LEARN JAVASCRIPT

BEGINNERS EDITION

A Complete Beginner's Guide to Learn JavaScript

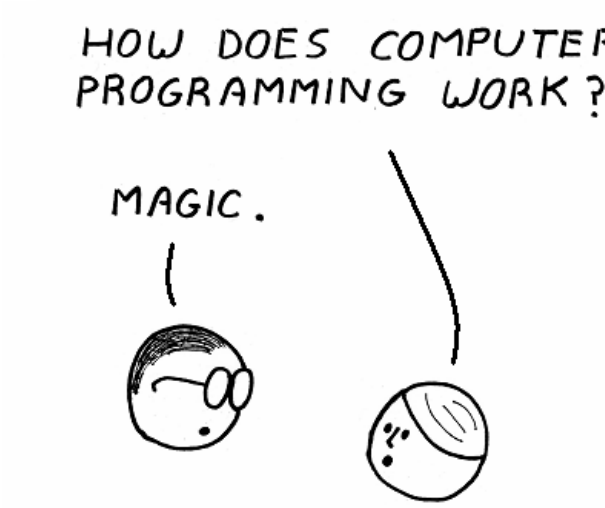
Suman Kunwar

Table of Contents

Introduction

Computers are common in today's world, as they are able to perform a wide variety of tasks quickly and accurately. They are used in many different industries, such as business, healthcare, education, and entertainment, and have become an essential part of daily life for many people. Besides this, they are also used to perform complex scientific and mathematical calculations, to store and process large amounts of data, and to communicate with people around the world.

Programming involves creating a set of instructions, called a program, for a computer to follow. It can be tedious and frustrating at times because computers are very precise and need specific instructions in order to complete tasks.



Programming languages are artificial languages used to give instructions to computers. They are used in most programming tasks and are based on the way humans communicate with each other. Like human languages, programming languages allow words and phrases to be combined to express new concepts. It is interesting to note that the most effective way to communicate with computers involves using a language that is similar to human language.

In the past, the primary way to interact with computers was through language-based interfaces like BASIC and DOS prompts. These have been largely replaced by visual interfaces, which are easier to learn but offer less flexibility. However, computer languages like *JavaScript* are still in use and can be found in modern web browsers and on most devices.

JavaScript (*JS for short*) is the programming language that is used to create dynamic interaction while developing webpages, games, applications, and even servers. JavaScript started at Netscape, a web browser developed in the 1990s, and is today one of the most famous and used programming languages.

Initially, it was created for making webpages alive and was able to run on a browser only. Now, it runs on any device that supports the JavaScript engine. Standard objects such as Array, Date, and Math are available in JavaScript, as well as operators, control structures, and statements. *Client-side JavaScript* and *Server-side JavaScript* are the extended versions of Core JavaScript.

- *Client-side JavaScript* enables the enhancement and manipulation of web pages and client browsers. Responses to user events such as mouse clicks, form input, and page navigation are some of its examples.
- *Server-side JavaScript* enables access to servers, databases, and file systems.

JavaScript is an interpreted language. While running Javascript an interpreter interprets each line and runs it. The modern browser uses Just In Time (JIT) technology for compilation, which compiles JavaScript into executable bytecode.

> "LiveScript" was the initial name given to JavaScript.

This book is divided into three main parts. The first 14 chapters cover the JavaScript language. The following three chapters discuss how JavaScript is used to program web browsers. The final two chapters are miscellaneous, and exercises. Various important topics and cases related to JavaScript programming are described in the Miscellaneous chapter, which is followed exercises.

Code, and what to do with it

Code is the written instructions that make up a program. Many chapters in this book contain a lot of code, and it is important to read and write code as part of learning how to program. You should not just quickly scan the examples - read them carefully and try to understand them. This may be difficult at first, but with practice, you will improve. The same goes for the exercises - make sure you actually try to write a solution before assuming you understand them. It is also helpful to try running your solutions to the exercises in a JavaScript interpreter, as this will allow you to see if your code is working correctly and may encourage you to experiment and go beyond the exercises.

Typographic conventions

In this book, text written in a monospaced font represents elements of a program. This can be a self-contained fragment or a reference to part of a nearby program. Programs, like the one shown below, are written in this way:

```
const numbers = [45, 4, 9, 16, 25];
let txt = "";
for (let x in numbers) {
  txt += numbers[x];
}
```

Sometimes, the expected output of a program is written after it, preceded by two slashes with a *Result*, like this:

```
console.log(txt);

// Result: txt = '45491625'
```

Good Luck! 🍀

Basics

In this first chapter, we'll learn the basics of programming and the Javascript language.

Programming means writing code. A book is made up of chapters, paragraphs, sentences, phrases, words, and finally punctuation and letters, likewise a program can be broken down into smaller and smaller components. For now, the most important is a statement. A statement is analogous to a sentence in a book. On its own, it has structure and purpose, but without the context of the other statements around it, it isn't that meaningful.

A statement is more casually (and commonly) known as a *line of code*. That's because statements tend to be written on individual lines. As such, programs are read from top to bottom, left to right. You might be wondering what code (also called source code) is. That happens to be a broad term which can refer to the whole of the program or the smallest part. Therefore, a line of code is simply a line of your program.

Here is a simple example:

```
let hello = "Hello";
let world = "World";

// Message equals "Hello World"
let message = hello + " " + world;
```

This code can be executed by another program called an *interpreter* that will read the code, and execute all the statements in the right order.

Comments

Comments are statements that will not be executed by the interpreter, comments are used to mark annotations for other programmers or small descriptions of what code does, thus making it easier for others to understand what your code does.

In JavaScript, comments can be written in 2 different ways:

- *Single-line comments*: It starts with two forward slashes (`//`) and continue until the end of the line. Anything following the slashes is ignored by the JavaScript interpreter. For example :

```
// This is a comment, it will be ignored by the interpreter
let a = "this is a variable defined in a statement";
```

- Multi-line comments: It starts with a forward slash and an asterisk (`/*`) and end with an asterisk and a forward slash (`*/`). Anything between the opening and closing markers is ignored by the JavaScript interpreter. For example:

```
/*
This is a multi-line comment,
it will be ignored by the interpreter
*/
let a = "this is a variable defined in a statement";
```

Variables

The first step towards really understanding programming is looking back at algebra. If you remember it from school, algebra starts with writing terms such as the following.

```
3 + 5 = 8
```

You start performing calculations when you introduce an unknown, for example, x below:

```
3 + x = 8
```

Shifting those around you can determine x:

```
x = 8 - 3
-> x = 5
```

When you introduce more than one you make your terms more flexible - you are using variables:

```
x + y = 8
```

You can change the values of x and y and the formula can still be true:

```
x = 4
y = 4
```

or

```
x = 3
y = 5
```

The same is true for programming languages. In programming, variables are containers for values that change. Variables can hold all kinds of values and also the results of computations. Variables have a name and a value separated by an equals sign (=). Variable names can be any letter or word but bear in mind that there are restrictions from language to language of what you can use, as some words are reserved for other functionality.

Let's check out how it works in Javascript, The following code defines two variables, computes the result of adding the two, and defines this result as a value of a third variable.

```
let x = 5;
let y = 6;
let result = x + y;
```

Exercise

Define a variable `x` equal to 20.

```
let x =
```

ES6 Version

ECMAScript 2015 or ES2015 also known as E6 is a significant update to the JavaScript programming language since 2009. In ES6 we have three ways of declaring variables.

```
var x = 5;
const y = 'Test';
let z = true;
```

The types of declaration depend upon the scope. Unlike the `var` keyword, which defines a variable globally or locally to an entire function regardless of block scope, `let` allows you to declare variables that are limited in scope to the block, statement, or expression in which they are used. For example.

```
function varTest(){
    var x=1;
    if(true){
        var x=2; // same variable
        console.log(x); //2
    }
    console.log(x); //2
}

function letTest(){
    let x=1;
    if(true){
        let x=2;
        console.log(x); // 2
    }
    console.log(x); // 1
}
```

`const` variables are immutable - they are not allowed to be re-assigned.

```
const x = "hi!";
x = "bye"; // this will occurs an error
```

Types

Computers are sophisticated and can make use of more complex variables than just numbers. This is where variable types come in. Variables come in several types and different languages support different types.

The most common types are:

- **Number**: Numbers can be integers (e.g., `1`, `-5`, `100`) or floating-point values (e.g., `3.14`, `-2.5`, `0.01`). JavaScript does not have a separate type for integers and floating-point values; it treats them both as numbers.
- **String**: Strings are sequences of characters, represented by either single quotes (e.g., `'hello'`) or double quotes (e.g., `"world"`).
- **Boolean**: Booleans represent a true or false value. They can be written as `true` or `false` (without quotes).
- **Null**: The null type represents a null value, which means "no value." It can be written as `null` (without quotes).
- **Undefined**: The undefined type represents a value that has not been set. If a variable has been declared, but has not been assigned a value, it is undefined.
- **Object**: An object is a collection of properties, each of which has a name and a value. You can create an object using curly braces (`{}`) and assigning properties to it using name-value pairs.
- **Array**: An array is a special type of object that can hold a collection of items. You can create an array using square brackets (`[]`) and assigning a list of values to it.
- **Function**: A function is a block of code that can be defined and then called by name. Functions can accept arguments (inputs) and return a value (output). You can create a function using the `function` keyword.

JavaScript is a "*loosely typed*" language, which means that you don't have to explicitly declare what type of data the variables are. You just need to use the `var` keyword to indicate that you are declaring a variable, and the interpreter will work out what data type you are using from the context, and use of quotes.

Exercise

Declare three variables and initialize them with the following values: `age` as a number, `name` as a string and `isMarried` as a boolean.

```
let age =
let name =
let isMarried =
```

The `typeof` operator is used to checking the datatypes of a variable.

```
typeof "John"                 // Returns "string"
typeof 3.14                   // Returns "number"
typeof NaN                    // Returns "number"
typeof false                  // Returns "boolean"
typeof [1,2,3,4]              // Returns "object"
typeof {name:'John', age:34}  // Returns "object"
typeof new Date()             // Returns "object"
typeof function () {}         // Returns "function"
typeof myCar                  // Returns "undefined" *
typeof null                   // Returns "object
```

Data types used in JavaScript can be differentiated into two categories based on containing values.

Data types that can contain values:

- `string`
- `number`
- `boolean`
- `object`
- `function`

`Object` , `Date` , `Array` , `String` , `Number` , and `Boolean` are the types of objects available in JavaScript.

Data types that cannot contain values:

- `null`
- `undefined`

A primitive data value is a simple data value with no additional properties and methods and is not an object. They are immutable, meaning that they can't be altered. There are 7 primitive data types:

- string
- number
- bigint
- boolean
- undefined
- symbol
- null

Exercise

Declare a variable called `person` and initialize it with an object that contains the following properties: `age` as a number, `name` as a string and `isMarried` as a boolean.

```
let person =
```

Equality

Programmers frequently need to determine the equality of variables in relation to other variables. This is done using an equality operator.

The most basic equality operator is the `==` operator. This operator does everything it can to determine if two variables are equal, even if they are not of the same type.

For example, assume:

```
let foo = 42;
let bar = 42;
let baz = "42";
let qux = "life";
```

`foo == bar` will evaluate to `true` and `baz == qux` will evaluate to `false`, as one would expect. However, `foo == baz` will also evaluate to `true` despite `foo` and `baz` being different types. Behind the scenes the `==` equality operator attempts to force its operands to the same type before determining their equality. This is in contrast to the `===` equality operator.

The `===` equality operator determines that two variables are equal if they are of the same type *and* have the same value. With the same assumptions as before, this means that `foo === bar` will still evaluate to `true`, but `foo === baz` will now evaluate to `false`. `baz === qux` will still evaluate to `false`.

Exercise

Use the `==` and `===` operator to compare the values of `str1` and `str2`.

```
let str1 = "hello";
let str2 = "HELLO";
let bool1 = true;
let bool2 = 1;
// compare using ==
let stringResult1 =
let boolResult1 =
// compare using ===
let stringResult1 =
let boolResult2 =
```

Numbers

JavaScript has **only one type of number** – 64 bit float point. It's the same as Java's `double`. Unlike most other programming languages, there is no separate integer type, so 1 and 1.0 are the same value. Creating a number is easy, it can be done just like for any other variable type using the `var` keyword.

Numbers can be created from a constant value:

```
// This is a float:
let a = 1.2;

// This is an integer:
let b = 10;
```

Or from the value of another variable:

```
let a = 2;
let b = a;
```

The precision of integers is accurate up to 15 digits and the maximum number is 17.

```
let x = 9999999999999999;   // x will be 9999999999999999
let y = 99999999999999999;  // y will be 10000000000000000
```

It interprets numeric constants as hexadecimal if they are preceded by `0x`.

```
let z = 0xFF; // 255
```

Math

The `Math` object allows performing mathematical operations in JavaScript. The Math object is static and doesn't have a constructor. One can use method and properties of Math object without creating a Math object first. For accessing its property one can use *Math.property.* Some of the math properties are described below:

```
Math.E        // returns Euler's number
Math.PI       // returns PI
Math.SQRT2    // returns the square root of 2
Math.SQRT1_2  // returns the square root of 1/2
Math.LN2      // returns the natural logarithm of 2
Math.LN10     // returns the natural logarithm of 10
Math.LOG2E    // returns base 2 logarithm of E
Math.LOG10E   // returns base 10 logarithm of E
```

Examples of some of the math methods are:

```
Math.pow(8, 2); // 64
Math.round(4.6); // 5
Math.ceil(4.9); // 5
Math.floor(4.9); // 4
Math.trunc(4.9); // 4
Math.sign(-4); // -1
Math.sqrt(64); // 8
Math.abs(-4.7); // 4.7
Math.sin(90 * Math.PI / 180);  // 1 (the sine of 90 degrees)
Math.cos(0 * Math.PI / 180); // 1 (the cos of 0 degrees)
Math.min(0, 150, 30, 20, -8, -200); // -200
Math.max(0, 150, 30, 20, -8, -200); // 150
Math.random(); // 0.44763808380924375
Math.log(2); // 0.6931471805599453
Math.log2(8); // 3
Math.log10(1000); // 3
```

To access maths method, one can call its methods directly with arguments wherever necessary.

Method	Description
`abs(x)`	Returns absolute value of `x`
`acos(x)`	Returns arccosine of `x`, in radians
`acosh(x)`	Returns hyperbolic arccosine of `x`
`asin(x)`	Returns arcsine of `x`, in radians
`asinh(x)`	Returns hyperbolic arcsine of `x`
`atan(x)`	Returns arctangent of `x` as a numeric value between `-PI/2` and `PI/2` radians
`atan2(y,x)`	Returns arctangent of the quotient of its arguments
`atanh(x)`	Returns hyperbolic arctangent of `x`
`crbt(x)`	Returns cubic root of `x`
`ceil(x)`	Returns rounded upwards to the nearest integer of `x`
`cos(x)`	Returns consine of `x`, in radians
`cosh(x)`	Returns hyperbolic cosine of `x`
`exp(x)`	Returns exponential value of `x`
`floor(x)`	Returns round downwards to the neareast integer of `x`
`log(x)`	Returns natural logarithmetic of `x`
`max(x,y,z,... n)`	Returns number with the highest value
`min(x,y,z,... n)`	Returns number with the lowest value
`pow(x,y)`	Returns value of `x` to the power of `y`
random()	Returns number between 0 and 1
round(x)	Rounds number to the neareast x
sign(x)	Returns if x is negative, `null` or positive (-1,0,1)
sin(x)	Returns sine of x, in radians
sinh(x)	Returns hyperbolic sine of x
sqrt(x)	Returns square root of x
tan(x)	Returns tangent of an angle
tanh(x)	Returns hyperbolic tangent of x
trunc(x)	Returns integer part of a number (x)

Basic Operators

Mathematic operations to numbers can be performed using some basic operators like:

- **Addition operator (`+`)**: The addition operator adds two numbers together. For example:

```
console.log(1 + 2); // 3
console.log(1 + (-2)); // -1
```

- **Subtraction operator (`-`)**: The subtraction operator subtracts one number from another. For example:

```
console.log(3 - 2); // 1
console.log(3 - (-2)); // 5
```

- **Multiplication operator (`*`)**: The multiplication operator multiplies two numbers. For example:

```
console.log(2 * 3); // 6
console.log(2 * (-3)); // -6
```

- **Division operator (`/`)**: The division operator divides one number by another. For example:

```
console.log(6 / 2); // 3
console.log(6 / (-2)); // -3
```

- **Remainder operator (`%`)**: The remainder operator returns the remainder of a division operation. For example:

```
console.log(10 % 3); // 1
console.log(11 % 3); // 2
console.log(12 % 3); // 0
```

The JavaScript interpreter works from left to right. One can use parentheses just like in math to separate and group expressions: `c = (a / b) + d`

> JavaScript uses the `+` operator for both addition and concatenation. Numbers are added whereas strings are concatenated.

`NaN` is a reserved word indicating that a number is not a legal number, this arises when we perform arithmetic with a non-numeric string will result in `NaN` (Not a Number).

```
let x = 100 / "10";
```

The `parseInt` method parses a value as a string and returns the first integer.

```
parseInt("10"); // 10
parseInt("10.00"); // 10
parseInt("10.33"); // 10
parseInt("34 45 66"); // 34
parseInt(" 60 "); // 60
parseInt("40 years"); //40
parseInt("He was 40"); //NaN
```

In JavaScript, if we calculate a number outside the largest possible number it returns `Infinity` .

```
let x =  2 / 0; // Infinity
let y = -2 / 0; // -Infinity
```

Exercise

Use the math operators +, -, *, /, and % to perform the following operations on `num1` and `num2`.

```
let num1 = 10;
let num2 = 5;

// Add num1 and num2.
let addResult =
// Subtract num2 from num1.
let subtractResult =
// Multiply num1 and num2.
let multiplyResult =
// Divide num1 by num2.
let divideResult =
// Find the remainder num1 is divided by num2.
let reminderResult =
```

Advanced Operators

When operators are put together without parenthesis, the order in which they are applied is determined by the *precedence* of the operators. Multiplication `(*)` and division `(/)` has higher precedence than addition `(+)` and subtraction `(-)`.

```
// multiplication is done first, which is then followed by addition
let x = 100 + 50 * 3; // 250
// with parenthesis operations inside the parenthesis are computed first
let y = (100 + 50) * 3; // 450
// operations with the same precedences are computed from left to right
let z = 100 / 50 * 3;
```

Several advanced math operators can use be used in our code. Here is a list of some of the main advanced math operators:

- **Modulo operator (`%`)**: The modulo operator returns the remainder of a division operation. For example:

```
console.log(10 % 3); // 1
console.log(11 % 3); // 2
console.log(12 % 3); // 0
```

- **Exponentiation operator (* *)**: The exponentiation operator raises a number to the power of another number. It is a newer operator and is not supported in all browsers, so you may need to use the `Math.pow` function instead. For example:

```
console.log(2 ** 3); // 8
console.log(3 ** 2); // 9
console.log(4 ** 3); // 64
```

- **Increment operator (`++`)**: The increment operator increments a number by one. It can be used as a prefix (before the operand) or a postfix (after the operand). For example:

```
let x = 1;
x++; // x is now 2
++x; // x is now 3
```

- **Decrement operator (`--`)**: The decrement operator decrements a number by one. It can be used as a prefix (before the operand) or a postfix (after the operand). For example:

```
let y = 3;
y--; // y is now 2
--y; // y is now 1
```

- **Math object**: The `Math` object is a built-in object in JavaScript that provides mathematical functions and constants. You can use the methods of the `Math` object to perform advanced math operations, such as finding the square root of a number, calculating the sine of a number, or generating a random number. For example:

```
console.log(Math.sqrt(9)); // 3
console.log(Math.sin(0)); // 0
console.log(Math.random()); // a random number between 0 and 1
```

These are just a few examples of the advanced math operators and functions available in JavaScript. There are many more that you can use to perform advanced math operations in your code.

Exercise

Use the following advanced operators to perform operations on `num1` and `num2`.

```
let num1 = 10;
let num2 = 5;

// ++ operator to increment the value of num1.
const result1 =
// -- operator to decrement the value of num2.
const result2 =
//  += operator to add num2 to num1.
const result3 =
// -= operator to subtract num2 from num1.
const result4 =
```

Nullish coalescing operator '??'

The `nullish` coalescing operator returns the first argument if it's not `null/undefined`, else the second one. It is written as two question marks `??`. The result of `x ?? y` is:

- if `x` is defined, then `x`,
- if `y` isn't defined, then `y`.

> It's a recent addition to the language and might need polyfills to support old browsers

Strings

JavaScript strings share many similarities with string implementations from other high-level languages. They represent text-based messages and data.

In this course, we will cover the basics. How to create new strings and perform common operations on them.

Here is an example of a string:

```
"Hello World"
```

String indexes are zero-based, meaning that starting position of the first character at `0` followed by others in incremental order.

Various methods are supported by string and return a new value. These methods are described below.

Name	Description
`charAt()`	Returns character at specified index
`charCodeAt()`	Returns Unicode character at specified index
`concat()`	Returns two or more combined strings
`constructor`	Returns string's constructor function
`endsWith()`	Checks if a string ends with a specified value
`fromCharCode()`	Returns Unicode values as characters
`includes()`	Checks if a string contains with a specified value
`indexOf()`	Returns the index of its first occurance
`lastIndexOf()`	Returns the index of its last occurance
`length`	Returns the length of the string
`localeCompare()`	Compares two strings with locale
`match()`	Matches a string against a value or regular expression
`prototype`	Used to add properties and method of an object
`repeat()`	Returns new string with number of copies specified
`replace()`	Returns a string with values replaced by a regular expression or a string with a value
`search()`	Returns an index based on a string's match against a value or regular expression
`slice()`	Returns a string containing part of a string
`split()`	Splits string into array of substrings
`startsWith()`	Checks strings begining against specifed character
`substr()`	Extracts part of string, from start index
`substring()`	Extracts part of string, between two indices
`toLocalLowerCase()`	Returns string with lowercase characters using host's locale
`toLocalUpperCase()`	Returns string with uppercase characters using host's locale
`toLowerCase()`	Returns string with lowercase characters
`toString()`	Returns string or string object as string
`toUpperCase()`	Returns string with uppercase characters
`trim()`	Returns string with removed whitespaces
`trimEnd()`	Returns string with removed whitespaces from end
`trimStart()`	Returns string with removed whitespaces from start
`valueOf()`	Returns primitive value of string or string object

Creation

You can define strings in JavaScript by enclosing the text in single quotes or double quotes:

```
// Single quotes can be used
let str = "Our lovely string";

// Double quotes as well
let otherStr = "Another nice string";
```

In Javascript, Strings can contain UTF-8 characters:

```
"中文 español English हिन्दी العربية português বাংলা русский 日本語 ਪੰਜਾਬੀ 한국어";
```

You can also use the `String` constructor to create a string object:

```
const stringObject = new String('This is a string');
```

However, it is generally not recommended to use the `String` constructor to create strings, as it can cause confusion between string primitives and string objects. It is usually better to use string literals to create strings.

You can also use template literals to create strings. Template literals are strings that are enclosed in backticks (``) and can contain placeholders for values. Placeholders are denoted with the `${}` syntax.

```
const name = 'John';
const message = `Hello, ${name}!`;
```

Template literals can also contain multiple lines and can include any expression inside the placeholders.

> Strings can not be subtracted, multiplied, or divided.

Exercise

Use a template literal to create a string that includes the values of `name` and `age`. The string should have the following format: "My name is John and I am 25 years old.".

```
let name = "John";
let age = 25;

// My name is John and I am 25 years old.
let result =
```

Replace

The `replace` method allows us to replace a character, word, or sentence with a string. For example.

```
let str = "Hello World!";
let new_str = str.replace("Hello", "Hi");

console.log(new_str);

// Result: Hi World!
```

> To replace a value on all instances of a regular expression with a `g` modifier is set.

It searches for a string for a value or a regular expression and returns a new string with the value(s) replaced. It doesn't change the original string. Let's see the global case-insensitive replacement example.

```
let text = "Mr Blue has a blue house and a blue car";
let result = text.replace(/blue/gi, "red");

console.log(result);
//Result: Mr red has a red house and a red car
```

Length

It's easy in Javascript to know how many characters are in a string using the property `.length`. The `length` property returns the number of characters in the string, including spaces and special characters.

```
let size = "Our lovely string".length;
console.log(size);
// size: 17

let emptyStringSize = "".length
console.log(emptyStringSize);
// emptyStringSize: 0
```

The length property of an empty string is `0`.

> The `length` property is a read-only property, so you cannot assign a new value to it.

Concatenation

Concatenation involves adding two or more strings together, creating a larger string containing the combined data of those original strings. The concatenation of a string appends one or more strings to another string. This is done in JavaScript using the following ways.

- using the `+` operator
- using the `concat()` method
- using the array `join()` method
- using the template literal (introduced in ES6)

The string `concat()` method accepts the list of strings as parameters and returns a new string after concatenation i.e. combination of all the strings. Whereas the array `join()` method is used to concatenate all the elements present in an array by converting them into a single string.

The template literal uses backtick `(``)` and provides an easy way to create multiline strings and perform string interpolation. An expression can be used inside the backtick using `$` sign and curly braces `${expression}` .

```
const icon = '👋';
// using template Strings
`hi ${icon}`;

// using join() Method
['hi', icon].join(' ');

// using concat() Method
''.concat('hi ', icon);

//  using + operator
'hi ' + icon;
// hi 👋
```

Conditional Logic

A condition is a test for something. Conditions are very important for programming, in several ways:

First of all, conditions can be used to ensure that your program works, regardless of what data you throw at it for processing. If you blindly trust data, you'll get into trouble and your programs will fail. If you test that the thing you want to do is possible and has all the required information in the right format, that won't happen, and your program will be a lot more stable. Taking such precautions is also known as programming defensively.

The other thing conditions can do for you is allow for branching. You might have encountered branching diagrams before, for example when filling out a form. Basically, this refers to executing different "branches" (parts) of code, depending on if the condition is met or not.

In this chapter, we'll learn the basis of conditional logic in JavaScript.

If

The easiest condition is an if statement and its syntax is `if(condition){ do this … }` . The condition has to be true for the code inside the curly braces to be executed. You can for example test a string and set the value of another string dependent on its value as described below.

```
let country = "France";
let weather;
let food;
let currency;

if (country === "England") {
  weather = "horrible";
  food = "filling";
  currency = "pound sterling";
}

if (country === "France") {
  weather = "nice";
  food = "stunning, but hardly ever vegetarian";
  currency = "funny, small and colourful";
}

if (country === "Germany") {
  weather = "average";
  food = "wurst thing ever";
  currency = "funny, small and colourful";
}

let message =
  "this is " +
  country +
  ", the weather is " +
  weather +
  ", the food is " +
  food +
  " and the " +
  "currency is " +
  currency;

console.log(message);
// 'this is France, the weather is nice, the food is stunning, but hardly ever vegetarian and the currency is f
```

Conditions can also be nested.

Else

There is also an `else` clause that will be applied when the first condition isn't true. This is very powerful if you want to react to any value, but single out one in particular for special treatment.

```
let umbrellaMandatory;

if (country === "England") {
  umbrellaMandatory = true;
} else {
  umbrellaMandatory = false;
}
```

The `else` clause can be joined with another `if`. Let's remake the example from the previous article.

```
if (country === "England") {
  ...
} else if (country === "France") {
  ...
} else if (country === "Germany") {
  ...
}
```

Exercise

From the following values write a conditional statement that checks if `num1` is greater than `num2`. If it is, assign "num1 is greater than num2" to the `result` variable. If it is not, assign "num1 is less than or equal to num2".

```
let num1 = 10;
let num2 = 5;
let result;

// check if num1 is greater than num2
if( condition ) {

}else {

}
```

Switch

A `switch` is a conditional statement that performs actions based on different conditions. It uses strict (`===`) comparison to match the conditions and executes the code blocks of matched condition. The syntax of the `switch` expression is shown below.

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

The expression is evaluated once and is compared with each case. If a match is found, then the associated code block is executed if not `default` code block is executed. The `break` keyword stops the execution and can be placed anywhere. In its absence, the next condition is evaluated even if the conditions are not matched.

An example of getting a weekday name based on the switch condition is shown below.

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
     day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
}
```

In multiple matching cases, the **first** matching value is selected, if not the default value is selected. In the absence of default and no matching case, the program continues to the next statement(s) after switch conditions.

Exercise

From the following values write a `switch` statement that checks the value of dayOfWeek. If dayOfWeek is "Monday", "Tuesday", "Wednesday", "Thursday", or "Friday", assign "It's a weekday" to the result variable. If `dayOfWeek` is "Saturday" or "Sunday", assign "It's the weekend" to the result.

```
let dayOfWeek = "Monday";
let result;
// check if it's a weekday or the weekend
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

Comparators

Lets now focus on the conditional part:

```
if (country === "France") {
    ...
}
```

The conditional part is the variable `country` followed by the three equal signs (`===`). Three equal signs tests if the variable `country` has both the correct value (`France`) and also the correct type (`String`). You can test conditions with double equal signs, too, however a conditional such as `if (x == 5)` would then return true for both `var x = 5;` and `var x = "5";` . Depending on what your program is doing, this could make quite a difference. It is highly recommended as a best practice that you always compare equality with three equal signs (`===` and `!==`) instead of two (`==` and `!=`).

Other conditional tests:

- `x > a` : is x bigger than a?
- `x < a` : is x less than a?
- `x <= a` : is x less than or equal to a?
- `x >=a` : is x greater than or equal to a?
- `x != a` : is x not a?
- `x` : does x exist?

Logical Comparison

In order to avoid the if-else hassle, simple logical comparisons can be utilised.

```
let topper = marks > 85 ? "YES" : "NO";
```

In the above example, `?` is a logical operator. The code says that if the value of marks is greater than 85 i.e. `marks > 85` , then `topper = YES` ; otherwise `topper = NO` . Basically, if the comparison condition proves true, the first argument is accessed and if the comparison condition is false, the second argument is accessed.

Concatenate

Furthermore, you can concatenate different conditions with " `or` " or " `and` " statements, to test whether either statement is true, or both are true, respectively.

In JavaScript “or” is written as `||` and “and” is written as `&&` .

Say you want to test if the value of x is between 10 and 20—you could do that with a condition stating:

```
if (x > 10 && x < 20) {
    ...
}
```

If you want to make sure that country is either “England” or “Germany” you use:

```
if (country === "England" || country === "Germany") {
    ...
}
```

Note: Just like operations on numbers, Conditions can be grouped using parenthesis, ex: `if ( (name === "John" || name === "Jennifer") && country === "France")` .

Arrays

Arrays are a fundamental part of programming. An array is a list of data. We can store a lot of data in one variable, which makes our code more readable and easier to understand. It also makes it much easier to perform functions on related data.

The data in arrays are called **elements**.

Here is a simple array:

```
// 1, 1, 2, 3, 5, and 8 are the elements in this array
let numbers = [1, 1, 2, 3, 5, 8];
```

Arrays can be created easily using array literals or with a `new` keyword.

```
const cars = ["Saab", "Volvo", "BMW"]; // using array literals
const cars = new Array("Saab", "Volvo", "BMW"); // using the new keyword
```

An index number is used to access the values of an array. The index of the first element in an array is always `0` as array indexes start with `0` . The index number can also be used to change the elements of an array.

```
const cars = ["Saab", "Volvo", "BMW"];
console.log(cars[0]);
// Result: Saab

cars[0] = "Opel"; // changing the first element of an array
console.log(cars);
// Result: ['Opel', 'Volvo', 'BMW']
```

> Arrays are a special type of object. One can have objects in an array.

The `length` property of an array returns the count of numbers elements. Methods supported by Arrays are shown below:

Name	Description
`concat()`	Returns two or more combined arrays
`join()`	Joins all elements in an array into a string
`push()`	Adds one or more elements at the end of the array and returns the length
`pop()`	Removes the last element of an array and returns that element
`shift()`	Removes the first element of an array and returns that element
`unshift()`	Adds one or more elements at the front of an array and returns the length
`slice()`	Extracts the section of an array and returns the new array
`at()`	Returns element at the specified index or `undefined`
`splice()`	Removes elements from an array and (optionally) replaces them, and returns the array
`reverse()`	Transposes the elements of an array and returns a reference to an array
`flat()`	Returns a new array with all sub-array elements concatenated into it recursively up to the specified depth
`sort()`	Sorts the elements of an array in place, and returns a reference to the array
`indexOf()`	Returns the index of the first match of the search element
`lastIndexOf()`	Returns the index of the last match of the search element
`forEach()`	Executes a callback in each element of an array and returns undefined
`map()`	Returns a new array with a return value from executing `callback` on every array item.
`flatMap()`	Runs `map()` followed by `flat()` of depth 1
`filter()`	Returns a new array containing the items for which `callback` returned `true`
`find()`	Returns the first item for which `callback` returned `true`
`findLast()`	Returns the last item for which `callback` returned `true`
`findIndex()`	Returns the index of the first item for which `callback` returned `true`
`findLastIndex()`	Returns the index of the last item for which `callback` returned `true`
`every()`	Returns `true` if `callback` returns `true` for every item in the array
`some()`	Returns `true` if `callback` returns `true` for at least one item in the array
`reduce()`	Uses `callback(accumulator, currentValue, currentIndex, array)` for reducing purpose and returns the final value returned by `callback` function
`reduceRight()`	Works similarly lie `reduce()` but starts with the last element

Unshift

The `unshift` method adds new elements sequentially to the start, or front of the array. It modifies the original array and returns the new length of the array. For example.

```
let array = [0, 5, 10];
array.unshift(-5);  // 4

// RESULT: array = [-5 , 0, 5, 10];
```

> The `unshift()` method overwrites the original array.

The `unshift` method takes one or more arguments, which represent the elements to be added to the beginning of the array. It adds the elements in the order they are provided, so the first element will be the first element of the array.

Here is an example of using `unshift` to add multiple elements to an array:

```
const numbers = [1, 2, 3];
const newLength = numbers.unshift(-1, 0);
console.log(numbers); // [-1, 0, 1, 2, 3]
console.log(newLength); // 5
```

Map

The `Array.prototype.map()` method iterates over an array and modifies its elements using a callback function. The callback function will then be applied to each element of the array.

Here's the syntax for using `map`.

```
let newArray = oldArray.map(function(element, index, array) {
  // element: current element being processed in the array
  // index: index of the current element being processed in the array
  // array: the array map was called upon
  // Return element to be added to newArray
});
```

For example, let's say you have an array of numbers and you want to create a new array that doubles the values of the numbers in the original array. You could do this using `map` like this.

```
const numbers = [2, 4, 6, 8];

const doubledNumbers = numbers.map(number => number * 2);

console.log(doubledNumbers);

// Result: [4, 8, 12, 16]
```

You can also use the arrow function syntax to define the function passed to `map`.

```
let doubledNumbers = numbers.map((number) => {
  return number * 2;
});
```

or

```
let doubledNumbers = numbers.map(number => number * 2);
```

> The `map()` method doesn't execute function for empty elements and doesn't change the original array.

Spread

An array or object can be quickly copied into another array or object by using the Spread Operator `(...)`. It allows an iterable such as an array to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

Here are some examples of it:

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

console.log(...arr);
// Result: 1 2 3 4 5

let arr1;
arr1 = [...arr]; //copies the arr into arr1

console.log(arr1);    //Result: [1, 2, 3, 4, 5]

arr1 = [6,7];
arr = [...arr,...arr1];

console.log(arr);   //Result: [1, 2, 3, 4, 5, 6, 7]
```

> The spread operator only works in modern browsers that support the feature. If you need to support older browsers, you will need to use a transpiler like Babel to convert the spread operator syntax to equivalent ES5 code.

Shift

`shift` deletes the first index of that array and moves all indexes to the left. It changes the original array. Here's the syntax for using `shift`:

```
array.shift();
```

For example:

```
let array = [1, 2, 3];
array.shift();

// Result: array = [2,3]
```

You can also use the `shift` method in conjunction with a loop to remove all elements from an array. Here's an example of how you might do this:

```
while (array.length > 0) {
  array.shift();
}

console.log(array); // Result: []
```

> The `shift` method only works on arrays, and not on other objects that are similar to arrays such as arguments objects or NodeList objects. If you need to shift elements from one of these types of objects, you will need to convert it to an array first using the `Array.prototype.slice()` method.

Pop

The `pop` method removes the last element from an array and returns that element. This method changes the length of the array.

Here's the syntax for using `pop`:

```
array.pop();
```

For example:

```
let arr = ["one", "two", "three", "four", "five"];
arr.pop();

console.log(arr);

// Result: ['one', 'two', 'three', 'four']
```

You can also use the `pop` method in conjunction with a loop to remove all elements from an array. Here's an example of how you might do this:

```
while (array.length > 0) {
  array.pop();
}

console.log(array); // Result: []
```

> The `pop` method only works on arrays, and not on other objects that are similar to arrays such as arguments objects or NodeList objects. If you need to pop elements from one of these types of objects, you will need to convert it to an array first using the `Array.prototype.slice()` method.

Join

The `join` method, makes an array turn into a string and joins it all together. It does not change the original array. Here's the syntax for using `join`:

```
array.join([separator]);
```

The `separator` argument is optional and specifies the character to be used to separate the elements in the resulting string. If omitted, the array elements are separated with a comma (`,`).

For example:

```
let array = ["one", "two", "three", "four"];

console.log(array.join(" "));

// Result: one two three four
```

> Any separator can be specified but the default one is a comma `(,)`.

In the above example, a space is used as a separator. You can also use `join` to convert an array-like object (such as an arguments object or a NodeList object) to a string by first converting it to an array using the `Array.prototype.slice()` method:

```
function printArguments() {
  console.log(Array.prototype.slice.call(arguments).join(', '));
}

printArguments('a', 'b', 'c'); // Result: "a, b, c"
```

Length

Arrays have a property called `length`, and it's pretty much exactly as it sounds, it's the length of the array.

```
let array = [1, 2, 3];

let l = array.length;

// Result: l = 3
```

The length property also sets the number of elements in an array. For example.

```
let fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.length = 2;

console.log(fruits);
// Result: ['Banana', 'Orange']
```

You can also use the `length` property to get the last element of an array by using it as an index. For example:

```
console.log(fruits[fruits.length - 1]); // Result: Orange
```

You can also use the `length` property to add elements to the end of an array. For example:

```
fruits[fruits.length] = "Pineapple";
console.log(fruits); // Result: ['Banana', 'Orange', 'Pineapple']
```

> The `length` property is automatically updated when elements are added or removed from the array.

It's also worth noting that the `length` property is not a method, so you don't need to use parentheses when accessing it. It's simply a property of the array object that you can access like any other object property.

Push

One can push certain items to an array making the last index the newly added item. It changes the length of an array and returns a new length.

Here's the syntax for using `push`:

```
array.push(element1[, ...[, elementN]]);
```

The `element1, ..., elementN` arguments represent the elements to be added to the end of the array.

For example:

```
let array = [1, 2, 3];
array.push(4);

console.log(array);

// Result: array = [1, 2, 3, 4]
```

You can also use `push` to add elements to the end of an array-like object (such as an arguments object or a NodeList object) by first converting it to an array using the `Array.prototype.slice()` method:

```
function printArguments() {
  let args = Array.prototype.slice.call(arguments);
  args.push('d', 'e', 'f');
  console.log(args);
}

printArguments('a', 'b', 'c'); // Result: ["a", "b", "c", "d", "e", "f"]
```

Note that the `push` method modifies the original array. It does not create a new array.

For Each

The `forEach` method executes a provided function once for each array element. Here's the syntax for using `forEach`:

```
array.forEach(function(element, index, array) {
  // element: current element being processed in the array
  // index: index of the current element being processed in the array
  // array: the array forEach was called upon
});
```

For example, let's say you have an array of numbers and you want to print the double of each number to the console. You could do this using `forEach` like this:

```
let numbers = [1, 2, 3, 4, 5];
numbers.forEach(function(number) {
  console.log(number * 2);
});
```

You can also use the arrow function syntax to define the function passed to `forEach`:

```
numbers.forEach((number) => {
  console.log(number * 2);
});
```

or

```
numbers.forEach(number => console.log(number * 2));
```

`forEach` does not modify the original array. It simply iterates over the elements of the array and executes the provided function for each element.

> The `forEach()` method is not executed for the empty statment.

Sort

The `sort` method sorts the items of an array in a specific order (ascending or descending).

Here's the syntax for using `sort` :

```
array.sort([compareFunction]);
```

The `compareFunction` argument is optional and specifies a function that defines the sort order. If omitted, the elements are sorted in ascending order according to their string representation.

For example:

```
let city = ["California", "Barcelona", "Paris", "Kathmandu"];
let sortedCity = city.sort();

console.log(sortedCity);

// Result: ['Barcelona', 'California', 'Kathmandu', 'Paris']
```

> Numbers can be sorted incorrectly when they are sorted. For example, "35" is bigger than "100", because "3" is bigger than "1".

To fix the sorting issue in numbers, compare functions are used. Compare functions defines sort orders and return a **negative**, **zero**, or **positive** value based on arguments, like this:

- A negative value if `a` should be sorted before `b`
- A positive value if `a` should be sorted after `b`
- `0` if `a` and `b` are equal and their order doesn't matter

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

// Result: [1, 5, 10, 25, 40, 100]
```

> The `sort()` method overrides the original array.

Indices

So you have your array of data elements, but what if you want to access a specific element? That is where indices come in. An **index** refers to a spot in the array. indices logically progress one by one, but it should be noted that the first index in an array is 0, as it is in most languages. Brackets `[]` are used to signify you are referring to an index of an array.

```
// This is an array of strings
let fruits = ["apple", "banana", "pineapple", "strawberry"];

// We set the variable banana to the value of the second element of
// the fruits array. Remember that indices start at 0, so 1 is the
// second element. Result: banana = "banana"
let banana = fruits[1];
```

You can also use an array index to set the value of an element in an array:

```
let array = ['a', 'b', 'c', 'd', 'e'];
//  indices:  0    1    2    3    4
array[4] = 'f';
console.log(array); // Result: ['a', 'b', 'c', 'd', 'f']
```

Note that if you try to access or set an element using an index that is outside the bounds of the array (i.e., an index that is less than 0 or greater than or equal to the length of the array), you will get an `undefined` value.

```
console.log(array[5]); // Output: undefined
array[5] = 'g';
console.log(array); // Result: ['a', 'b', 'c', 'd', 'f', undefined, 'g']
```

Loops

Loops are repetitive conditions where one variable in the loop changes. Loops are handy, if you want to run the same code over and over again, each time with a different value.

Instead of writing:

```
doThing(cars[0]);
doThing(cars[1]);
doThing(cars[2]);
doThing(cars[3]);
doThing(cars[4]);
```

You can write:

```
for (var i = 0; i < cars.length; i++) {
  doThing(cars[i]);
}
```

For

The easiest form of a loop is the for statement. This one has a syntax that is similar to an if statement, but with more options:

```
for (condition; end condition; change) {
    // do it, do it now
}
```

Let's see how to execute the same code ten-times using a `for` loop:

```
for (let i = 0; i < 10; i = i + 1) {
  // do this code ten-times
}
```

> ***Note***: `i = i + 1` can be written `i++`.

To loop through the properties of an object or an array `for in` loop can also be used.

```
for (key in object) {
  // code block to be executed
}
```

Examples of `for in` loop for an object and array is shown below:

```
const person = {fname:"John", lname:"Doe", age:25};
let info = "";
for (let x in person) {
  info += person[x];
}

// Result: info = "JohnDoe25"

const numbers = [45, 4, 9, 16, 25];
let txt = "";
for (let x in numbers) {
  txt += numbers[x];
}

// Result: txt = '45491625'
```

The value of iterable objects such as `Arrays`, `Strings`, `Maps`, `NodeLists` can be looped using `for of` statement.

```
let language = "JavaScript";
let text = "";
for (let x of language) {
text += x;
}
```

While

While Loops repetitively execute a block of code as long as a specified condition is true.

```
while (condition) {
  // do it as long as condition is true
}
```

For example, the loop in this example will repetitively execute its block of code as long as the variable i is less than 5:

```
var i = 0,
  x = "";
while (i < 5) {
  x = x + "The number is " + i;
  i++;
}
```

Be careful to avoid infinite looping if the condition is always true!

Do...While

The do...while statement creates a loop that executes a specified statement until the test condition evaluates to be false. The condition is evaluated after executing the statement. The syntax for do... while is

```
do {
  // statement
} while (expression);
```

Lets for example see how to print numbers less than 10 using `do...while` loop:

```
var i = 0;
do {
  document.write(i + " ");
  i++; // incrementing i by 1
} while (i < 10);
```

> ***Note***: `i = i + 1` can be written `i++`.

Functions

Functions are one of the most powerful and essential notions in programming. Functions like mathematical functions perform transformations, they take input values called **arguments** and **return** an output value.

Functions can be created in two ways: using `function declaration` or `function expression` . The *function name* can be omitted in f `unction expression` making it an `anonymous function` . Functions, like variables, must be declared. Let's declare a function `double` that accepts an *argument* called `x` and **returns** the double of x :

```
// an example of a function declaration
function double(x) {
  return 2 * x;
}
```

> *Note:* the function above **may** be referenced before it has been defined.

Functions are also values in JavaScript; they can be stored in variables (just like numbers, strings, etc ...) and given to other functions as arguments :

```
// an example of a function expression
let double = function (x) {
  return 2 * x;
};
```

> *Note:* the function above **may not** be referenced before it is defined, just like any other variable.

A callback is a function passed as an argument to another function.

An arrow function is a compact alternative to traditional functions which has some semantic differences with some limitations. These function doesn't have their own bindings to `this` , `arguments` and `super` , and cannot be used as constructors. An example of an arrow function.

```
const double =  (x) =>  2 * x;
```

The `this` keyword in the arrow function represents the object that defined the arrow function.

Higher order

Higher order functions are functions that manipulate other functions. For example, a function can take other functions as arguments and/or produce a function as its return value. Such *fancy* functional techniques are powerful constructs available to you in JavaScript and other high-level languages like python, lisp, etc.

We will now create two simple functions, `add_2` and `double`, and a higher order function called `map`. `map` will accept two arguments, `func` and `list` (its declaration will therefore begin `map(func,list)`), and return an array. `func` (the first argument) will be a function that will be applied to each of the elements in the array `list` (the second argument).

```
// Define two simple functions
let add_2 = function (x) {
  return x + 2;
};
let double = function (x) {
  return 2 * x;
};

// map is cool function that accepts 2 arguments:
//  func    the function to call
//  list    a array of values to call func on
let map = function (func, list) {
  let output = []; // output list
  for (idx in list) {
    output.push(func(list[idx]));
  }
  return output;
};

// We use map to apply a function to an entire list
// of inputs to "map" them to a list of corresponding outputs
map(add_2, [5, 6, 7]); // => [7, 8, 9]
map(double, [5, 6, 7]); // => [10, 12, 14]
```

The functions in the above example are simple. However, when passed as arguments to other functions, they can be composed in unforeseen ways to build more complex functions.

For example, if we notice that we use the invocations `map(add_2, ...)` and `map(double, ...)` very often in our code, we could decide we want to create two special-purpose list processing functions that have the desired operation baked into them. Using function composition, we could do this as follows:

```
process_add_2 = function (list) {
  return map(add_2, list);
};
process_double = function (list) {
  return map(double, list);
};
process_add_2([5, 6, 7]); // => [7, 8, 9]
process_double([5, 6, 7]); // => [10, 12, 14]
```

Now let's create a function called `buildProcessor` that takes a function `func` as input and returns a `func`-processor, that is, a function that applies `func` to each input in list.

```
// a function that generates a list processor that performs
let buildProcessor = function (func) {
  let process_func = function (list) {
    return map(func, list);
  };
  return process_func;
};
// calling buildProcessor returns a function which is called with a list input

// using buildProcessor we could generate the add_2 and double list processors as follows:
process_add_2 = buildProcessor(add_2);
process_double = buildProcessor(double);

process_add_2([5, 6, 7]); // => [7, 8, 9]
process_double([5, 6, 7]); // => [10, 12, 14]
```

Let's look at another example. We'll create a function called `buildMultiplier` that takes a number `x` as input and returns a function that multiplies its argument by `x` :

```
let buildMultiplier = function (x) {
  return function (y) {
    return x * y;
  };
};

let double = buildMultiplier(2);
let triple = buildMultiplier(3);

double(3); // => 6
triple(3); // => 9
```

Objects

In javascript the objects are **mutable** because we change the values pointed by the reference object, instead, when we change a primitive value we are changing its reference which now is pointing to the new value and so primitive are **immutable**. The primitive types of JavaScript are `true` , `false` , numbers, strings, `null` and `undefined` . Every other value is an `object` .Objects contain `propertyName` : `propertyValue` pairs. There are three ways to create an `object` in JavaScript:

1. literal

```
let object = {};
// Yes, simply a pair of curly braces!
```

> ***Note:*** this is the **recommended** way.

2. object-oriented

```
let object = new Object();
```

> ***Note:*** it's almost like Java.

3. and using `object.create`

```
let object = Object.create(proto[, propertiesObject]);
```

> ***Note:*** it creates a new object with the specified prototype object and properties.

Properties

Object's property is a `propertyName` : `propertyValue` pair, where **property name can be only a string**. If it's not a string, it gets casted into a string. You can specify properties **when creating** an object **or later**. There may be zero or more properties separated by commas.

```
let language = {
  name: "JavaScript",
  isSupportedByBrowsers: true,
  createdIn: 1995,
  author: {
    firstName: "Brendan",
    lastName: "Eich",
  },
  // Yes, objects can be nested!
  getAuthorFullName: function () {
    return this.author.firstName + " " + this.author.lastName;
  },
  // Yes, functions can be values too!
};
```

The following code demonstrates how to **get** a property's value.

```
let variable = language.name;
// variable now contains "JavaScript" string.
variable = language["name"];
// The lines above do the same thing. The difference is that the second one lets you use litteraly any string a
variable = language.newProperty;
// variable is now undefined, because we have not assigned this property yet.
```

The following example shows how to **add** a new property **or change** an existing one.

```
language.newProperty = "new value";
// Now the object has a new property. If the property already exists, its value will be replaced.
language["newProperty"] = "changed value";
// Once again, you can access properties both ways. The first one (dot notation) is recomended.
```

Mutable

The difference between objects and primitive values is that we can **change objects**, whereas primitive values are **immutable**.

For example:

```
let myPrimitive = "first value";
myPrimitive = "another value";
// myPrimitive now points to another string.
let myObject = { key: "first value" };
myObject.key = "another value";
// myObject points to the same object.
```

You can add, modify, or delete properties of an object using the dot notation or the square bracket notation.

```
let object = {};
object.foo = 'bar'; // Add property 'foo'
object['baz'] = 'qux'; // Add property 'baz'
object.foo = 'quux'; // Modify property 'foo'
delete object.baz; // Delete property 'baz'
```

> Primitive values (such as numbers and strings) are immutable, while objects (such as arrays and objects) are mutable.

Reference

Objects are **never copied**. They are passed around by reference. An object reference is a value that refers to an object. When you create an object using the `new` operator or object literal syntax, JavaScript creates an object and assigns a reference to it.

Here's an example of creating an object using the object literal syntax:

```
var object = {
  foo: 'bar'
};
```

Here's an example of creating an object using the `new` operator:

```
var object = new Object();
object.foo = 'bar';
```

When you assign an object reference to a variable, the variable simply holds a reference to the object, not the object itself. This means that if you assign the object reference to another variable, both variables will point to the same object.

For example:

```
var object1 = {
  foo: 'bar'
};

var object2 = object1;

console.log(object1 === object2); // Output: true
```

In the example above, both `object1` and `object2` are variables that hold references to the same object. The `===` operator is used to compare the references, not the objects themselves, and it returns `true` because both variables hold references to the same object.

> You can use the `Object.assign()` method to create a new object that is a copy of an existing object.

Following is an example of an object by reference.

```
// Imagine I had a pizza
let myPizza = { slices: 5 };
// And I shared it with You
let yourPizza = myPizza;
// I eat another slice
myPizza.slices = myPizza.slices - 1;
let numberOfSlicesLeft = yourPizza.slices;
// Now We have 4 slices because myPizza and yourPizza
// reference to the same pizza object.
let a = {},
  b = {},
  c = {};
// a, b, and c each refer to a
// different empty object
a = b = c = {};
// a, b, and c all refer to
// the same empty object
```

Prototype

Every object is linked to a prototype object from which it inherits properties.

All objects created from object literals (`{}`) are automatically linked to `Object.prototype` , which is an object that comes standard with JavaScript.

When a JavaScript interpreter (a module in your browser) tries to find a property, which You want to retrieve, like in the following code:

```
let adult = { age: 26 },
  retrievedProperty = adult.age;
// The line above
```

First, the interpreter looks through every property the object itself has. For example, `adult` has only one own property — `age` . But besides that one, it actually has a few more properties, which were inherited from `Object.prototype.`

```
let stringRepresentation = adult.toString();
// the variable has value of '[object Object]'
```

`toString` is an Object.prototype's property, which was inherited. It has a value of a function, which returns a string representation of the object. If you want it to return a more meaningful representation, then you can override it. Simply add a new property to the adult object.

```
adult.toString = function () {
  return "I'm " + this.age;
};
```

If you call the `toString` function now, the interpreter will find the new property in the object itself and stop.

Thus the interpreter retrieves the first property it will find on the way from the object itself and further through its prototype.

To set your own object as a prototype instead of the default Object.prototype, you can invoke `Object.create` as follows:

```
let child = Object.create(adult);
/* This way of creating objects lets us easily replace the default Object.prototype with the one we want. In th
child.age = 8;
/* Previously, child didn't have its own age property, and the interpreter had to look further to the child's p
 Now, when we set the child's own age, the interpreter will not go further.
 Note: adult's age is still 26. */
let stringRepresentation = child.toString();
// The value is "I'm 8".
/* Note: we have not overridden the child's toString property, thus the adult's method will be invoked. If adul
```

`child` 's prototype is `adult` , whose prototype is `Object.prototype` . This sequence of prototypes is called a **prototype chain**.

Delete

A `delete` property can be used to **remove a property** from an object. When a property is deleted, it is removed from the object and cannot be accessed or enumerated (i.e., it does not show up in a for-in loop).

Here's the syntax for using `delete`:

```
delete object.property;
```

For example:

```
let adult = { age: 26 },
  child = Object.create(adult);

child.age = 8;

delete child.age;

/* Remove age property from child, revealing the age of the prototype, because then it is not overriden. */

let prototypeAge = child.age;
// 26, because child does not have its own age property.
```

> The `delete` operator only works on own properties of an object, and not on inherited properties. It also does not work on properties that have the `configurable` attribute set to `false`.

The `delete` operator does not modify the object's prototype chain. It simply removes the specified property from the object and also it does not actually destroy the object or its properties. It simply makes the properties inaccessible. If you need to destroy an object and release its memory, you can set the object to `null` or use a garbage collector to reclaim the memory.

Enumeration

Enumeration refers to the process of iterating over the properties of an object and performing a certain action for each property. There are several ways to enumerate the properties of an object in JavaScript.

One way to enumerate the properties of an object is to use the `for-in` loop. The `for-in` loop iterates over the enumerable properties of an object in an arbitrary order, and for each property it executes a given block of code.

The `for in` statement can loop over all of the property names in an object. The enumeration will include functions and prototype properties.

```
let fruit = {
    apple: 2,
    orange: 5,
    pear: 1,
  },
  sentence = "I have ",
  quantity;
for (kind in fruit) {
  quantity = fruit[kind];
  sentence += quantity + " " + kind + (quantity === 1 ? "" : "s") + ", ";
}
// The following line removes the trailing comma.
sentence = sentence.substr(0, sentence.length - 2) + ".";
// I have 2 apples, 5 oranges, 1 pear.
```

Another way to enumerate the properties of an object is to use the `Object.keys()` method, which returns an array of the object's own enumerable property names.

For example:

```
let object = {
  foo: 'bar',
  baz: 'qux'
};

let properties = Object.keys(object);
properties.forEach(function(property) {
  console.log(property + ': ' + object[property]);
});

// foo: bar
// baz: qux
```

Date and Time

The `date` object stores date and time and provides methods for managing it. Date objects are static and use a browser's default timezone to display the date as a full-text string.

To create `date` we use a `new Date()` constructor and can be created in the following ways.

```
new Date()
new Date(date string)
new Date(year,month)
new Date(year,month,day)
new Date(year,month,day,hours)
new Date(year,month,day,hours,minutes)
new Date(year,month,day,hours,minutes,seconds)
new Date(year,month,day,hours,minutes,seconds,ms)
new Date(milliseconds)
```

> Months can be specified from `0` to `11`, more than that will result in an overflow to the next year.

Methods and properties supported by date are described below:

Name	Description
`constructor`	Returns function that created the Date object's prototype
`getDate()`	Returns the day (1-31) of a month
`getDay()`	Returns the day (0-6) of a week
`getFullYear()`	Returns the year (4 digits)
`getHours()`	Returns the hour (0-23)
`getMilliseconds()`	Returns the milliseconds (0-999)
`getMinutes()`	Returns the minutes (0-59)
`getMonth()`	Returns the month (0-11)
`getSeconds()`	Returns the seconds (0-59)
`getTime()`	Returns the numeric value of a specified date in milliseconds since midnight Jan 1 1970
`getTimezoneOffset()`	Returns timezone offset in minutes
`getUTCDate()`	Returns the day (1-31) of a month according to universal time
`getUTCDay()`	Returns the day (0-6) according to universal time
`getUTCFullYear()`	Returns the year (4-digits) according to universal time
`getUTCHours()`	Returns the hours (0-23) according to universal time
`getUTCMilliseconds()`	Returns the milliseconds (0-999) according to universal time
`getUTCMinutes()`	Returns the minutes (0-59) according to universal time
`getUTCMonth()`	Returns the month (0-11) according to universal time
`getUTCSeconds()`	Returns the seconds (0-59) according to universal time
`now()`	Returns the numeric value in milliseconds since midnight Jan 1, 1970
`parse()`	Parses the date string and returns the numeric value in milliseconds since midnight Jan 1, 1970
`prototype`	Allows to add properties
`setDate()`	Sets the day of a month
`setFullYear()`	Sets the year
`setHours()`	Sets the hour
`setMilliseconds()`	Sets the milliseconds
`setMinutes()`	Sets the minutes
`setMonth()`	Sets the month
`setSeconds()`	Sets the second
`setTime()`	Sets the time
`setUTCDate()`	Sets the day of the month according to universal time
`setUTCFullYear()`	Sets the year according to the universal time
`setUTCHours()`	Sets the hour according to the universal time

Name	Description
`setUTCMilliseconds()`	Sets the milliseconds according to the universal time
`setUTCMinutes()`	Sets the minutes according to the universal time
`setUTCMonth()`	Sets the month according to the universal time
`setUTCSeconds()`	Sets the second according to the universal time
`toDateString()`	Returns the date in human readable format
`toISOString()`	Returns the date according to the ISO format
`toJSON()`	Returns the date in a string, formatted as a JSON date
`toLocaleDateString()`	Returns the date in a string using locale conventions
`toLocaleTimeString()`	Returns the time in a string using locale conventions
`toLocaleString()`	Returns date using locale conventions
`toString()`	Returns string representation of the specified date
`toTimeString()`	Returns the *time* portion into a human-readable format
`toUTCString()`	Converts date into a string according to the universal format
`toUTC()`	Returns the milliseconds since midnight Jan 1 1970 in UTC format
`valueOf()`	Returns the primitive value of `Date`

JSON

Java**S**cript **O**bject **N**otation (JSON) is a text-based format for storing and transporting data. The Javascript Objects can be easily converted into JSON and vice versa. For example.

```
//  a JavaScript object
let myObj = { name:"Ryan", age:30, city:"Austin" };

// converted into JSON:
let myJSON = JSON.stringify(myObj);
console.log(myJSON);
// Result: '{"name":"Ryan","age":30,"city":"Austin"}'

//converted back to JavaScript object
let originalJSON = JSON.parse(myJSON);
console.log(originalJSON);

// Result: {name: 'Ryan', age: 30, city: 'Austin'}
```

`stringify` and `parse` are the two methods supported by JSON.

Method	Description
`parse()`	Returns JavaScript object from the parsed JSON string
`stringify()`	Returns JSON string from JavaScript Object

The following data types are supported by JSON.

- string
- number
- array
- boolean
- object with valid JSON values
- `null`

It can not be `function`, `date` or `undefined`.

try... catch

In programming errors happen for various reasons, some happen from code errors, some due to wrong input, and other unforeseeable things. When an error happens, the code stops and generates an error message usually seen in the console.

Instead of halting the code execution, we can use the `try...catch` construct that allows catching errors without dying the script. The `try...catch` construct has two main blocks; `try` and then `catch`.

```
try {
  // code...
} catch (err) {
  // error handling
}
```

At first, the code in the `try` block is executed. If no errors are encountered then it skips the `catch` block. If an error occurs then the `try` execution is stopped, moving the control sequence to the `catch` block. The cause of the error is captured in `err` variable.

```
try {
  // code...
  alert('Welcome to Learn JavaScript');
  asdk; // error asdk variable is not defined
} catch (err) {
  console.log("Error has occurred");
}
```

`try...catch` works for runtime errors meaning that the code must be runnable and synchronous.

To throw a custom error, a `throw` statement can be used. The error object, that gets generated by errors has two main properties.

- **name**: error name
- **message**: details about the error

If we don't need an `error` message catch can omit it.

try...catch...finally

We can add one more construct to `try...catch` called `finally`, this code executes in all cases. i.e. after `try` when there is no error and after a `catch` in case of error. The syntax for `try ...catch...finally` is shown below.

```
try {
   // try to execute the code
} catch (err) {
    // handle errors
} finally {
   // execute always
}
```

Running real-world example code.

```
try {
  alert( 'try' );
} catch (err) {
  alert( 'catch' );
} finally {
  alert( 'finally' );
}
```

In the above example, the `try` block is executed first which is then followed by `finally` as there are no errors.

Exercise

Write a function `divideNumbers()` that takes two arguments numerator and denominator and returns the result of dividing numerator by denominator using following settings.

```
function divideNumbers(numerator, denominator) {
    try {
      // try statement to divide numerator by denominator.
    } catch (error) {
      // print error message
    } finally {
      // print execution has finished
    }
   // return result
  }
  let answer = divideNumbers(10, 2);
```

Regular Expression

A regular expression is an object that can either be constructed with the `RegEx` constructor or written as a literal value by enclosing a pattern in a forward slash `(/)` characters. The syntaxes for creating a regular expression are shown below.

```
// using regular expression constructor
new RegExp(pattern[, flags]);

// using literals
/pattern/modifiers
```

The flags are optional while creating a regular expression using literals. Example of creating identical regular using above mentioned method is as follows.

```
let re1 = new RegExp("xyz");
let re2 = /xyz/;
```

Both ways will create a regex object and have the same methods and properties. There are cases where we might need dynamic values to create a regular expression, in that case, literals won't work and have to go with the constructor.

> In cases where we want to have a forward slash to be a part of a regular expression, we have to escape the forward slash `(/)` with backslash `(\)`.

The different modifiers that are used to perform case-insensitive searches are:

- `g` - global search (finds all matches instead of stopping after the first match)
- `i` - case insensitive search
- `m` - multiline matching

Brackets are used in a regular expression to find a range of characters. Some of them are mentioned below.

- `[abc]` - find any character between the brackets
- `[^abc]` - find any character, not between the brackets
- `[0-9]` - find any digit between the bracket
- `[^0-9]` - find any character, not between the brackets (non-digit)
- `(x|y)` - find any of the alternatives separated by |

Metacharacters are special character that has special meaning in the regular expression. These characters are further described below:

Metacharacter	Description
`.`	Match a single character excpet newline or a terminator
`\w`	Match a word character (alphanumeric character `[a-zA-Z0-9_]`)
`\W`	Match a non word character (same as `[^a-zA-Z0-9_]`)
`\d`	Match any digit character(same as `[0-9]`)
`\D`	Match any non digiti character
`\s`	Match a whitespace character (spaces, tabs etc)
`\S`	Match a non whitespace character
`\b`	Match at the begining / end of a word
`\B`	Match but not at the begining / end of a word
`\0`	Match a `NULL` character
`\n`	Match a new line character
`\f`	Match a form feed character
`\r`	Match a carriage return character
`\t`	Match a tab character
`\v`	Match a tab vertical character
`\xxx`	Match a character specified by an octal number `xxx`
`\xdd`	Match a character specified by a hexadecimal number `dd`
`\udddd`	Match Unicode character specified by a hexadecimal number `dddd`

Properties and methods supported by RegEx are listed below.

Name	Description
`constructor`	Returns function that created RegExp object's protype
`global`	Checks if the `g` modifier is set
`ignoreCase`	Checks if the `i` modifier is set
`lastIndex`	Specifies the index at which to start the next match
`multiline`	Checks if the m modifier is set
`source`	Returns the text of the string
`exec()`	Test for the match and returns the first match, if no match then it returns `null`
`test()`	Test for the match and returns the `true` or `false`
`toString()`	Returns the string value of the regular exression

A `complie()` method complies the regular expression and is deprecated.

Some examples of regular expressions are shown here.

```
let text = "The best things in life are free";
let result = /e/.exec(text); // looks for a match  of e in a string
// result: e


let helloWorldText = "Hello world!";
// Look for "Hello"
let pattern1 = /Hello/g;
let result1 = pattern1.test(helloWorldText);
// result1: true

let pattern1String = pattern1.toString();
// pattern1String : '/Hello/g'
```

Modules

In the real world, a program grows organically to cope with the needs of new functionality. With growing codebase structuring and maintaining the code requires additional work. Though it will pay off in the future, it's tempting to neglect it and allow programs to be deeply tangled. In reality, it increases the complexity of the application, as one is forced to build a holistic understanding of the system and has difficulty to look any piece in isolation. Secondly, one has to invest more time in untangling to use its functionality.

Modules come to avoid these problems. A `module` specifies which pieces of code it depends on, along with what functionality it provides for other modules to use. Modules that are dependent on another module are called *dependencies*. Various module libraries are there to organize code into modules and load it on demand.

- AMD - one of the oldest module systems, initially used by require.js.
- CommonJS - module system created for Node.js server.
- UMD - module system that is compatible with AMD and CommonJS.

Modules can load each other, and use special directives `import` and `export` to interchange functionality, and call functions of each other.

- `export` - labels functions and variables that should be accessible from outside the current module
- `import` - imports functionality from outside module

Let's see the `import` , and `export` mechanism in modules. We have `sayHi` function exported from `sayHi.js` file.

```
// 📁 sayHi.js
export const sayHi = (user) => {
  alert(`Hello, ${user}!`);
}
```

The `sayHi` function is consumed in the `main.js` file with the help of the `import` directive.

```
// 📁 main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('Kelvin'); // Hello, Kelvin!
```

Here, the import directive loads the module by importing the relative path and assigns the `sayHi` variable.

Modules can be exported in two ways: **Named** and **Default**. Furthermore, the Named exports can be assigned inline or individually.

```
// 📁 person.js

// inlined named exports
export const name = "Kelvin";
export const age = 30;

// at once
const name = "Kelvin";
const age = 30;
export {name, age};
```

> One can only have one default `export` in a file.

```
// 📁 message.js
const message = (name, age) => {
    return `${name} is ${age} years old.`;
};
export default message;
```

Based on the type of export, we can import it in two ways. The named export are constructed using curly braces whereas, default exports are not.

```
import { name, age } from "./person.js"; // named export import
import message from "./message.js"; // default export import
```

While assigning modules, we should avoid *circular dependency*. Circular dependency is a situation where module A depends on B, and B also depends on A directly or indirectly.

Classes

Classes are templates for creating an object. It encapsulates data with code to work on with data. The keyword `class` is used to create a class. And a specific method called `constructor` is used for creating and initializing an object created with a `class`. An example of car class is shown below.

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  age() {
    let date = new Date();
    return date.getFullYear() - this.year;
  }
}

let myCar = new Car("Toyota", 2021);
console.log(myCar.age()) // 1
```

> Class must be defined before its usage.

In the class body, methods or constructors are defined and executed in `strict mode`. Syntax not adhering to the strict mode results in error.

Static

The `static` keyword defines the static methods or properties for a class. These methods and properties are called in the class itself.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Hello " + x.name;
  }
}
let myCar = new Car("Toyota");

console.log(myCar.hello()); // This will throw an error
console.log(Car.hello(myCar));
// Result: Hello Toyota
```

One can access the static method or property of another static method of the same class using `this` keyword.

Inheritance

The inheritance is useful for code reusability purposes as it extends existing properties and methods of a class. The `extends` keyword is used to create a class inheritance.

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model;
  }
}

let myCar = new Model("Toyota", "Camry");
console.log(myCar.show()); // I have a Camry, it is a Toyota.
```

> The prototype of the parent class must be an `Object` or `null`.

The `super` method is used inside a constructor and refers to the parent class. With this, one can access the parent class properties and methods.

Access Modifiers

`public` , `private` , and `protected` are the three access modifiers used in class to control its access from the outside. By default, all members (properties, fields, methods, or functions) are publicly accessible from outside the class.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Hello " + x.name;
  }
}
let myCar = new Car("Toyota");
console.log(Car.hello(myCar)); // Hello Toyota
```

`private` members can access only internally within the class and cannot be accessible from outside. Private should start with `#` .

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Hello " + x.name;
  }
  #present(carname) {
    return 'I have a ' + this.carname;
  }
}
let myCar = new Car("Toyota");
console.log(myCar.#present("Camry")); // Error
console.log(Car.hello(myCar)); // Hello Toyota
```

`protected` fields are accessible only from inside the class and those extending it. These are useful for the internal interface as the inheriting class also gains access to the parent class. Protected fields with `_` .

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  _present() {
    return 'I have a ' + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this._present() + ', it is a ' + this.model;
  }
}

let myCar = new Model("Toyota", "Camry");
console.log(myCar.show()) // I have a Toyota, it is a Camry
```

Browser Object Model (BOM)

The browser object model lets us interact with the browser window. `window` object represents the browser's window and is supported by all browsers.

Object `window` is the default object for a browser, so we can specify `window` or call directly all the functions.

```
window.alert("Welcome to Learn JavaScript");

alert("Welcome to Learn JavaScript")
```

In a similar fashion, we can call other properties underneath the window object such as history, screen, navigator, location, and so on.

Window

The `window` object represents the browser window and is supported by the browsers. Global variables, objects, and functions are also part of the window object.

Global **variables** are **properties** and **functions** are **methods** of the window object.

Let's take an example of the screen properties. It is used to determine the size of the browser window and is measured in pixels.

- `window.innerHeight` - the inner height of the browser window
- `window.innerWidth` - the inner width of the browser window

> ***Note***: `window.document` is same as `document.location` as the document object model(DOM) is part of window object.

Few examples of the window methods

- `window.open()` - open a new window
- `window.close()` - close the current window
- `window.moveTo()` - move the current window
- `window.resizeTo()` - resize the current window

Popup

Popups are an additional way to show information, take user confirmation, or take user input from additional documents. A popup can navigate to a new URL and send information to the opener window. **Alert box**, **Confirmation box**, and **Prompt box** are the global functions where we can show the popup information.

1. **alert()**: It displays information to the user and has an "**OK**" button to proceed.

```
alert("Alert message example");
```

2. **confirm()**: Use as a dialog box to confirm or accept something. It has "**Ok**" and "**Cancel**" to proceed. If the user clicks "**Ok**" then it returns `true`, if click "**Cancel**" it returns `false`.

```
let txt;
if (confirm("Press a button!")) {
  txt = "You pressed OK!";
} else {
  txt = "You pressed Cancel!";
}
```

3. **prompt()**: Takes user input value with "**Ok"** and "**Cancel"** buttons. It returns `null` if the user does not provide any input value.

```
//syntax
//window.prompt("sometext","defaultText");

let person = prompt("Please enter your name", "Harry Potter");

if (person == null || person == "") {
  txt = "User cancelled the prompt.";
} else {
  txt = "Hello " + person + "! How are you today?";
}
```

Screen

The `screen` object contains the information about the screen on which the current window is being rendered. To access `screen` object we can use the `screen` property of `window` object.

```
window.screen
//or
screen
```

The `window.screen` object has different properties, some of them are listed here:

Property	Description
`height`	Represents the pixel height of the screen.
`left`	Represents the pixel distance of the current screen's left side.
`pixelDepth`	A read-only property that returns the bit depth of the screen.
`top`	Represents the pixel distance of the current screen's top.
`width`	Represents the pixel width of the screen.
`orientation`	Returns the screen orientation as specified in the Screen Orientation API
`availTop`	A read-only property that returns the first pixel from the top that is not taken up by system elements.
`availWidth`	A read-only property that returns the pixel width of the screen excluding system elements.
`colorDepth`	A read-only property that returns the number of bits used to represent colors.
`height`	Represents the pixel height of the screen.
`left`	Represents the pixel distance of the current screen's left side.
`pixelDepth`	A read-only that returns the bit depth of the screen.
`top`	Represents the pixel distance of the current screen's top.
`width`	Represents the pixel width of the screen.
`orientation`	Returns the screen orientation as specified in the Screen Orientation API

Navigator

`window.navigator` or `navigator` is a **read-only** property and contains different methods and functions related to the browser.

Let's look at a few examples of navigation.

1. **navigator.appName**: It gives the name of the browser application

```
navigator.appName;
// "Netscape"
```

> ***Note:*** "Netscape" is the application name for IE11, Chrome, Firefox, and Safari.

2. **navigator.cookieEnabled**: Returns a boolean value based on the cookie value in the browser.

```
navigator.cookieEnabled;
//true
```

3. **navigator.platform**: Provides information about the browser operating system.

```
navigator.patform;
"MacIntel"
```

Cookies 🍪

Cookies are pieces of information that are store on a computer and can be accessed by the browser.

Communication between a web browser and the server is stateless meaning that it treats each request independently. There are cases where we need to store user information and make that information available to the browser. With cookies, information can be fetched from the computer whenever it is required.

Cookies are saved in name-value pair

```
book = Learn Javascript
```

`document.cookie` property is used to create, read and delete cookies.Creating cookie is pretty easy you need to provide the name and value

```
document.cookie = "book=Learn Javacript";
```

By default, a cookie gets deleted when the browser is closed. To make it persistent, we need to specify the expiry date(in UTC time).

```
document.cookie = "book=Learn Javacript; expires=Fri, 08 Jan 2022 12:00:00 UTC";
```

We can add a parameter to tell which path the cookie belongs to. By default, the cookie belongs to the current page.

```
document.cookie = "book=Learn Javacript; expires=Fri, 08 Jan 2022 12:00:00 UTC; path=/";
```

Here is a simple example of a cookie.

```
let cookies = document.cookie;
// a simple way to reterive all cookie.

document.cookie = "book=Learn Javacript; expires=Fri, 08 Jan 2022 12:00:00 UTC; path=/";
// setting up a cookie
```

History

When we open a web browser and surf a web page it creates a new entry in the history stack. As we keep navigating to different pages new entries get pushed into the stack.

To access the history object we can use

```
window.history
// or
history
```

To navigate between the different history stack we can use `go()` , `foward()` and `back()` methods of **history** object.

1. **go()**: It is used to navigate the specific URL of the history stack.

   ```
   history.go(-1); // moves page backward
   history.go(0);  // refreshes the current page
   history.go(); // refreshes the current page
   history.go(1) // moves page forward
   ```

   > ***Note:*** the current page position in history stack is **0**.

2. **back()** : To navigate page backward we use `back()` method.

   ```
   history.back();
   ```

3. **forward()**: It loads the next list in the browser history. It is similar to clicking the forward button in the browser.

   ```
   history.forward();
   ```

Location

`Location` object is used to retrieve the current location (URL) of the document and provides different methods to manipulate document location. One can access the current location by

```
window.location
//or
document.location
//or
location
```

> ***Note***: `window.location` and `document.location` references the same location object.

Let's take an example of the following URL and explore the different properties of `location`

`http://localhost:3000/js/index.html?type=listing&page=2#title`

```
location.href //prints current document URL
location.protocol //prints protocol like http: or https:
location.host //prints hostname with port like localhost or localhost:3000
location.hostname //prints hostname like localhost or www.example.com
location.port //prints port number like 3000
location.pathname //prints pathname like /js/index.html
location.search //prints query string like ?type=listing&page=2
location.hash //prints fragment identifier like #title
```

Events

In programming, *events* are actions or occurrences in a system that the system informs you about so you can respond to them. For example, when you click the reset button it clears the input.

Interactions from the keyboard such as keypresses need to be constantly read to catch the key's state before it's released again. Performing other time-intensive computations might cause you to miss a key press. This used to be the input handling mechanism of some primitive machines. A further step up is to use a queue, I.e. a program that periodically checks the queue for new events and reacts to it. This approach is called *polling*.

The main drawback of this approach is that it has to look at the queue every now and then, causing disruption when an event is triggered. The better mechanism for this is to notify the code when an event occurs. This is what modern browsers do by allowing us to register functions as *handlers* for specific events.

```
<p>Click me to activate the handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("clicked");
  });
</script>
```

Here, the `addEventListener` is called on the `window` object (built-in object provided by the browser) to register a handler for the whole `window`. Calling its `addEventListener` method registers the second argument to be called whenever the event described by its first argument occurs.

> Event listeners are called only when the event happens in the context of the object they are registered on.

Some of the common HTML events are mentioned here.

Event	Description
`onchange`	When the user changes or modifies the value of form input
`onclick`	When the user clicks on the element
`onmouseover`	When cursor of the mouse comes over the element
`onmouseout`	When cursor of the mouse comes leaves the element
`onkeydown`	When the user press and then releases the key
`onload`	When the browser has finished the loading

It is common for handlers registered on nodes with children to also receive events from the children. For example, if a button inside a paragraph is clicked, handlers registered on the paragraph will also receive the click event. In case of the presence of handlers in both, the one at the bottom gets to go first. The event is said to *propagate* outward, from the initiating node to its parent node and on the root of the document.

The event handler can call the `stopPropagation` method on the event object to prevent handlers further up from receiving the event. This is useful in cases like, you have a button inside a clickable element and you don't want to trigger the outer element's clickable behavior from a button click.

```
<p>A paragraph with a <button>button</button>.</p>
<script>
  let para = document.querySelector("p"),
      button = document.querySelector("button");
  para.addEventListener("mousedown", () => {
    console.log("Paragraph handler.");
  });
  button.addEventListener("mousedown", event => {
    console.log("Button handler.");
    event.stopPropagation();
  });
</script>
```

Here, the “ *mousedown* ” handlers are registered by both paragraph and button. Upon clicking the button, the handler for the button calls `stopPropagation`, which will prevent the handler on the paragraph from running.

Events can have a default behavior. For example, links navigate to the link’s target upon click, you get navigated to the bottom of a page upon clicking the down arrow, and so on. These default behaviors can be prevented by calling a `preventDefault` method on the event object.

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a");
  link.addEventListener("click", event => {
    console.log("Nope.");
    event.preventDefault();
  });
</script>
```

Here, the default behavior of the link upon click is prevented, i.e. navigating towards the link' target.

Promise, async/await

Imagine you are a popular book writer, and you are planning to release a new book on a certain day. Readers who have an interest in this book are adding this book to their wishlist and are notified when published or even if the release day got postponed too. On the release day, everyone gets notified and can buy the book making all parties happy. This is a real-life analogy that happens in programming.

1. A "*producing code*" is something that takes time and accomplishes something. Here it's a book writer.
2. A "*consuming code*" is someone who consumes the "producing code" once it's ready. In this case, it's a "reader".
3. The linkage between the "*producing code*" and the "*consuming code*" can be called a *promise* as it assures getting the results from the "*producing code*" to the "*consuming code*".

The analogy that we made is also true for the JavaScript `promise` object. The constructor syntax for the `promise` object is:

```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code, "writer")
});
```

Here, a function is passed to `new Promise` also known as the *executor*, and runs automatically upon creation. It contains the producing code that gives the result. `resolve` and `rejects` are the arguments provided by the JavaScript itself and are called one of these upon results.

- `resolve(value):` a callback function that returns `value` upon result
- `reject(error)` : a callback function that returns `error` upon error, it returns an error object

The internal properties of `promise` object returned by the `new Promise` constructor are as follows:

- `state` - initially `pending`, then changes to either `fulfill` upon resolve or `rejected` when `reject` is called
- result - initially `undefined` , then changes to `value` upon `resolve` or `error` when `reject` is called

Promise with resolve and reject callbacks

> One cannot access promise properties: `state` and `result` . Promise methods are needed to handle promises.

Example of a promise.

```
let promiseOne = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1-second signal that the job is done with the result "done"
  setTimeout(() => resolve("done"), 1000);
})

let promiseTwo = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1-second signal that the job is done with the result "error"
  setTimeout(() => reject(new Error("Whoops!")), 1000);
})
```

Here, the `promiseOne` is an example of a "*fulfilled promise*" as it successfully resolves the values, whereas the `promiseTwo` is a "*rejected promise*" as it gets rejected. A promise that is either rejected or resolved is called a *settled* promise, as opposed to an initially *pending* promise. Consuming function from the promise can be registered using the `.then` and `.catch` methods. We can also add `.finally` method for performing cleanup or finalizing after previous methods have been completed.

```
let promiseOne = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promiseOne.then(
  result => alert(result), // shows "done!" after 1 second
  error => alert(error) // doesn't run
);

let promiseTwo = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promiseTwo.then(
  result => alert(result), // doesn't run
  error => alert(error) // shows "Error: Whoops!" after 1 second
);

let promiseThree = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promiseThree.catch(alert); // shows "Error: Whoops!" after 1 second
```

In the `Promise.then()` method, both callback arguments are optional.

Async/Await

With promises, one can use a `async` keyword to declare an asynchronous function that returns a promise whereas the `await` syntax makes JavaScript wait until that promise settles and returns its value. These keywords make promises easier to write. An example of async is shown below.

```
//async function f
async function f() {
  return 1;
}
// promise being resolved
f().then(alert); // 1
```

The above example can be written as follows:

```
function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

`async` ensures that the function returns a promise, and wraps non-promises in it. With `await` , we can make JavaScript wait until the promise is settled with its value returned.

```
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Welcome to Learn JavaScript!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)
  alert(result); // "Welcome to Learn JavaScript!"
}

f();
```

The `await` keyword can only be used inside an `async` function.

Template Literals

Template literals are literals delaminated with backtick (``) and are used in variable and expression interpolation into strings.

```
let text = `Hello World!`;
// template literals with both single and double code inside a single string
let text = `He's often called "Johnny"`;
// template literals with multiline strings
let text =
`The quick
brown fox
jumps over
the lazy dog`;

// template literals with variable interpolation
const firstName = "John";
const lastName = "Doe";

const welcomeText = `Welcome ${firstName}, ${lastName}!`;

// template literals with expression interpolation
const price = 10;
const VAT = 0.25;

const total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;
```

Hoisting

Hosting is a default behavior in JavaScript of moving declarations at the top. While executing a code, it creates a global execution context: creation and execution. In the creation phase, JavaScript moves the variable and function declaration to the top of the page, which is known as hoisting.

```
// variable hoisting
console.log(counter);
let counter = 1; // throws ReferenceError: Cannot access 'counter' before initialization
```

Although the `counter` is present in the heap memory but hasn't been initialized so, it throws an error. This happens because of hoisting, the `counter` variable is hoisted here.

```
// function hoisting
const x = 20,
    y = 10;

let result = add(x,y); // ❌ Uncaught ReferenceError: add is not defined
console.log(result);

let add = (x, y) => x + y;
```

Here, the `add` function is hoisted and initialized with `undefined` in heap memory in the creation phase of the global execution context. Thus, throwing an error.

Currying

`Currying` is an advanced technique in functional programming of transforming a function with multiple arguments into a sequence of nesting functions. It transforms a function from callable `f(a,b,c)` into callable as `f(a)(b)(c)`. It doesn't call a function instead it transforms it.

To get a better understanding of currying let's create a simple `add` function add that takes three arguments and returns the sum of them. Then, we create a `addCurry` function that takes a single input and returns a series of functions with its sum.

```
// Noncurried version
const add = (a, b, c)=>{
    return a+ b + c
}
console.log(add(2, 3, 5)) // 10

// Curried version
const addCurry = (a) => {
    return (b)=>{
        return (c)=>{
            return a+b+c
        }
    }
}
console.log(addCurry(2)(3)(5)) // 10
```

Here, we can see that both the curried and noncurried versions returned the same result. Currying can be beneficial for many reasons, some of which are mentioned here.

- It helps to avoid passing the same variable again and again.
- It divides the function into smaller chunks with a single responsibility, making the function less error-prone.
- It is used in functional programming to create a high-order function.

Polyfills and Transpilers

JavaScript evolves every now and then. Regularly, new language proposals are submitted, analyzed, and added to https://tc39.github.io/ecma262/ and then incorporated into the specification. There may be differences in how it is implemented in JavaScript engines depending on the browser. Some may implement the draft proposals, while others wait until the whole specification is released. Backward compatibility issues arise as new things are introduced.

To support the modern code in old browsers we use two tools: `transpilers` and `polyfills`

Transpilers

It is a program that translates modern code and rewrites it using older syntax constructs so, that the older engine can understand it. For example, " `nullish` coalescing operator" `??` was introduced in 2020, and outdated browsers can't understand it.

Now, it's the transpiler's job to make the `nullish` coalescing operator" `??` understandable to the old browsers.

```
// before running the transpiler
height = height ?? 200;

// after running the transpiler
height = (height !== undefined && height !== null) ? height: 200;
```

> Babel is one of the most prominent transpilers. In the development process, we can use build tools like webpack or parcel to transpile code.

Polyfills

There are times when new functionality isn't available in outdated browser engines. In this case, the code that uses the new functionality won't work. To fill the gaps, we add the missing functionality which is called a " `polyfill` ". For example, the `filter()` method was introduced in ES5 and is not supported in some of the old browsers. This method accepts a function and returns an array containing only the values of the original array for which the function returns `true`

```
const arr = [1, 2, 3, 4, 5, 6];
const filtered = arr.filter((e) => e % 2 === 0); // filter outs the even number
console.log(filtered);

// [2, 4, 6]
```

The polyfill for the filter is.

```
Array.prototype.filter = function (callback) {
  // Store the new array
  const result = [];
  for (let i = 0; i < this.length; i++) {
    // call the callback with the current element, index, and context.
    //if it passes the text then add the element in the new array.
    if (callback(this[i], i, this)) {
      result.push(this[i]);
    }
  }
  //return the array
  return result
}
```

caniuse shows the updated functionality and syntax supported by different browser engines.

Linked List

It is a common data structure found in all programming languages. A Linked List is very similar to a normal array in Javascript, it just acts a little bit differently.

Here each element in the list is a separate object containing a link or a pointer to the next. There is no built-in method or function for Linked Lists in Javascript so one has to implement it. An example of a linked list is shown below.

```
["one", "two", "three", "four"]
```

Types of Linked Lists

There are three different types of linked lists:

1. **Singly Linked Lists:** Each node contains only one pointer to the next node.
2. **Doubly Linked Lists:** There are two pointers at each node, one to the next node and one to the previous node.
3. **Circular Linked Lists:** A circular linked list forms a loop by having the last node pointing to the first node or any other node before it.

Add

The `add` method is created here to add value to a linked list.

```
class Node {
    constructor(data) {
        this.data = data
        this.next = null
    }
}

class LinkedList {
    constructor(head) {
        this.head = head
    }
    append = (value) => {
        const newNode = new Node(value)
        let current = this.head
        if (!this.head) {
            this.head = newNode
            return
        }
        while (current.next) {
            current = current.next
        }
        current.next = newNode
    }
}
```

Pop

Here, a `pop` method is created to remove a value from the linked list.

```
class Node {
    constructor(data) {
        this.data = data
        this.next = null
    }
}

class LinkedList {
    constructor(head) {
        this.head = head
    }
    pop = () => {
        let current = this.head
        while (current.next.next) {
            current = current.next
        }
        current.next = current.next.next
    }
}
```

Prepend

Here, a `prepend` method is created to add a value before the first child of the linked list.

```
class Node {
    constructor(data) {
        this.data = data
        this.next = null
    }
}

class LinkedList {
    constructor(head) {
        this.head = head
    }
    prepend = (value) => {
        const newNode = new Node(value)
        if (!this.head) {
            this.head = newNode
        }
        else {
            newNode.next = this.head
            this.head = newNode
        }
    }
}
```

Shift

Here, the `shift` method is created to remove the first element of the Linked List.

```
class Node {
    constructor(data) {
        this.data = data
        this.next = null
    }
}

class LinkedList {
    constructor(head) {
        this.head = head
    }
    shift = () => {
        this.head = this.head.next
    }
}
```

Global footprint

If you are developing a module, which might be running on a web page, which also runs other modules, then you must beware of the variable name overlapping.

Suppose we are developing a counter module:

```
let myCounter = {
  number: 0,
  plusPlus: function () {
    this.number = this.number + 1;
  },
  isGreaterThanTen: function () {
    return this.number > 10;
  },
};
```

> ***Note:*** this technique is often used with closures, to make the internal state immutable from the outside.

The module now takes only one variable name — `myCounter`. If any other module on the page makes use of such names like `number` or `isGreaterThanTen` then it's perfectly safe because we will not override each other's values.

Debugging

In programming, errors can occur while writing code. It could be due to syntactical or logical errors. The process of finding errors can be time-consuming and tricky and is called code debugging.

Fortunately, most modern browsers come with built-in debuggers. These debuggers can be switched on and off, forcing errors to be reported. It is also possible to set up breakpoints during the execution of code to stop execution and examine variables. For this one has to open a debugging window and place the `debugger` keyword in the JavaScript code. The code execution is stopped in each breakpoint, allowing developers to examine the JavaScript values and, resume the execution of code.

One can also use the `console.log()` method to print the JavaScript values in the debugger window.

```
const a = 5, b = 6;
const c = a + b;
console.log(c);
// Result : c = 11;
```

Console

In JavaScript, we use `console.log()` to write a message (the content of a variable, a given string, etc.) in `console`. It is mainly used for debugging purposes, ideally to leave a trace of the content of variables during the execution of a program.

Example:

```
console.log("Welcome to Learn JavaScript Beginners Edition");
let age = 30;
console.log(age);
```

📝 Tasks:

- [ ] Write a program to print `Hello World` on the console. Feel free to try other things as well!
- [ ] Write a program to print variables to the `console`.
    1. Declare a variable `animal` and assign the dragon value to it.
    2. Print the `animal` variable to the `console`.

💡 Hints:

- Visit the variable chapter to understand more about variables.

Multiplication

In JavaScript, we can perform the multiplication of two numbers by using the asterisk `(*)` arithmetic operators.

Example:

```
let resultingValue = 3 * 2;
```

Here, we stored the product of `3 * 2` into a `resultingValue` variable.

📝 Tasks:

- [ ] Write a program to store the product of `23` times `41` and print its value.

💡 Hints:

- Visit the Basic Operators chapter to understand the mathematical operations.

User Input Variables

In JavaScript, we can take input from users and use it as a variable. One doesn't need to know their value to work with them.

📝 Tasks:

- [ ] Write a program to take input from a user and add `10` to it, and print its result.

💡 Hints:

- The content of a variable is determined by the user's inputs. The `prompt()` method saves the input value as a string.
- You will need to make sure that the string value is converted into an integer for calculations.
- Visit the Basic Operators chapter for the type conversion of `string` to `int`.

Constants

Constants were introduced in ES6(2015), and use a `const` keyword. Variables that are declared with `const` cannot be reassigned or redeclared.

Example:

```
const VERSION = '1.2';
```

📝 Task:

- [ ] Run the program mentioned below and fix the error that you see in the console. Make sure that the code result is `0.9` when it is fixed in the console.

```
const VERSION = '0.7';
VERSION = '0.9';
console.log(VERSION);
```

💡 Hints:

- Visit the Variables chapter for more info about const and also look for "*TypeError assignment to constant variable*" in search engines to learn a fix.

Concatenation

In any programming language, string concatenation simply means appending one or more strings to another string. For example, when strings "*World*" and "*Good Afternoon*" are concatenated with string "*Hello*", they form the string "*Hello World, Good Afternoon*". We can concatenate a string in several ways in JavaScript.

Example:

```
const icon = '👋';

// using template Strings
`hi ${icon}`;

// using join() Method
['hi', icon].join(' ');

// using concat() Method
''.concat('hi ', icon);

//  using + operator
'hi ' + icon;

// RESULT
// hi 👋
```

📝 Task:

- [ ] Write a program to set the values for `str1` and `str2` so the code prints '*Hello World*' to the console.

💡 Hints:

- Visit the concatenation chapter of strings for more info about string concatenation.

Functions

A function is a block of code designed to perform a specific task and executed when "something" invokes it. More info about functions can be found in the functions chapter.

📝 Task:

- [ ] Write a program to create a function named `isOdd` that passes a number `45345` as an argument and determines whether the number is odd or not.
- [ ] Call this function to get the result. The result should be in a boolean format and should return `true` in `console`.

💡 Hints:

- Visit the functions chapter to understand functions and how to create them.

Conditional Statements

Conditional logic is vital in programming as it makes sure that the program works regardless of what data you throw in and also allows branching. This exercise is about the implementation of various conditional logic in real-life problems.

📝 Task:

- [ ] Write a program to create a prompt "*How many km is left to go?*" and based on the user and the following conditions, print the results in the `console`.
  - If there are more than 100 km left to go, print: `"You still have a bit of driving left to go"`.
  - If there are more than 50 km, but less or equal to 100 km, print: `"I'll be there in 5 minutes"`.
  - If there are less than or equal to 50 km, print: `"I'm parking. I'll see you right now"`.

💡 Hints:

- Visit the conditional logic chapter to understand how to use conditional logic and conditional statements.

Objects

Objects are the collection of `key` , `value` pairs and each pair of key-value are known as a property. Here, the property of the `key` can be a `string` whereas its `value` can be of any value.

📝 Tasks:

Given a Doe family that includes two-member, where each member's information is provided in form of an object.

```
let person = {
    name: "John",                    //String
    lastName: "Doe",
    age: 35,                         //Number
    gender: "male",
    luckyNumbers: [ 7, 11, 13, 17], //Array
    significantOther: person2       //Object,
};

let person2 = {
    name: "Jane",
    lastName: "Doe",
    age: 38,
    gender: "female",
    luckyNumbers: [ 2, 4, 6, 8],
    significantOther: person
};

let family = {
    lastName: "Doe",
    members: [person, person2]       //Array of objects
};
```

- [ ] Find a way to print the name of the first member of the Doe family in a `console` .
- [ ] Change the fourth `luckyNumbers` of the second member of the Doe family to `33` .
- [ ] Add a new member to the family by creating a new person (`Jimmy,` `Doe` , `13` , `male` , `[1, 2, 3, 4]` , `null`) and update the member list.
- [ ] Print the `SUM` of the lucky numbers of Doe family in the `console` .

💡 Hints:

- Visit the objects chapter to understand how the object work.
- You can get `luckyNumbers` from each person object inside the family object.
- Once you get each array just loop over it adding every element and then add each sum of the 3 family members.

FizzBuzz Problem

The *FizzBuzz* problem is one of the commonly asked questions, here one has to print *Fizz* and *Buzz* upon some conditions.

📝 Tasks:

- [ ] Write a program to print all the numbers between 1 to 100 in such a way that the following conditions are met.
  - For multiples of 3, instead of the number, print `Fizz`.
  - For multiples of 5, print `Buzz`.
  - For numbers that are multiples of both 3 and 5, print `FizzBuzz`.

```
/
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
....
....
98
Fizz
Buzz
/
```

💡 Hints:

- Visit the loops chapter to understand how the loop works.

Get the Titles!

The *Get the Tiles!* problem is an interesting problem where we have to get the title from a list of books. This is a good exercise for the implementation of arrays and objects.

📝 Tasks:

Given an array of objects that represent books with an author.

```
const books = [
  {
    title: "Eloquent JavaScript, Third Edition",
    author: "Marijn Haverbeke"
  },
  {
    title: "Practical Modern JavaScript",
    author: "Nicolás Bevacqua"
  }
]
```

- [ ] Write a program to create a function `getTheTitles` that takes the array and returns the array of title and print its value in the `console`.

💡 Hints:

- Visit the arrays and objects chapter to understand how the array and object work.