
function message(text, name) { // `text` and `name` are parameters

  return `${text}, ${name}!`;

}


const greeting = message('Hi', 'Alex'); // 'Hi' and 'Alex' are arguments


console.log(greeting); // 'Hi, Alex!'
```

If we call a function with fewer arguments than it has parameters, the parameters without arguments will be `undefined`:

```
function message(text, name){

  return `${text}, ${name}!`;

}
```

```
const greeting = message('Hello'); // only one argument 'Hello' is passed


console.log(greeting); // 'Hello undefined'
```

However, we can set default values for parameters, so if we don't declare any later on, these defaults will be used instead of `undefined`:

```
function multiply(a, b = 3) { // if no value for `b` is passed, it defaults to `3`
  return a * b;
}


console.log(multiply(5, 2)); // 10
console.log(multiply(5)); // 5 * 3 = 15
```

We can even use an expression as a default parameter:

```
function multiply(a, b = a + 10) { // if `b` isn't passed, it will be `a + 10`
  return a * b;
}


console.log(multiply(3)); // 3 * 13 = 39
console.log(multiply(7)); // 7 * 17 = 119
```

Extra Arguments

If a function gets more arguments than it expects, the extra ones are simply ignored:

```
function multiply(a, b) {
  return a * b;
}


multiply(3, 4, 5, 6) // поверне `12`, аргументи `5` і `6` ігноруються
```

If we need to capture all passed arguments, we can use the rest operator, such as `...args`. In this case, `args` will be an array containing all the arguments passed to the function:

```
function sum(...args) {
  console.log(args)
}
```

```
sum(3, 4); // args = [3, 4]

sum(3, 4, 5, 6); // args = [3, 4, 5, 6]
```

Also, if needed, we can use a part of the arguments as separate parameters, and the rest as an array:

```
function sum(x, ...args) {
  console.log(x, args)
}


sum(); // x = undefined, args = []

sum(1, 2); // x = 1, args = [2]

sum(3, 4, 5, 6); // x = 3, args = [4, 5, 6]
```

Function Declaration and Function Expression

We've always declared functions `likeThese()` separately from other code. It's worth knowing they can be called before and after their declaration:

```
const result1 = sum(1, 2);


function sum(a, b) {
  return a + b;
}


const result2 = sum(3, 4);
```

But sometimes it's handy to assign a function to a variable. For instance:

```
const operation = function sum(a, b) {
  return a + b;
};


const result = operation(1, 2); // result = 3
```

This way of creating an function ***is called a Function Expression***. It can **only be called after assignment, and only by the name of the variable** it's assigned to:

```
operation(1, 2); // Uncaught ReferenceError: Cannot access 'operation' before initialization
```

```
const operation = function sum(a, b) {

  return a + b;

}


sum(1, 2); // Uncaught ReferenceError: sum is not defined
```

Arrow Functions

Functions are are a mainstay in programming, and to keep things concise, we can use `=>` (the arrow) right after the parentheses instead of `function`. These two functions below do the same thing:

```
const sum1 = function(a, b) {

  return a + b;

};


const sum2 = (a, b) => {

  return a + b;

};
```

If a function is just returning a result without much else going on, we can skip the `{}` and `return`, and simply write the result straight after the arrow:

```
let sum = (a, b) => a + b;
```

When there's only one argument, we can even drop the parentheses:

```
let square = a => a * a;
```

However, if there are no arguments, we DO need the parentheses:

```
let getAnswer = () => 42;
```

Just like in regular functions, we can set default values for parameters and use the `rest operator` in arrow functions:

```
let greet = (greeting = 'Hello', ...names) => {

  console.log(`${greeting} ${names.join(', ')}`);

};
```

These functions have a more streamlined syntax and are a great fit for situations where we want to write less code for the same outcome. They're especially handy in when working with arrays, callbacks, and when we need to keep our code short and sweet.

SWITCH

Sometimes in our programs, we need to take different actions based on a specific value. We can do this using `else if` blocks, like so:

```
const number = 2;


if (number === 1) {
  console.log('Number is 1');
} else if (number === 2) {
  console.log('Number is 2'); // we'll see 'Number is 2' in the console
} else if (number === 3) {
  console.log('Number is 3');
} else {
  console.log('Other number');
}
```

The problem is, reading this code can get tricky. We have to check each condition for strict equality (===), and we need to make sure we haven't added extra conditions to some `if` blocks.
To make our lives easier, we can use a `switch`, which **always checks for strict equality**. It should have at least one `case` block with an optional `break` keyword, and an optional `default` block, like this:

```
const number = 2;


switch (number) {
  case 1:
    console.log('Number is 1');
    break;
  case 2:
    console.log('Number is 2'); // we'll see 'Number is 2' in the console
    break;
  case 3:
    console.log('Number is 3');
    break;
  default:
    console.log('Other number');
```

```
}
```

Here's a short overview how `switch` works:
First, we calculate the expression in parentheses `()` after the `switch` keyword. Then, we compare it with the value after the first `case`. If they match, we execute all commands from that `case` up to the `break`, then exit the `switch`.
If they **don't** match, we check the next `case`. If there's no match between the `switch` and any `case`, we execute commands after the `default` block and exit the `switch`.

A CASE WITHOUT A BREAK

Now, let's see a how a `switch` without any `break` works:

```
const number = 2;



switch (number) {

  case 1:

    console.log('Number is 1');

  case 2:

    console.log('Number is 2');

  case 3:

    console.log('Number is 3');

  default:

    console.log('Other number');

}
```

If `case` blocks don't have any `break` keywords, the `switch` is executed from the first matching `case`, and continues to the nearest `break`. If there's no break (including the default block), it jumps to the end.
In the console, we'll see the following:

```
Number is 2

Number is 3

Other number
```

But we should avoid such scenarios. The proper practice is to group multiple `case`s together, like so:

```
case 1:

  //…

  break;



case 2:

case 3:
```

```
case 4:

  //…

  break;


default:

  //…
```

A SWITCH WITH RETURN

If we're using `switch` inside a function, we can use `return` in a `case`. The function will return the value from that `case`, and commands after `return` won't be executed:

```
const number = 2;


function printMessage(number) {

  switch (number) {

    case 1:

      return 'Number is 1';


    case 2:

      return 'Number is 2'; // the function will return 'Number is 2'


    case 3:

      return 'Number is 3';


    default:

      return 'Other number';

   }

}


console.log(printMessage(number)); // 'Number is 2'
```

SWITCH (TRUE)

For a better understanding of `switch`, let's consider this theoretical example of its use.

If we want to check arbitrary conditions in `switch`, not the exact value of a variable, we can use `switch (true)`. The following two examples will work the same:

```
function getWord(count) {

  if (count <= 4) {

    return 'a few'

  }


  if (count < 10) {

    return 'several'

  }


  return 'a lot'

}

function getWord(count) {

  switch (true) {

    case count <= 4:

      return 'a few';


    case count < 10:

      return 'several';


    default:

      return 'a lot';

}
```

Here, `switch` compares the `case` condition (`true` or `false`) with `true` inside `switch ()` and runs the commands of the first matching `case`.
But such syntax can be confusing to the reader. Therefore, it is better to avoid it and use `if else` blocks.

Type Conversion

JavaScript isn't a strictly typed language, which means we can use values of any type in any expression.

Converting to `boolean`

When checking conditions, the value can be of any type, not necessarily `true` or `false`:

```
const value = 10;



if (value) {

  console.log('Value is not empty');

}
```

In this example, when checking the condition, JavaScript converts the `value` variable to a `boolean` type, and executes the commands in `{}` only if the conversion result is true. There are only 7 values that convert to `false`:

- `false`;
- `0`;
- `0n` — `BigInt` value;
- `NaN`;
- `''` — empty string;
- `undefined`;
- `null`.

All other values will be automatically converted to `true`, even empty arrays `[]` or objects `{}` (we'll talk about objects later). This is what we call ***implicit conversion***.

To ***explicitly*** convert a value, we can use the **`Boolean`** function or double negation operators — `!!` – like so:

```
console.log(Boolean(10)); // true

console.log(!!10); // true
```

Conversion During Comparison

Now, let's compare values of different types. This can happen when our program needs to process data entered by users:

```
console.log(

  2 < 18, // true

  2 < '18', // true - string '18' is converted to a number

  '2' < 18, // true - string '2' is converted to a number

  '2' < '18', // false - two strings are compared character by character

)
```

Values of different types are converted to numbers during comparison, and the two strings are compared character by character.

== vs. ===

To check if two values are equal, we previously used the `===` operator. This operator compares values **without converting types**. However, for the value `NaN`, this operator won't work correctly:

```
console.log(

  NaN === NaN, // false
```

```
);
```

To ensure our checks are reliable, we can use the `Object.is` method:

```
console.log(

  Object.is(NaN, NaN), // true

);
```

We can also use the == operator, which compares values after converting types, **leading to some non-intuitive results**:

```
console.log(

  0 == false, // true

  0 == '', // true

  0 == '0', // true

  '' == '0', // false

  null == undefined, // true

  null == 0, // false

)
```

As you can see, its results don't guarantee that two values will equal each other, if they're also equal to a third. For reliable comparisons, stick to the === operator, which checks for equality without type coercion, avoiding such ambiguities.

Logical Operators

In this lesson, we'll take a detailed look at how the && and || operators work.

Logical OR (||)

Previously, we used the || operator to combine `boolean` values, always getting `true` or `false` as a result. Let's write an `or` function that works in a similar way:

```
function or(a, b) {

  return a ? a : b;

}
```

If the first argument `a` converts to `true` in `boolean` terms, we return `a`. Otherwise, we return `b`. For example:

```
console.log(10 || null); // 10

console.log(null || 10); // 10

console.log(0 || 1); // 1

console.log('' || undefined); // undefined
```

In practice, this can be used to set a default value for a variable:

```
const user = currentUser || 'Unknown';
```

If `currentUser` is empty (converts to `false` in `boolean` terms), the value `'Unknown'` will be used. When chaining `||` (logical OR) operators, the evaluation proceeds from left to right, stopping at the first operand that is not equivalent to `false`. If all operands are equivalent to `false`, the last operand is returned.

```
console.log(0 || 1 || 2 || 3); //1
```

Now let's break it down:

1. We start with `0 || 1`. Since 0 is equivalent to `false`, the evaluation moves to `1`, which is not equivalent to `false` and thus returned.
2. For `1 || 2`, the evaluation stops at `1` because it is not equivalent to false.
3. The same logic applies to `1 || 3`, resulting in 1.

So, multiple uses of `||` return the first `true` value — or the last value if there are no `true` values. Here's another example:

```
console.log(1 || 0 || 2); //1

console.log(false || 0 || 3); //3

console.log(0 || 1 || false || 3); //1
```

Logical AND (&&)

The && operator returns the first operand if it evaluates as `false` in a logical context. If the first operand is not `false`, && then returns the second operand, like this:

```
function and(a, b) {

  return !a ? a : b;

}
```

An extended example would be:

```
console.log(1 && true); //true

console.log(1 && 2); //2

console.log(0 && 1); //0

console.log(1 && 0); //0

console.log(1 && false); //false

console.log(0 && false); //0
```

These features make the && operator useful for conditional function calls:

```
(typeof f === 'function') && f();
```

The call to `f` will only happen if `f` is a function. Of course, we can also achieve the same goal using `if`, which could produce arguably more understandable code.

```
if (typeof f === 'function') {

  f();

}
```

When using several && operators in a row, we get the first `false` value, or the last one if there are no `false` values:

```
console.log(1 && 2 && 3); //3

console.log(1 && 2 && true); //true

console.log(1 && 0 && true); //0

console.log(1 && false && true); //false
```

Combining Conditions

The priority of the && operator is higher than ||, so it will be executed first. We can combine conditions in an expression, for example:

```
console.log(1 && 2 || 3 && 4); //2
```

Let's break it down:

1. We check `1 && 2`, which results in `2`.
2. Then we check `3 && 4`, which results in `4`.
3. Finally, we have `2 || 4`, so the result of the calculation is `2`.

```
1 && 2 || 3 && 4

2 || 3 && 4

2 || 4

2
```

We've noted that programmers often struggle to remember operator priorities. For this reason, we recommend using `()` to avoid potential errors:

```
console.log(1 && (2 || 3) && 4); //4
```

Basics of Working with Objects

Objects are used to group data that's related. Each ***value*** in an ***object*** is stored under a specific name (a ***key***). For example, information about a person can be written like this:

```
const user = {

  firstName: 'John',

  lastName: 'Smith',

  age: 20,

};



console.log(user); //{ firstName: 'John', lastName: 'Smith', age: 20 }
```

In this example, `firstName`, `lastName`, and `age` are ***keys***, and `John`, `Smith`, and `20` are ***values***. A key and a value together are often called an object's ***property***. If a key matches the name of a variable, **adding such a property can be shortened**:

```
const age = 20;

const user = {

  age: 30,

  age: age, // full record

  age, // shortened record

};



console.log(user); // { age: 20 }
```

💡 When a key in an object is added multiple times, only its last value is used.

An object's keys are strings, and they can be enclosed in either single or double quotes. Using quotes allows us to place any character we'd like in the key, including spaces and special characters. However, if you choose not to use quotes, the **key must adhere to specific rules**.

1. It must be a valid variable name, which means it can only include letters, digits, underscores (_), or dollar signs ($). Additionally, it cannot start with a digit.
2. It can be a whole, non-negative number, functioning like an index in an array. Keys that don't meet these criteria must be enclosed in quotes.

Here's an example of these rules in practice:

```
const user = {

  '!@#$%^&*() first name': 'John',

  "1234567890 last name": 'Smith',

  age: 20,

  0: 'zero',

  9: 'nine',

};
```

When creating an object, we can also use square brackets [ ] to compute a key:

```
const prefix = 'last';



const user = {

  [prefix + 'Name']: 'Smith',

  ['a' + 'g' + 'e']: 20,

};



console.log(user); // { lastName: 'Smith', age: 20 }
```

Getting Properties

To get a value stored under a certain key, we can use the `.` operator (user.age) or square brackets with a string inside (user['age']):

```
const user = {

  firstName: 'John',

  lastName: 'Smith',

  age: 20,

};



console.log(

  user.firstName, // John

  user['age'], // 20

);
```

💡 The dot `.` can be used only for a key that's a valid variable name (which can be created without quotes).
As for square brackets `[]`, any string in quotes or an expression to compute the key can be used. For example, we can concatenate or even call a function:

```
function getKey() {

  return 'age';

}



const prefix = 'last';



console.log(

  user[prefix + 'Name'], // user['lastName'] === 'Smith'

  user[getKey()], // user['age'] === 20

);
```

Editing or Adding Properties

To edit or add a new property, we can also use the `.` operator (***dot notation***) or square brackets:

```
const user = {

  firstName: 'John',

  lastName: 'Smith',
```

```
  age: 20,
};

// editing
user.age = 21;
user['firstName'] = 'Michael';

console.log(
  user.firstName, // Michael
  user.age, // 21
);

// adding
user.isMarried = false;
user['hasJob'] = true;

console.log(user);
/*
{
 firstName: 'Michael',
 lastName: 'Smith',
 age: 21,
 isMarried: false,
 hasJob: true
}
*/
```

Deleting Properties

To delete a property from an object, we can use the `delete` operator:

```
const user = {
```

```
  firstName: 'John',
  lastName: 'Smith',
  age: 20,
  hasJob: true,
};

delete user.age;
delete user['hasJob'];

console.log(user); // { firstName: 'John', lastName: 'Smith' }
```

Checking Properties

To check whether an object has a property, we can use the `in` operator. It returns `true` if the object has the specified property:

```
const user = {
  firstName: 'John',
  lastName: 'Smith',
  age: 20,
};

console.log(
  'age' in user, // true
  'isMarried' in user, // false
);
```

The `in` operator has a drawback – it checks for inherited properties too (we'll discuss them later in this course). But there are methods `hasOwnProperty` and `Object.hasOwn` that return `true` if the object has its own (not inherited) property:

```
console.log(
  user.hasOwnProperty('age'), // true
  user.hasOwnProperty('isMarried'), // false
)
```

```
console.log(
  Object.hasOwn(user, 'age'), // true
  Object.hasOwn(user, 'isMarried'), // false
);
```

💡 The `Object.hasOwn` method is the most reliable, but it's fairly new and not supported in all environments. So, `hasOwnProperty()` is still relevant.

Iterating Over Object Properties

To iterate over an object's properties, we can use the `for in` loop:

```
const user = {
  firstName: 'John',
  lastName: 'Smith',
  age: 20,
};

for (let key in user) {
  console.log(`${key} is ${user[key]}`);
}

/*
firstName is John
lastName is Smith
age is 20
*/
```

Additionally, with methods `Object.keys`, `Object.values`, and `Object.entries`, we can create arrays of keys, values, or `[key, value]` pairs, like this:

```
console.log(
  Object.keys(user), // ['firstName', 'lastName', 'age']
  Object.values(user), // ['John', 'Smith', 20]
```

```
  Object.entries(user),
  /*
 [
  ['firstName', 'John'],
  ['lastName', 'Smith'],
  ['age', 20]
 ]
 */
);
```

And iterating over an object can look like this:

Deep Dive into Objects

In the module **JavaScript Basics Extended**, we discussed how to create objects, iterate through them, and check their properties. Now, it's time to delve deeper into working with objects.

References

Let's start by considering three variables:

```
const personName = 'John';
let personAge = 25;


const person = {
  name: 'John',
  age: 25,
};
```

Now imagine that the variables are written on a piece of paper as `name: value`. Then, our variables can be written as follows:

```
personName: 'John'
personAge: 25


person: #1
```

Note that for the object, instead of keys and values, we wrote #1. Let's consider this the "address" or "number" of another sheet where the data of our object is recorded. It would be depicted like this:

```
#1 {
```

```
  name: 'John',

  age: 25,

}
```

Now let's look at what happens when using variables in a program.

Suppose we assign the value of one variable to another:

```
const friendName = personName;
```

To execute this command, JavaScript first needs to find out what is stored in the `personName` variable. On the sheet with variables, opposite it is written the value `'John'`. So, we need to substitute it into our command:

```
const friendName = personName;

// const friendName = 'John';
```

Now we add this variable to our sheet:

```
personName: 'John'

personAge: 25


person: #1


friendName: 'John'
```

And now let's consider such an assignment with a variable that contains a reference to an object:

```
const friend = person;
```

We look on our sheet for what is written opposite `person`, and we see `#1`. We substitute it instead of the variable `person`:

```
const friend = person;

// const friend = #1;
```

And we add a new record to the sheet:

```
personName: 'John'

personAge: 25


person: #1


friendName: 'John'
```

```
friend: #1

#1 {

  name: 'John',

  age: 25,

}
```

Now, we have one object and two variables that refer to it (`person` and `friend`).
Let's change `friend.name` and see how it affects `person`:

```
friend.name = 'Mike';


console.log(friend.name); // Mike

console.log(person.name); // Mike
```

Why did the value of `person.name` change? To answer this question, let's substitute the values instead of the variables:

```
friend.name = 'Mike';

// #1.name = 'Mike';


console.log(friend.name); // Mike

// console.log(#1.name);


console.log(person.name); // Mike

// console.log(#1.name);
```

Now, we see that we are actually changing and reading the properties of the same object.

LET VS CONST

Note that in the previous example, the `person` variable was declared with the keyword `const`:

```
const person = {

  name: 'John',

  age: 25,

};
```

But this didn't prevent us from changing `person.name`:

```
console.log(person.name); // 'John'
```

```
person.name = 'Mike';


console.log(person.name); // 'Mike'
```

However, if we try to assign a new value to the `person` variable, we will get an error:

```
person = 123; // Uncaught TypeError: Assignment to constant variable.
```

Thus, `const` simply prohibits assigning a new value to the variable. But when we change a property of an object, we are actually reading the variable's reference to the object and then making a substitution at that reference:

```
#1 {
  name: 'John'
}

person.name = 'Mike';

// #1.name = 'Mike';

#1 {
  name: 'Mike'
}
```

Let's draw an analogy with life, suppose a variable is a "glass jar" with a name, inside of which lies a "piece of paper" with a written value:

- Strings are written in quotes;
- Numbers and boolean values are written without quotes;
- For objects, we write "references";
- `null` is an empty "piece of paper";
- `undefined` means that there is no "piece of paper" in the jar at all.

The keyword `let` creates an "open" jar, where you can put a new "piece of paper" instead of the old one at any moment. In contrast, `const` creates a "closed" jar, the "piece of paper" in which cannot be replaced. But through the glass, we can still see what is inside.
Then, an object can be imagined as a "cabinet with jars" which has a "number" (reference), by which you can find the cabinet and interact with the "jars" in it.

Cloning Objects

Cloning an object refers to creating a new object with the same properties. To clone an object, you can create an empty object and then use a `for in` loop to copy all properties from the old object:

```
const person = {
  name: 'John',
  age: 25,
  role: 'admin',
};
```

```
const friend = {}; // a new empty object


for (const key in person) {

  friend[key] = person[key];

}
```

Note: key is a variable that will contain the next key of the object on each iteration of the loop (`'name'`, `'age'`, etc.). We use `person[key]` to read the value stored in the object `person` under the current key (`person['name']`, `person['age']`, etc.).
In terms of the "jar" analogy, this can be described as:

- Create a new "cabinet";
- Go through the names of all jars in the old "cabinet";
- In the new "cabinet", we put "jars" with the same names and the same "pieces of paper" as in the old one.

Another way to copy an object is using the **spread operator**:

```
let friend = { ...person };
```

Object.assign Method

If you need to copy all properties from one object to another, you can use the `Object.assign` method. The first object is the target, the second and all subsequent objects are the sources:

```
const point = { x: 0, y: 0 };

const yCoord = { y: 42, z: 999 };

const info = { name: 'A', age: 1 };


Object.assign(point, yCoord, info);


console.log(point); // { x: 0, y: 42, z: 999, name: 'A', age: 1 }

console.log(yCoord); // { y: 42 }

console.log(point === yCoord); // false
```

`Object.assign` returns a reference to the first object. This can be used to clone an object by passing a new empty object as the first argument:

```
const person = {

  name: 'John',

  age: 25,

  role: 'admin',

};
```

```
const friend = Object.assign({}, person);
```

This yields the same result as `{ ...person }`. But the spread operator (`...`) is shorter and has a simpler syntax when combining new properties with the copied ones:

```
const extraData = { salary: 1000 };

const infant = { age: 0 };



const friend = {

  id: 1,

  ...person,

  role: 'user',

  ...extraData,

  ...infant,

};
```

As a result, we get an object with all copied and added properties. The order is important if the keys match, because the last added value is taken.

```
{

  id: 1,

  name: 'John',

  age: 0,

  role: 'user',

  salary: 1000,

}
```

Deep Copying

All the methods to make a copy of an object that we've discussed above allow cloning an object without nesting. But sometimes, an object contains references to other objects, arrays, or methods. Consider the following example:

```
const student = {

  name: 'John',

  marks: [],

};
```

```
const anotherStudent = { ...student };


student.marks.push(50);

anotherStudent.marks.push(99);


console.log(student.marks); // [50, 99]
```

As you can see, the students' grades were added to one array. Let's understand what happened:

When we copied the object `{ ...student }`, we got a copy of all properties. And for the array `[]`, as for an object or function, a reference is stored in the property. So, if we write our objects on paper, we'll have the following situation:

```
student: #1

anotherStudent: #2


#1 {

  name: 'John',

  marks: #arr1,

}


#arr1: []


#2 {

  name: 'John',

  marks: #arr1,

}
```

Thus, the array for grades was not cloned; we only copied the reference to the existing one. To make a "deep" copy, one can use the [`structuredClone`](https ://developer.mozilla.org/en-US/docs/Web/API/structuredClone) function:

```
const student = {

  name: 'John',

  marks: [],
```

```
};


const anotherStudent = structuredClone(student);


student.marks.push(50);

anotherStudent.marks.push(99);


console.log(student.marks); // [50]

console.log(anotherStudent.marks); // [99]
```

It is relatively new but already supported by all modern browsers.

Comparing Objects

Strict comparison `===` in JavaScript is done by values. If the values match, we get `true`, otherwise `false`. For objects, this means that we will be comparing references, not properties:

```
const person = { // #1

  name: 'John',

};


const friend = person; // #1

const buddy = { ...person }; // #2


console.log(person === friend); // true

// console.log(#1 === #1);


console.log(person === buddy); // false

// console.log(#1 === #2);
```

Destructuring Objects

Sometimes in a program, we only need individual properties of an object. For code reduction, it can be convenient to write their values into separate variables. For example:

```
const user = {
```

```
  name: 'John',
  lastName: 'Smith',
  age: 25,
  isMarried: false,
};

printPersonInfo(user); // 'John is 25'

function printPersonInfo(person) {
  console.log(`${person.name} is ${person.age}`);
}
```

To get the values, we have to constantly write `person.`. If there are more such places, using separate variables can simplify the code:

```
function printPersonInfo(person) {
  const name = person.name;
  const age = person.age;

  console.log(`${name} is ${age}`);
}
```

Of course, this method requires us to write much more code at the beginning. Therefore, in JavaScript, there is **destructuring assignment**, which allows declaring the necessary variables in one line:

```
function printPersonInfo(person) {
  const { name, age } = person;

  console.log(`${name} is ${age}`);
}
```

To the left of the = sign, we write the keyword for declaring variables. Next, in curly braces, we list the keys of the object that we need, separated by commas. After =, we write the object (or an expression that will result in an object). As a result, variables with the specified names (the same as the listed object keys) are created. So, we get the same result as in the previous example:

```
const { name, age } = person;
```

the same as:

```
const name = person.name;
```

```
const age = person.age;
```

If desired, we can change the names of the variables by adding them after `:`:

```
function printPersonInfo(person) {

  const { name: personName, age: personAge } = person;


  console.log(`${personName} is ${personAge}`);

}
```

We can also set default values for a variable if its value equals `undefined`:

```
function printPersonInfo(person) {

  const {

    name = 'Unknown',

    age: personAge = 0,

  } = person;


  console.log(`${name} is ${personAge}`);

}
```

In this case, if the `person` object does not have a `name` property (or it has a value of `undefined`), the `name` variable will receive the value `'Unknown'`. And the `personAge` variable will receive the value `person.age` or `0`.
And the last step is the function parameter destructuring:

```
function printPersonInfo({ name, age }) {

  console.log(`${name} is ${age}`);

}
```

Instead of the first function parameter `person`, we wrote `{}` in which we listed the desired keys. It works the same as:

```
function printPersonInfo(person) {

  const { name, age } = person;


  //…

}
```

Of course, if we do not pass arguments to the function, we will get an error:

```
function printPersonInfo({ name, age }) {
```

```
  console.log(`${name} is ${age}`);
}

printPersonInfo(); // Uncaught TypeError: Cannot destructure property 'name' of 'undefined'
```

Therefore, it's worth setting {}, as the default value for the first parameter:

```
function printPersonInfo({ name, age } = {}) {
  console.log(`${name} is ${age}`);
}

printPersonInfo(); // 'undefined is undefined'
```

And you can also set default values for variables:

```
function printPersonInfo({ name = 'Unknown', age = 0 } = {}) {
  console.log(`${name} is ${age}`);
}

printPersonInfo(); // 'Unknown is 0'
printPersonInfo({ name: 'Bob' }); // 'Bob is 0'
printPersonInfo({ age: 25 }); // 'Unknown is 25'
printPersonInfo({ name: 'Bob', age: 25 }); // 'Bob is 25'
```

Object Methods

"Objects can house values as well as functions, and we call such function **object methods.** Easing into the topic, let's consider a function that prints user information:

```
function printFullName(user) {
  console.log(`${user.firstName} ${user.lastName}`);
}

const firstUser = {
  firstName: 'John',
  lastName: 'Smith',
  age: 25,
```

```
  printInfo: printFullName,
};

printFullName(firstUser); // John Smith
```

In this example, we receive the object as a parameter and perform necessary actions. Where's the issue, you might ask? Well, theoretically, we could pass a value of another type — instead of an object — or an object that doesn't contain necessary properties, e.g.:

```
printFullName({}); // undefined undefined
```

We can do three things to counteract this issue, from worst to best…

1. [WORST] **Adding a property type check** with the `typeof` operator:

```
function printFullName(user) {
  if (typeof user.firstName !== 'string'
    || typeof user.lastName !== 'string'
  ) {
    return; // just finish the function
  }

  console.log(`${user.firstName} ${user.lastName}`);
}
```

2. [MEH] **Defining a method** *outside* **an object:**

```
function printFullName(user) {
  console.log(`${user.firstName} ${user.lastName}`);
}

const user = {
  firstName: 'John',
  lastName: 'Smith',
  age: 25,

  printInfo: printFullName,
};
```

We don't have to give the object as an argument though. It's enough to use the keyword `this`, which by default refers to the object we call the method ***on:***

```
function printFullName() {

  // Use `this` instead of `user`

  console.log(`${this.firstName} ${this.lastName}`);

}


const user = {

  firstName: 'John',

  lastName: 'Smith',

  age: 25,


  printInfo: printFullName,

};


user.printInfo(); // John Smith
```

If we don't call the function as an object method, `this` is undefined, hence:

```
printFullName(); // Uncaught TypeError: Cannot read properties of undefined (reading 'firstName')
```

3. [BEST] **Defining a method** *inside* **an object:**

```
const user = {

  firstName: 'John',

  lastName: 'Smith',

  age: 25,


  printInfo: function() {

    console.log(`${this.firstName} ${this.lastName}`);

  },

};
```

We can shorten this code with the object method syntax:

```
const user = {

  firstName: 'John',
```

```
  lastName: 'Smith',
  age: 25,

  printInfo() {
    console.log(`${this.firstName} ${this.lastName}`);
  },
};
```

A drawback of this approach is that we have to add all necessary methods to each object. In the coming lessons, we will solve this issue through inheritance.

💡 Such a combination of data with ***behavior*** underpins the now-dominant programming paradigm, **Object-Oriented Programming** (OOP).

Computed Properties (Getters)

While working with an object, we might need to compute some information based on the object's properties, like the user's full name:

```
const user = {
  firstName: 'John',
  lastName: 'Smith',
};

user.fullName = `${user.firstName} ${user.lastName}`;

console.log(user.fullName); // 'John Smith'
```

Now, even if we change the user's name, `fullName` remains as is. That's because we don't assign a new value to `fullName`:

```
user.firstName = 'Bob';

console.log(user.fullName); // 'John Smith'
```

We can solve this problem with a method:

```
const user = {
  firstName: 'John',
  lastName: 'Smith',
```

```
  getFullName() {
    return `${this.firstName} ${this.lastName}`
  }
};

console.log(user.getFullName()); // 'John Smith'

user.firstName = 'Bob';

console.log(user.getFullName()); // 'Bob Smith'
```

…but this approach forces us to rewrite all the code that uses our object, which is just a lot of unnecessary work. To keep the syntax unchanged, we can precede the object method's name with the keyword `get`, which we refer to as ***getter*** or a ***computed property:***

```
const user = {
  firstName: 'John',
  lastName: 'Smith',

  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },
};

console.log(user.fullName); // John Smith

user.firstName = 'Bob';

console.log(user.fullName); // Bob Smith
```

When we read the property `user.fullName`, the getter `fullName` is called and returns the computed value. Oh, and do note, we don't succeed the `fullName` with parentheses, which means the getter **doesn't have any parameters.**

Setters

By preceding the method's name with the `set` keyword, we can execute some logic each time a new value is assigned to an object property. Like so:

```
const user = {

  firstName: 'John',

  lastName: 'Smith',


  set fullName(value) {

    const spacePosition = value.indexOf(' '); // find the first space


    this.firstName = value.slice(0, spacePosition); // The name should be first

    this.lastName = value.slice(spacePosition); // The surname - second

  },

};


user.fullName = 'Mike Brown';


console.log(user.firstName); // 'Mike'

console.log(user.lastName); // 'Brown'
```

The setter must take exactly 1 parameter. It's particularly useful for validating new values, and importantly enough, we can combine setters with getters together:

```
const user = {

  firstName: 'John',

  lastName: 'Smith',


  get fullName() {

    return `${this.firstName} ${this.lastName}`;

  },


  set fullName(value) {

    if (typeof value !== 'string') {
```

```
      return; // invalid data type
    }

    if (!value.includes(' ')) {
      return; // invalid format
    }

    const spacePosition = value.indexOf(' ');

    this.firstName = value.slice(0, spacePosition);
    this.lastName = value.slice(spacePosition);
  },
};
```

If we add a setter without a getter, we'll get `undefined` while reading the value. Same issue will occur if we try to add a getter without a setter, and then attempt to assign some value:

```
const user = {
  firstName: 'John',
  lastName: 'Smith',

  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },
};

user.fullName = 'Bob Black'; // Uncaught TypeError: Cannot set property
```

Working with Arrays

Look around your home: bookshelves, wardrobes, kitchen drawers, fridge… all these places store homogeneous items, like books, clothes, kitchenware, and food. In JavaScript, we store such items — or ***data*** – in arrays.

Creating an Array

If you're asking yourself "wait a moment, haven't we covered arrays already?", indeed we have! And as you might remember, there are three ways to declare an array:

```
const daysOfWeek = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'];

const numbers = [2, 5, 3, 4, 5, 1, 3];

const words = ['John', 25, true]; // tuple


// this is an empty array, it currently has no elements

const friends = [];


// you can also create an array this way, but [] is used in practice

const marks = new Array(2, 5, 3, 4, 5, 1, 3);

const marks2 = Array(2, 5, 3, 4, 5, 1, 3);
```

We can also create an array out of a string with the `split` method, which — as the name indicates — splits a string into substrings based on a given delimiter. For example:

```
const str = 'one; two; three; four; five';


console.log(

  str.split(' '),  // ['one;', 'two;', 'three;', 'four;', 'five']

  str.split('; '),  // ['one', 'two', 'three', 'four', 'five']

  str.split('; ', 3),  // ['one', 'two', 'three']

  str.split('three'), // ['one; two; ', '; four; five']

);
```

Making an array out of individual string characters is an option, too:

```
const str = 'abc def';


console.log(

  str.split(''), // ['a', 'b', 'c', ' ', 'd', 'e', 'f']

  Array.from(str), // creates an array from string characters

  [...str], // spread operator copies characters from the string to a new array

);
```

Combining Array Elements into a String

To combine array elements into a single string, we use the `join` method. It takes a delimiter as an argument and returns said string:

```
const words = ['JavaScript', 'is', 'awesome'];


console.log(
  words.join(' '), // 'JavaScript is awesome'
  words.join(', '), // 'JavaScript, is, awesome'
  words.join('---'), // 'JavaScript---is---awesome'
);
```

Accessing Array Elements

We can access an array element by its index:

```
const daysOfWeek = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'];


console.log(
  daysOfWeek[0], // 'Mon'
  daysOfWeek[1], // 'Tue'
  daysOfWeek[6], // 'Sun'

  // if the index is too large, we get undefined
  daysOfWeek[10], // undefined

  // or negative
  daysOfWeek[-2], // undefined
);
```

`at` method is a great alternative. It works the same as `arr[index]` for positive indexes, but if we precede the index with a minus `-`, the method counts from the end. So the last element is `-1`, the second last — `-2`, and so on. For example:

```
console.log(
  daysOfWeek.at(-1), // 'Sun'
  daysOfWeek.at(-2), // 'Sat'
```

```
  daysOfWeek.at(1), // 'Tue'
);
```

Iterating Over Array Elements

To iterate over array elements, we can use one of the two loops:

```
const daysOfWeek = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday'];


for (let i = 0; i < daysOfWeek.length; i++) {
  console.log(daysOfWeek[i]);
}


for (const day of daysOfWeek) {
  console.log(day);
}
```

It's possible to iterate with methods, too, but we'll cover them later.

Callback functions

Callback is a function **passed as a parameter** to another function. Developers often use it while iterating arrays, handling asynchronous events — such as browser events, timers, server responses — and more. And since you're studying to become a developer…

Example of a Callback

Consider the `map` function, which takes an array of numbers and a callback. Consequently, it returns a new array with results of callback calls for each of the original array's elements:

```
function map(values, callback) {
  const results = [];


  for (const value of values) {
    results.push(
      callback(value)
    );
  }
```

```
  return results;
}


const numbers = [1, 2, 3, 4, 5];
const double = x => x * 2;
const square = x => x ** 2;


console.log(
  map(numbers, double), // [2, 4, 6, 8, 10]
  map(numbers, square), // [1, 4, 9, 16, 25]
);
```

Callbacks let us reuse the `map` function's logic for different arrays and functions to transform the input array's elements. Things get even simpler with arrow functions:

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


console.log(
  map(numbers, x => x + 100), // [101, 102, 103, 104, 105]
  map(numbers, x => x * 10), // [10, 20, 30, 40, 50]
  map(numbers, x => x ** 2), // [1, 4, 9, 16, 25]
);
```

Here, we create an anonymous function while calling the `map`. It's executed for each element, and outputs an array of results.

TIMERS

JavaScript lets us schedule a function call with the following methods:

- `setTimeout` calls a function **once** after a certain time passes
- `setInterval` calls a function **repeatedly** at fixed intervals
- `clearTimeout` cancels the scheduled action with `setTimeout`
- `clearInterval` cancels the scheduled action with `setInterval`

For example, the following code displays `Hello` every second for five seconds, and cancels all subsequent calls:

```
function greet() {
  console.log('Hello');
```

```
}


const timerId = setInterval(greet, 1000);


setTimeout(() => {
  clearInterval(timerId);
}, 5000);
```

Event handling

Callbacks are also used to handle events on page elements:

```
const element = document.querySelector('css_selector'); // find the element


element.addEventListener('click', (event) => {
  console.log(event.target); // the element on which the click occurred
});
```

To the `addEventListener` method, we pass the event's name and the function that will execute each time the event occurs on the selected element. We'll discuss event handling in detail later.

Array Iteration Methods

Working with arrays often means executing the same action on each element. Boring? Indeed. Necessary? Yes. Laborious? Not necessarily! You see, JavaScript offers a few methods that make iterating arrays a breeze, which we'll now review.

forEach

`.forEach(callback)` executes the `callback` once on each of the array's elements and returns `undefined`. The `callback` receives three arguments:

- Current element of the array
- Index of the current element
- Reference to the array

Arguments remain identical for `map`, `filter`, `find`, `findIndex`, `some`, and `every` methods. Here's an example:

```
const callback = (element, index, arr) => {
  console.log(element, index, arr);
};


const result = ['a', 'b', 'c'].forEach(callback);
```

```
// 'a' 0 ['a', 'b', 'c']

// 'b' 1 ['a', 'b', 'c']

// 'c' 2 ['a', 'b', 'c']


console.log(result); // undefined
```

MAP

`.map(callback)` executes the `callback` once on each of the array's elements and returns a new array with the resulting elements. Here's an example:

```
const callback = (element, index, arr) => (element + index);


console.log(

  [10, 20, 30].map(callback), // [10, 21, 32]

  [10, 20, 30].map(el => el + 5), // [15, 25, 35]

);
```

FILTER

`.filter(callback)` executes the `callback` once on each of the array's elements and returns a new array with elements, for which said `callback` **returned a truthy value.** In other words — a value which converts to a boolean `true`. Here's an example:

```
const callback = (x, i, arr) => x > arr.length;


console.log(

  [1, 4, 7].filter(callback), // [4, 7]

  [1, 4, 7].filter(x => (x % 2 === 0)), // [4] - only even numbers

);
```

FIND AND FINDINDEX

Two almost identical methods with a slight tweak. **`.find(callback)`** returns the first element, for which the `callback` returns a truthy value, while **`.findIndex(callback)`** — this element's index. Here's an example:

```
const callback = (el, i, arr) => el > arr.length;


console.log(

  [1, 4, 7].find(callback), // 4

  [1, 4, 7].findIndex(callback), // 1
```

```
  [1, 4, 7].find(el => el > 10), // undefined

  [1, 4, 7].findIndex(el => el > 10), // -1

);
```

EVERY AND SOME

.every(callback) returns `false` if the `callback` returns a falsy value for any of the array's elements, otherwise — `true`. We use it to check if an entire array meets some criteria.

💡 A falsy value is the opposite of the truthy one, as in: it's `false` in boolean terms.

.some(callback) acts in reverse. It returns `true` if the `callback` returns a truthy value for any element of the array (if ***any*** element meets some criteria), otherwise — `false`.

Here's an example:

```
const callback = (el, i, arr) => el > arr.length;


console.log(

  [1, 4, 7].every(callback), // false

  [1, 4, 7].some(callback), // true


  [1, 4, 7].every(el => el > 0), // true

  [1, 4, 7].some(el => el > 0), // true


  [1, 4, 7].every(el => el > 10), // false

  [1, 4, 7].some(el => el > 10), // false


  [].every(el => el


 > 10), // true

  [].some(el => el > 10), // false

);
```

`REDUCE'

.reduce(callback, startValue) takes two arguments, `callback` — which is executed once for each element and returns the last call's results — and a value of any type. The `callback`, on the other hand, receives four arguments:

- Result of the previous `callback` call, or the "starting" value for the first call
- Current element of the array
- Current element's index
- Reference to the array

Here's an example:

```
// calculate the sum of numbers

[2, 5, 3, 1, 7].reduce(

  (sum, x) => sum + x,

  0,

);



// count the number of occurrences of characters

'abrakadabra'.split('')

  .reduce((counter, char) => ({

    counter,

    [char]: (counter[char] || 0) + 1,

  }), {});
```

SORT

.sort(callback) sorts the array's elements by the `callback` results and returns a reference to the array. The callback function receives 2 arguments representing elements from the array being compared at any given moment. The internal sorting mechanism uses the callback's return value to determine the order of these elements:

- Positive result indicates the first element should come after the second in the sorted array.
- Negative result means the second element should come after the first.
- `0` implies the relative order of the two elements should remain unchanged.

```
// Sorting numbers

const numbers = [3, 12, 2, 11];



numbers.sort((a, b) => a - b);

console.log(numbers); // [2, 3, 11, 12]



// Shuffling in random order

numbers.sort(() => Math.random() - 0.5)
```

```
// Sorting strings caseinsensitive considering special characters
const words = ['apple', 'Zebra', 'car'];

console.log(
  words.sort(), // ['Zebra', 'apple', 'car']
  words.sort(
    (word1, word2) => word1.localeCompare(word2)
  ), // ['apple', 'car', 'Zebra']
);

// Sorting objects by a specific property
users.sort(
  (user1, user2) => user1.age - user2.age, // by numeric value
);

users.sort(
  (user1, user2) => user1.name.localeCompare(user2.name) // by text field
);
```

Closure and Variable Scope

In this lesson, we will discuss variable visibility in JavaScript in more detail, as it forms the basis for restricting access to variables within certain parts of code, as well as the ability to interact with "external" variables.

Variable Scope

In JavaScript, there are three main types of variable scopes: **global**, **block**, and **functional**.

- **Global Scope:** Contains variables declared outside of any blocks or functions. They are accessible everywhere in your code.
- **Block Scope:** Contains variables declared within blocks `{}`. For example, variables declared using the `let` or `const` keywords inside `if`, `for`, `while` loops, etc., have block scope and are only accessible within that block. The same applies to functions declared inside a block. According to the standard ('use strict'), they are also only accessible within the block where they are declared.
- `let a = 1; // Global scope`
-

```
if (true) {
  let b = 2; // Block scope

  if (true) {
    let c = 3; // Block scope of nested block
  }

  console.log(a); // Accessible because `a` is global
  console.log(b); // Accessible because `b` is within the same block
  // console.log(c); // Not accessible, `c` is declared in another block
}
```

- **Functional Scope:** Variables and functions declared within functions (but not within nested blocks) are only accessible within that function.

Block scope was introduced in JavaScript along with the appearance of the `let` and `const` keywords. Before that, only **global** and **functional** scopes existed, and variables were declared using the `var` keyword.

Primarily for interviews or working with legacy code, let's list the main differences between a variable declared with the `var` keyword compared to the modern `let`:

- The visibility of a `var` variable is not limited to the block where it is declared (only to the function);
- (**hoisting**) A `var` variable can be used before declaration (at the beginning of the function or program). Before the first assignment, it has a value of `undefined`.

Closure

Variable visibility is particularly important for functions that use them. Let's consider a simple example:

```
let sum = 0;


function add(x) {
  sum += x;
  console.log(sum);
}


console.log(sum); // 0


add(10); // 10
add(10); // 20
```

As you can see, the `add` function has access to the `sum` variable and can not only read but also modify its value.

Closure is a function along with all the external variables that it has access to.

Of course, in this example, you could argue that the `sum` variable is declared outside any block, so it is global, and we could directly modify it without calling the `add` function.
So let's change our code a bit:

```
'use strict';


if (true) {
  let sum = 0;


  function add(x) {
    sum += x;
    console.log(sum);
  }


  add(10); // 10
  add(10); // 20
}


add(10); // Uncaught ReferenceError: add is not defined
```

Now neither the variable `sum` nor the function `add` are accessible outside the block. (If you remove `'use strict'`, the function will be accessible outside the block).
If we declare the `add` function outside the block, it won't see the `sum` variable:

```
'use strict';


if (true) {
  let sum = 0;
}


function add(x) {
  sum += x;
  console.log(sum);
}
```

```
add(10); // Uncaught ReferenceError: sum is not defined
```

To control the sum from outside the block, I suggest declaring the `add` variable outside the block and creating a function and assigning a reference to it within the block:

```
'use strict';


let add;


if (true) {
  let sum = 0;


  function internalAdd(x) {
    sum += x;
    console.log(sum);
  }


  add = internalAdd;
}


add(10); // 10
add(10); // 20
```

Now we can control the sum using the `add` function, but we cannot directly modify or read the `sum` variable.
For convenience, you can create a separate function to get the current value:

```
'use strict';


let add;
let getSum;


if (true) {
  let sum = 0;
```

```
  add = function(x) {
    sum += x;
  };

  getSum = function() {
    return sum;
  };
}

console.log(getSum()); //0

add(10);
add(10);
```

Instead of declaring functions, we now use functional expressions (creating a function within an expression during assignment).

In this case, we can say that the `add` and `getSum` functions have access to the same external variables (they have a common closure).
Of course, the same can be written using arrow functions, and everything will work the same way.

```
'use strict';

let add;
let getSum;

if (true) {
  let sum = 0;

  add = x => sum += x;
  getSum = () => sum;
}
```

```
console.log(getSum()); //0


add(10);

add(10);
```

Loop Variable i and Function Creation in a Loop

Now let's consider creating functions in a loop. For example, to populate an array of players with functions that print their ordinal number:

```
const team = [];

let currentNo = 1;


while (currentNo <= 3) {
  const player = () => {
    console.log(currentNo);
  };


  team.push(player);
  currentNo++;
}


team[0](); //4

team[1](); //4

team[2](); //4
```

Since the `currentNo` variable is declared globally, after the loop, it has a value of 4, which we can see. To ensure each function can print its number, we need to create a new variable on each loop iteration:

```
const team = [];

let currentNo = 1;


while (currentNo <= 3) {
  const playerNo = currentNo;
```

```
  const player = () => {
    console.log(playerNo);
  };

  team.push(player);
  currentNo++;
}

team[0](); //1
team[1](); //2
team[2](); //3
```

Then each function will have its own variable in the closure with its unique value set on the current loop iteration.

This code can be significantly shortened using a `for` loop:

```
const team = [];

for (let playerNo = 1; playerNo <= 3; playerNo++) {
  const player = () => {
    console.log(playerNo);
  };

  team.push(player);
}

team[0](); //1
team[1](); //2
team[2](); //3
```

Function Returning Another Function

In the previous example, we created functions in a loop, which allowed us to create our own variable for each function added on each iteration. Now let's consider an example where we create and return a function from another function. As an example, we can mention a factory that produces recorders. A

recorder has a function - to remember words. And, of course, each recorder has internal memory, so between launches, it does not lose what was recorded before.

Here's a simple example of code implementing this logic:

```
function createRecorder(name) {
  let words = []; // Internal memory

  // Recorder function
  const recorder = (word) => {
    words.push(word);
  };

  // Listen to the recording
  recorder.play = () => {
    console.log(`${name}: ${words.join(' ')}`);
  };

  // Clear memory
  recorder.clear = () => {
    words = [];
  };

  return recorder;
}

const captainRecorder = createRecorder('Jim');

captainRecorder('Hi,');
captainRecorder('I');
captainRecorder('am');
captainRecorder.play(); // 'Jim: Hi, I am'
```

```
captainRecorder('your');

captainRecorder('captain');

captainRecorder.play(); // 'Jim: Hi, I am your captain'


const commanderRecorder = createRecorder('Spock');


commanderRecorder('Live');

commanderRecorder('long');

commanderRecorder.play(); // 'Spock: Live long'


commanderRecorder.clear();


commanderRecorder('and');

commanderRecorder('prosper');

commanderRecorder.play(); // 'Spock: and prosper'
```

In this example, when the `createRecorder` function is run, we create three new functions:

- `recorder`, which remembers words;
- the `recorder.play` method, for listening to the recording;
- the `recorder.clear` method, for clearing memory.

They use a common `words` array in the closure.
When `createRecorder` is run again, three more functions are created, and a new closure with a new `words` array is created.

PROTOTYPE INHERITANCE

Property inheritance lets an object (so-called ***child*** object) use a property from another object (***parent*** object). Let's consider an example:

```
const father = {

  firstName: 'John',

  lastName: 'Smith'

};


const child = {

  firstName: 'Alice',
```

```
};


console.log(child.lastName); // undefined
```

Here, the `child` object doesn't have the `lastName` property, which results in `undefined`. We can fix this by establishing inheritance between the `child` and the `father`:

```
const father = {
  firstName: 'John',
  lastName: 'Smith'
};


const child = {
  firstName: 'Alice',
};


// establishing inheritance between `child` and `father`
Object.setPrototypeOf(child, father);


console.log(child.lastName); // 'Smith'
```

This time around, when we try to read the `lastName` property from `child`, JavaScript looks for it in the prototype — the `father` object — and returns the value `Smith`. (We can denote this as `child -> father`). Inheritance works across multiple levels, too:

```
const grandFather = {
  firstName: 'Bob',
  lastName: 'Smith'
};


const father = {
  firstName: 'John',
};


const child = {
```

```
  firstName: 'Alice',
};

// father -> grandFather
Object.setPrototypeOf(father, grandFather);

// child -> father -> grandFather
Object.setPrototypeOf(child, father);

console.log(child.lastName); // 'Smith'
```

Here, we link `child` to `father` and then to `grandFather`, creating an inheritance chain, which lets us look for a property in `child`, then `father`, and finally `grandFather`. But tracking property ownership across multiple objects can get… confusing. That's where `hasOwnProperty` helps:

```
const grandFather = {
  firstName: 'Bob',
  lastName: 'Smith'
};

const father = {
  firstName: 'John',
};

const child = {
  firstName: 'Alice',
};

Object.setPrototypeOf(father, grandFather);
Object.setPrototypeOf(child, father);

console.log(child.hasOwnProperty('lastName')); //false
```

As you can see, the method states that `child` does not have its own `lastName` property. How come, if we haven't declared this method in the first place? Turns out that, by default, every object has a hidden service property `[[Prototype]]`, which facilitates the inheritance mechanism.
Circling back to the example, `Object.setPrototypeOf(child, father)` writes a reference to `father` into this property of `child`. But for the `grandFather` object, we didn't set anything explicitly. So, it retained the default value — reference to an object (obtainable with `Object.prototype`).
If we break the inheritance chain, we'll get an error when calling the `hasOwnProperty` method:

```
const child = {
  firstName: 'Alice',
};

// Breaking inheritance from Object.prototype
Object.setPrototypeOf(child, null);

console.log(child.hasOwnProperty('lastName'));
// Uncaught TypeError: child.hasOwnProperty is not a function
```

That's why we use `Object.hasOwn(child, 'lastName')` to check for a certain property. Though, we use the prototype only when trying to **read** a property, and only if the property **isn't** in the object itself:

```
const father = {
  firstName: 'John',
  lastName: 'Smith'
};

const child = {
  firstName: 'Alice',
};

Object.setPrototypeOf(child, father);

// This property exists in the object
console.log(child.firstName); // 'Alice'

// Deletion from the prototype does not occur
```

```
delete child.lastName;

console.log(child.lastName); // 'Smith'


// The new value will be written into the object itself, not the prototype

child.lastName = 'Black';

console.log(child.lastName); // 'Black'

console.log(father.lastName); // 'Smith'
```

GETPROTOTYPEOF

We can obtain the prototype of an object with the `Object.getPrototypeOf(child)` method:

```
const father = {

  firstName: 'John',

  lastName: 'Smith'

};


const child = {

  firstName: 'Alice'

};


console.log(Object.getPrototypeOf(child) === Object.prototype); // true


Object.setPrototypeOf(child, father);


console.log(Object.getPrototypeOf(child) === father); // true
```

SETPROTOTYPEOF

The `Object.setPrototypeOf` method lets us set `null` or a reference as a prototype (object, array, function, etc.), but doesn't allow primitive values:

```
const child = {

  firstName: 'Alice'

};


Object.setPrototypeOf(child, null);
```

```
console.log(child.length); // undefined
```

```
Object.setPrototypeOf(child, [1, 2, 3]);

console.log(child.length); // 3
```

```
Object.setPrototypeOf(child, 'Hello');

// Uncaught TypeError: Object prototype may only be an Object or null: 'Hello'
```

THIS IN INHERITED METHODS

When calling an inherited method — getter or setter — `this` refers to the object preceding the dot `.`, rather than the prototype. For example:

```
const father = {
  firstName: 'John',
  lastName: 'Smith',
  age: 40,

  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },

  celebrateBirthday() {
    this.age++;
  },
};

const child = {
  firstName: 'Alice',
  age: 10,
};
```

```
Object.setPrototypeOf(child, father);


console.log(child.fullName); // 'Alice Smith'


child.celebrateBirthday();

console.log(father.age); // 40

console.log(child.age); // 11
```

__PROTO__

We manage the inheritance of existing objects with `getPrototypeOf` and ``setPrototypeOfmethods – but that wasn't always the case. Before, we used getter and setter`proto`, declared in Object.prototype`:

```
const father = {

  firstName: 'John',

  lastName: 'Smith'

};


const child = {

  firstName: 'Alice'

};


child.__proto__ = father;

console.log(child.__proto__ === father); // true
```

They work the same as the new methods, with slight differences:

```
const father = {

  firstName: 'John',

  lastName: 'Smith',

};


const child = {

  firstName: 'Alice',
```

```
  // Can be added when creating an object
  __proto__: father
};

// this will not work without an error
child.__proto__ = 'Hello';
console.log(child.lastName); // 'Smith'

child.__proto__ = null;
// now access to the setter is lost
// it's now a regular property of the object itself
// not a way to set inheritance
child.__proto__ = father;

console.log(Object.getPrototypeOf(child)); // null
console.log(Object.hasOwn(child, '__proto__')); // true
```

Constructor Functions

Developers often need to create many homogeneous objects, i.e., with identical properties and methods, which seems… laborious. Fortunately, there's a special kind of function we can use for this exact purpose, a ***constructor***, declared with the `new` keyword. The constructor's name should be capitalized and describe the created objects, e.g.:

```
function User(name, role = 'user') {
  this.name = name;
  this.role = role;
}

const bob = new User('Bob'); // { name: 'Bob', role: 'user' }
const alice = new User('Alice', 'admin'); // { name: 'Alice', role: 'admin' }
```

We can access the newly created object with `this` keyword. If the constructor doesn't explicitly return a reference to another object, then a reference to `this` is returned at the end.

💡 Arrow functions are incompatible with `new`.

The prototype Property

Every non-arrow function has a `prototype` property, which by default contains a reference to an object `{ constructor: #f }` (where `#f` is a reference to the function itself). Objects created by invoking the function with `new` inherit from the referenced object:

```
function User(name, role = 'user') {

  // this = {};

  // Object.setPrototypeOf(this, User.prototype)



  this.name = name;

  this.role = role;



  // return this;

}



const bob = new User('Bob');



console.log(Object.getPrototypeOf(bob) === User.prototype); // true
```

If needed, we can assign a reference to a new object to `User.prototype`, or add new methods and properties to it. They'll be available to all objects created by the `User` constructor:

```
function User(name) {

  this.name = name;

  this.role = 'admin';

}



User.prototype = {

  sayHi() {

    console.log(`Hello, ${this.name}!`);

  }

};



User.prototype.sayBye = function() {

  console.log(`Goodbye, ${this.name}!`);
```

```
};


const john = new User('John');

john.sayHi(); // Hello, John!

john.sayBye(); // Goodbye, John!
```

In practice, only methods shared by objects are added to the `prototype`, while properties are added to individual objects. That's because the data in similar objects can be different (the keys match, values don't), but the behavior should be the same.

Prototypes of Standard Objects

Standard objects, such as arrays and functions, are constructed using built-in constructors like Object, Array, and Function. This means arrays inherit their features from the **`Array.prototype`** object, which houses all array methods. Similarly, `Function.prototype` holds methods like **`call`**, **`apply`**, and **`bind`** for functions.
Primitive types, such as strings, numbers, and booleans also possess constructors — `String`, `Number`, and `Boolean`, respectively. When a primitive value is followed by a dot `.`, JavaScript creates a hidden object of the corresponding type, inheriting necessary methods from `Number.prototype` or `String.prototype`.
Though, using explicit `new Number(5)` is discouraged, since it results in an object, rather than a number — like here:

```
console.log(typeof new Number(5)); // 'object'
```

All Possible Values of `this`

During tech interviews (a part of recruitment process in IT), developers are often asked about the possible values of **`this`**. The answer is context-dependent:

- Outside of a function, `this` refers to the global object (`window` in browsers, `global` in Node.js).
- Arrow functions always inherit `this` from the enclosing scope.
- When a function is called without an object reference (`f()`), `this` is undefined.
- In an object method, `this` refers to the object preceding the dot `.`.
- When using `new f()`, `this` refers to a new object inheriting from `f.prototype`.
- Using `f.call(expectedThis, arg1, arg2)` sets `this` to `expectedThis`, passing `arg1` and `arg2` as function parameters.
- With `f.apply(expectedThis, [arg1, arg2])`, `this` is set to `expectedThis`, and arguments must be passed as an array.
- The `.bind(expectedThis, arg1, arg2)` method creates a new function where this is permanently set to expectedThis, along with additional arguments.

```
function f(a, b) {

  console.log(this, a, b);

}


const user = { name: 'John' };
```

```
f.call(user, 1, 2); // { name: 'John' } 1 2

f.apply(user, [10, 20]); // { name: 'John' } 10 20


const wrapper = f.bind(user, [100, 200]);

wrapper(); // { name: 'John' } 100 200
```

💡 `this` behaves like a constant.

CLASSES

In our last lesson, we created uniform objects with constructor functions, adding properties to the constructor itself, and methods — to the prototype. Doing so guaranteed that all objects created by one constructor have the same methods, and saved us from duplicating code:

```
function User(name) {
  this.name = name;
  this.age = 0;
}


User.prototype.printInfo = function() {
  console.log(`${this.name} is ${this.age}`);
};


User.prototype.celebrateBirthday = function() {
  this.age++;
};


const john = new User('John');
```

Classes take this idea a step further, as they're **templates** for creating uniform objects, declared with the `class` keyword. They vastly simplify the constructor and class method syntax:

```
class User {
  constructor(name) {
    this.name = name;
    this.age = 0;
  }
```

```
  printInfo() {
    console.log(`${this.name} is ${this.age}`);
  }

  celebrateBirthday() {
    this.age++;
  }
}

const john = new User('John');
```

Under the hood, all is as before, i.e., references are added to `constructor`, and methods — to `User.prototype`. The only difference is that `User` **cannot be executed without `new`.**

Public and Private Properties

The `class` syntax provides more flexibility in adding properties. If the initial value of a property doesn't depend on constructor parameters, it can be written separately:

```
class User {
  age = 0;

  constructor(name) {
    this.name = name;
    // this.age = 0;
  }
}

const john = new User('John');
console.log(john.age); // 0
```

To make a property private, i.e., prohibit access from outside the class, we prefix it with `#`:

```
class User {
  #age = 0;
```

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

  getAge() {
    return this.#age;
  }
}

const john = new User('John');
console.log(john.getAge()); // 0
console.log(john.#age); // Uncaught SyntaxError: Private field '#age' must be declared in an enclosing class
```

Getters and Setters

Getters and setters are also added to `User.prototype`:

```
class User {
  #age = 0;

  constructor(name) {
    this.name = name;
  }

  get age() {
    return this.#age;
  }

  set age(newAge) {
    if (newAge < this.#age) {
      return;
```

```
    }

    this.#age = newAge;
  }
}

const john = new User('John');

john.age = 10;
console.log(john.age); // 10

john.age = 5;
console.log(john.age); // 10
```

Static Properties and Methods

Suppose all users in the system have a certain role — `admin`, `user`, and `guest`. When creating a user, we can pass the initial role to the constructor:

```
class User {
  constructor(name, role = 'guest') {
    this.name = name;
    this.role = role;
  }
}

const guest = new User('Bob');
const admin = new User('Alice', 'admin');
const user1 = new User('John', 'use');

console.log(guest.role); // 'guest'
console.log(admin.role); // 'admin'
console.log(user1.role); // 'use'
```

Ooops, someone made a typo, and `user1` now has an unknown role of `use` instead of the intended `user`. Fortunately, we can safeguard our code against such typos and declare user roles as **class constants** (since they remain, well… constant):

```
class User {

  constructor(name, role = User.ROLE_GUEST) {

    this.name = name;

    this.role = role;

  }

}


User.ROLE_ADMIN = 'admin';

User.ROLE_USER = 'user';

User.ROLE_GUEST = 'guest';


const guest = new User('Bob');

const admin = new User('Alice', User.ROLE_ADMIN);

const user1 = new User('John', User.ROLE_USER);
```

Much more orderly now, isn't it? But we can do even more with the `static` keyword, which lets us add properties to the class itself, rather than objects created from the class:

```
class User {

  static ROLE_ADMIN = 'admin';

  static ROLE_USER = 'user';

  static ROLE_GUEST = 'guest';


  constructor(name, role = User.ROLE_GUEST) {

    this.name = name;

    this.role = role;

  }

}


const guest = new User('Bob');
```

```
const admin = new User('Alice', User.ROLE_ADMIN);

const user1 = new User('John', User.ROLE_USER);
```

Static parameters aren't limited to constants only, but can also be properties and methods — private and public — for a variety of purposes. Like counting statistics across all class objects:

```
class User {

  static #instances = [];


  static getCount() {

    return this.#instances.length;

  }


  constructor(name) {

    this.name = name;


    User.#instances.push(this);

  }

}


console.log(User.getCount()); // 0


const bob = new User('Bob');

const alice = new User('Alice');


console.log(User.getCount()); // 2

console.log(User.#instances);  // Uncaught SyntaxError: Private field '#instances' must be declared in an enclosing class
```

Note, in the static method `getCount`, `this` refers to the `User` class because it is before the dot in the call `User.getCount()`.

Alternative Constructor

Sometimes it's convenient to create objects with predefined properties — e.g., users with roles — but at the same time, keep the constructor for creating regular objects. In such cases, we can use a static method that calls the constructor with specified arguments:

```
class User {

  static ROLE_ADMIN = 'admin';

  static ROLE_USER = 'user';

  static ROLE GUEST = 'guest';


  static createAdmin(name) {

    return new User(name, User.ROLE_ADMIN);

  }


  constructor(name, role = User.ROLE_GUEST) {

    this.name = name;

    this.role = role;

  }

}


const guest = new User('Bob');

const user1 = new User('John', User.ROLE_USER);

const admin = User.createAdmin('Alice');
```

Class Inheritance

Class inheritance is a mechanism that allows all descendant objects to inherit the base class' properties and use its methods. For example:

```
class User {

  constructor(name) {

    this.name = name;

  }


  getInfo() {
```

```
    return `My name is ${this.name}`;
  }
}

class Admin extends User {
  role = 'admin';
}

const admin = new Admin('John');

console.log(admin); // Admin { name: 'John', role: 'admin' }
console.log(admin.getInfo()); // 'My name is John'
```

The super Keyword

If the child class has its own constructor, it must call the base class constructor with the `super` keyword before using `this`:

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

  getInfo() {
    return `My name is ${this.name}`;
  }
}

class Admin extends User {
  role = 'admin';

  constructor(name, privileges = []) {
    super(name);
```

```
    this.privileges = privileges;
  }
}

const admin = new Admin('John');

console.log(admin); // Admin { name: 'John', role: 'admin', privileges: [] }
console.log(admin.getInfo()); // 'My name is John'
```

`super` also lets child classes use methods from their parent class to improve or add new features, among other things:

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

  getInfo() {
    return `My name is ${this.name}`;
  }
}

class Admin extends User {
  role = 'admin';

  getInfo() {
    return `${super.getInfo()} and I am an admin.`;
  }
}

const admin = new Admin('John');
```

```
console.log(admin); // Admin { name: 'John', role: 'admin' }

console.log(admin.getInfo()); // 'My name is John and I am an admin.'
```

Inheritance of Static and Private Methods and Properties

When the `Admin` class inherits from the `User class`, it creates links between them:

- For instances: `admin` links to `Admin.prototype`, which links to `User.prototype`.
- For the classes themselves: `Admin` links to `User`.

This setup lets the `Admin` class access all the static features (like properties and methods) of the `User` class, **with the exception of **private — both regular and static. They can't be used directly in descendant classes.

The `instanceof` Operator

The `instanceof` operator checks whether the `prototype` property of the specified class or constructor is in the object's prototype inheritance chain. Which, in simpler terms, means we can see whether an object belongs to a given class:

```
class User {

  constructor(name, age) {

    this.name = name;

    this.age = age;

  }

}



class Admin extends User {

  role = 'admin';

}



class Student {

  constructor(name, age) {

    this.name = name;

    this.age = age;

  }

}



const admin = new Admin('Mike', 30);
```

```
console.log(admin instanceof Admin); // true, `admin -> Admin.prototype`

console.log(admin instanceof User); // true, `admin -> Admin.prototype -> User.prototype`

console.log(admin  instanceof  Object);  // true,  `admin  ->  Admin.prototype  ->  User.prototype  ->
Object.prototype`

console.log(admin instanceof Student); // false
```