JavaScript

History of JavaScript:

Quickly after the internet was invented, and first two web browsers were developed. Developers wanted to start making websites more interactive. In other words, they needed a programming language for the browser. So, in 1995, the Netscape Navigator, which by the time was the dominant browser hired a guy name Brendan Eich to create the very first version of JavaScript in just 10 days, it was called “Mocha”. So not JavaScript yet, but it already had many of the fundamental features that modern JavaScript has today. Then in 1996, Mocha was renamed to Livescript, which was then again, renamed to JavaScript for one simple reason, to attract developers from the hottest language at the time, which was Java. So, the Java in JavaScript was really for marketing reasons because the language itself has basically nothing to do with Java. JavaScript and Java are two completely different languages. Also in 1996, Microsoft launched the Internet Explorer, which basically copied JavaScript from Netscape, but they called it Jscript for legal reasons because, you actually cannot just go around and copy other people’s programming languages. Now, what this means, is that we now had two very similar but competing languages, which of course is never good idea in the long run. And so, the internet growing like crazy around this time, people realized they needed to standardize JavaScript. So, the language was submitted to an independent standard organization called ECMA, which in 1997 released ECMAScript one or ES1. This was the very first official standards for JavaScript language. And with this, every browser could now implement the same standard "JavaScript." And the real world we usually use the term ECMAScript to refer to the standard, while JavaScript is used when we talk about the language in practice, as it is implemented in browsers. Now fast forward to 2009, after a lot of complications and disagreements about where the language should be headed, ES5 was released with a lot of great new features. And then finally, another six years later, the much-awaited new version ES6 was launched in June, 2015. And this was the single biggest update to the language ever. So, it contained a ton of new exciting features, This was, and still is a really big deal for JavaScript and for the whole web developer community. Now you will also see ES6 being called ES2015, which has actually the official name, but most people just call it ES6. And actually, the reason for ES6 being called ES2015 officially is that in 2015, ECMAScript changed to an annual release cycle. So right now, there's gonna be a new release every single year. The reason for that is that they prefer to just add a small number of new features per year, instead of shipping a huge new version every couple of years, like it happened with ES6. And so, this way, it's gonna be much easier for everyone to keep up to date. And so according to this new annual release cycle, in 2016 ES2016or ES7 was released ES2017 and 2017. And like this, it will continue until the end of time or something like that.

Now there is one particularity about JavaScript releases, which is pretty unique for any programming language and that's backwards compatibility all the way to ES1. So, what does that actually mean? Well, basically it means that if you were to take some JavaScript code written back in 1997 and put it in a modern browser with a modern JavaScript engine today, it would still work just the same. So again, the JavaScript engine that is in our browser today is able to understand old code written 25 years ago, without having to rely on version numbers or anything like that. It just works. And it works this way because of the fundamental principle that is baked into the JavaScript language and its development, which is to not break the web. This means that there is almost never anything removed from the language, but only added in new versions. And actually, we cannot really call them new versions even, because they do not contain breaking changes like when other languages moved to a new version. Instead, new versions are always just incremental updates, which add new stuff. And so, I like to call them releases and not versions. The ECMAScript committee who works on updating the language, does all this. So that old websites basically keep working forever. Just imagine they removed some important feature that made a website from 2008, work just fine. If you then want it to visit that page, it will be broken. And that is why we fortunately have to don't break the web principle. Now, of course, this comes with problems because there are tons of old bugs and weird things and the language. But anyway, these bugs and weird quirks in the language have been giving the language a bad reputation among many programmers who can really take JavaScript serious because of this. But here is the thing, we can actually go around many of this weird stuff by simply learning the modern JavaScript that matters today and just ignore most of the old weird stuff.

Forward compatibility:

so what you think would happen if we took this totally made up code from the year 2089 and try to run it in today's browsers? Well, you are probably right. It would not work at all. There would be errors left and right. And nothing would work. That is why we say that JavaScript is not forwards compatible, basically because current browsers do not understand code from the future.

and there is no forwards compatibility. Right? how we can use modern JavaScript today, we need to consider two distinct scenarios, development, and production. So, the development phase is simply when you're building the site or application on your computer. To ensure you can use the latest JavaScript features in this face. All you have to do is to use the most Up To Date version of the Google Chrome browser. This will then ensure that all the features will work for you as well. The second scenario is production, which is when your web application is finished. You deploy it on the internet and it's then running in your users' browsers. And this is where problems might appear, because this is the part that we actually cannot control. We cannot control which browser the user uses. And we also cannot assume that all our users always use the latest browsers, right. Now, the solution to this problem is to basically convert these modern JavaScript versions back to ES5 using a process called transpiling and also polyfilling. We will use a tool called Babel to transpile or code.

Open chrome developer tools

Ctrl + shift + J

To open a blank page in chrome

Search about:blank

Activating Strict Mood:

Strict mode is a special mode that we can activate in JavaScript, which makes it easier for us to write a secure JavaScript code. And all we have to do to activate strict mode is to write this string at the beginning of the script

```
'use strict';
```

So, with this, we activated strict mode for the entire script. Now what is important is that this line of code, so this statement here basically has to be the very first statement in the script. So, if we have any code before this then strict mode will not be activated. Comments are allowed because JavaScript will just ignore them but no code.

Okay, now we actually can also activate strict mode, only for a specific function or a specific block.

Strict mode makes it easier for us developers to avoid accidental errors. So basically, it helps us introduce the bugs into our code and that's because of 2 reasons. First, strict mode forbids us to do certain things and second, it will actually create visible errors for us in certain situations in which without strict mode JavaScript will simply fail silently without letting us know that we did a mistake.

Values and Variables

Value:

A value is basically the smallest unit of information that we have in JavaScript.

Now, one extremely useful thing that we can do with values is to store them into variables. And so this way we can reuse them over and over again.

Variables:

A variable is something that can be changed. In computer programming we use variables to store information that might change and can be used later in our program.

Variables are containers for storing data (storing data values).

```
let firstName = "Koushik";
```

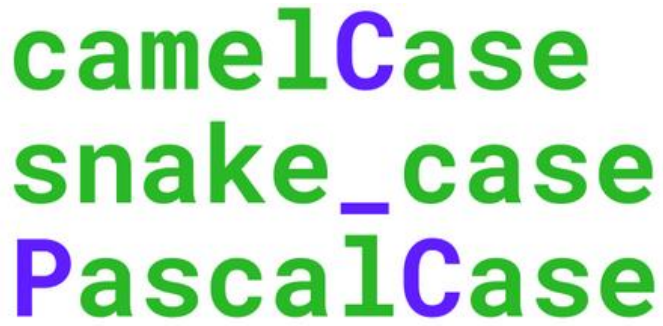
In this example, firstName is a variable, declared with the let keyword:

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter.
- Names can also begin with $ and _ (but we will not use it in this tutorial).
- Names are case sensitive (y and Y are different variables).
- Reserved words (like JavaScript keywords) cannot be used as names.

Naming Conventions:

- Variable and function names written as camelCase.



- Global variables written in UPPERCASE (We do not, but it's quite common)
- Constants (like PI) written in UPPERCASE. variables that are all in uppercase are reserved for constants that we know will never change, like the number PI which is like 3.1415 (so we know that this number is never gonna change.)

- So now another convention is that we should not start a variable name with a uppercase letter. It's a convention, not illegal. It's just that we use this kind of variable names with an uppercase letter for a specific use case in JavaScript, which is object-oriented programming.

- On the same note make sure our variables names are descriptive and that is very important to write cleaner code.

Note: JavaScript identifiers are case-sensitive.

Datatypes

There are eight basic data types in JavaScript.

Seven primitive data types:

- String.

  A string in JavaScript must be surrounded by quotes.

  In JavaScript, there are 3 types of quotes.

```
// double quotes :
 "koushik"
 // single quote :
 'koushik'
 // backticks :
 `koushik`
```

- Number.

  The number type represents both integer and floating point numbers.

```
// With decimals:
let x1 = 26.00;

// Without decimals:
let x2 = 26;
```

- BigInt.

A BigInt value is created by appending `n` to the end of an integer:

```
const bigInt = 1234567890123456789012345678901234567890n;
```

- Boolean.

  The boolean type has only two values: true and false.

  This type is commonly used to store yes/no values: true means “yes, correct”, and false means “no, incorrect”

- Undefined.

  If a variable is declared, but not assigned, then its value is undefined:

```
let age;
```

- Null.

The special null value does not belong to any of the types described above.

It forms a separate type of its own which contains only the null value:

```
let age = null;
```

- Symbol.

  Value that is unique and cannot be changed.

And one non-primitive data type:

- Object.

Declaring Variable

Let:

The let keyword was introduced in ES6 (2015).

Variables defined with let cannot be Redeclared.

```
let firstName = 'Koushik';
 let firstName = 'atanu';
 SyntaxError: Identifier 'firstName' has already been declared
```

It perfectly okay to declare a variable with let at one point in the program, and later assign a new value to it. In technical terms, we call this reassigning a value to a variable, or also we say that we mutate the age variable in this case.

When we need to mutate a variable, that is the perfect use case for using let

```
let age = 26;
age = 27;
```

We also declare empty variables using let. For example, as we did here, where we declared an empty birthYear and then later reassigned that variable to 1996 .

```
let birthYear;
birthYear = 1996;
```

Variables defined with let must be Declared before use.

Variables defined with let have Block Scope.

const:

We use the const keyword to declare variables that are not supposed to change at any point in the future.

The const keyword was introduced in ES6(2015).

Variables defined with const cannot be Redeclared.

```
const birthYear = 1996;
const birthYear = 1997;

SyntaxError: Identifier 'birthYear' has already been declared.
```

Variables defined with const cannot be Reassigned. In technical terms, an immutable variable.

```
const birthYear = 1996;
 birthYear = 1997;

 TypeError: Assignment to constant variable.
```

We cannot declare empty const variables.

```
const job;
SyntaxError: Missing initializer in const declaration.
```

Variables defined with const have Block Scope.

Note: as a best practice for writing clean code, use const by default and let only when you are really sure that the variable needs to change at some point in the future.

var:

var is basically the old way of defining variables, prior to ES6.

And at first sight, it works actually pretty much the same as let.

Variable defined with var can be redeclared.

```
var job = 'programmer';
 var job = 'teacher';
```

We can reassigned the variable using var.

```
var job = 'programmer';
 job = 'teacher';
```

Operators:

Arithmetic operators:

Arithmetic Operators are used to perform arithmetic on numbers:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

```
const ageKoushik = 2023 - 1996;
const ageAtanu = 2023 -1997;
console.log(ageKoushik, ageAtanu);
27 26
```

```
const a = 4;
const b = 2;
console.log(a / 2, a * 2 , 4 ** 2); // 4 ** 2 means 4 to the power of 2. = 4 * 4
2 8 16
```

+ operator:

We can use the plus operator to join ‘strings’, or in another words, to concatenate different ‘strings’.

```
const firstName = 'Koushik';
const lastName = 'Mahapatra';
console.log(firstName + ' ' + lastName);
// ' ' is use to create a string for space and then concatenate it here with these two stings.
Koushik Mahapatra
```

Assignment Operators:

Assignment operators assign values to JavaScript variables.

```
let x = 10 + 5; // 15
```

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

```
let x = 10 + 5; // 15
x += 10; // x = x + 10 = 25
```

Comparison Operators:

Comparison and Logical operators are used to test for true or false.

```
x = 5;
```

Operator	Description	Comparing	Returns
==	equal to	x == 8	false
		x == 5	true
		x == "5"	true
===	equal value and equal type	x === 5	true
		x === "5"	false
!=	not equal	x != 8	true
!==	not equal value or not equal type	x !== 5	false
		x !== "5"	true
		x !== 8	true
>	greater than	x > 8	false
<	less than	x < 8	true
>=	greater than or equal to	x >= 8	false
<=	less than or equal to	x <= 8	true

```
const age = 18;
if (age === 18) console.log('You just become an adult.');
// whenever if block only has one line, we actually don't need two curly braces.
You just become an adult.
```

Don't confuse Assignment (=) which is just a single equal, with the comparison operator, which is this triple equal (===) . Now besides this triple equal, we also have a double equal. The difference is that triple equals is called the strict equality operator. It is strict because it does not perform type coercion. And so it only returns true when both values are exactly the same. On the other hand there's also the loose equality operator, which is only two equals, and loose equality operator actually does type coercion. Let's see that

```
'18' == 18
true
```

double equal does type coercion, so this mean is that this string here is '18' will be converted to a number. Then the number 18 is the same as this number 18.

Note: as a general rule for clean code, avoid the loose equality operator as much you can. So when comparing values always use strict equality (===) .

Logical operators:

Logical operators are used to determine the logic between variables or values.

Operator	Description
&&	and
||	or
!	not

and :

The and operator is written with two ampersands: (&&) . When used with Booleans, and returns true if all operands are true. If one or both operands are falsy, and will return false:

```
const hasDrivingLicense = true; // A
const hasGoodVision = true; // B

console.log(hasDrivingLicense && hasGoodVision);
True
```

```
const hasDrivingLicense = true; // A
const hasGoodVision = false; // B

console.log(hasDrivingLicense && hasGoodVision);
False
```

or :

The or operator is written with two pipes: (| |). When used with booleans, or returns true if at least one operand is true:

```
const hasDrivingLicense = true; // A
const hasGoodVision = false; // B

console.log(hasDrivingLicense || hasGoodVision);
True
```

not :

Logical not is written with a single exclamation point: (!). It accepts one operand. It returns false if the operand can convert to true, otherwise, it returns true:

```
const hasDrivingLicense = false; // A
```

```
const hasGoodVision = false; // B

console.log(!hasDrivingLicense);
True
```

Short Circuit Evaluation

Two important aspects of logical operators in JavaScript is that they evaluate from left to right, and they short-circuit.

With a logical or, if the first operand is true, JavaScript will short-circuit and not even look at the second operand.

The conditional (Ternary) Operator:

The conditional operator allows us to write something similar to an if/else statement but all in one line. The conditional operator is the only JavaScript operator that takes three operands: a condition followed by a question mark (?), then an expression to execute if the condition is truthy followed by a colon (:), and finally the expression to execute if the condition is falsy.

```
const age = 23;
age >= 18 ? console.log('I like to drink wine 🍷') :
console.log('I like to drink water 🥛')
I like to drink wine 🍷
```

The conditional operator is also called the ternary operator. Because it has three parts. For example, the plus (+) operator has only two parts. but this one has three parts, so the condition then the if part, and then the else part.

```
const age = 16;

const drink = age >= 18 ? 'wine 🍷' : 'water 🥛';
console.log(drink);
water 🥛
```

so, drink is now really defined conditionally, based on this condition. And all in one simple line using the conditional operator.

Since the ternary operator is really an expression, we can now use it, for example , in a template literal we cannot insert a normal if/else statement, but using the ternary operator which produces a value, we can actually have conditionals inside of a template literal so let's simply try that.

```
console.log(`I like to drink ${age >= 18 ? 'wine 🍷' : 'water 🥛'}`)
I like to drink water 🥛
```

operator precedence:

```
let x, y;
x = y = 25 - 10 - 5;
// x = y = 10; at this point x = y , and y = 10; then x = 10;
```

```
console.log(x , y);
10 10
```

Level	Operators	Description	Associativity
15	() [] .	Function Call Array Subscript Member Selection	Left to Right
14	++ --	Postfix Increment / Decrement	Right to Left
13	++ -- + - ! ~ (type)	Prefix Increment / Decrement Unary plus / minus Logical negation / bitwise complement Casting	Right to Left
12	* / %	Multiplication Division Modulo	Left to Right
11	+ -	Addition / Subtraction	Left to Right
10	<< >> >>>	Bitwise Left Shift Bitwise Right Shift with sign extension Bitwise Right Shift with zero extension	Left to Right
9	< <= > >= instance of	Relational Less Than / Less than Equal To Relational Greater / Greater than Equal To Type Comparison for objects	Left to Right
8	== !=	Equality Inequality	Left to Right
7	&	Bitwise AND	Left to Right
6	^	Bitwise XOR	Left to Right
5	|	Bitwise OR	Left to Right
4	&&	Logical AND	Left to Right
3	||	Logical OR	Left to Right
2	?:	Conditional Operator	Right to Left
1	= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operators	Right to Left

Template literals:

```
const firstName = 'Koushik';
const job = 'programmer';
const birthYear = 1996;
const currentYear = 2023;

const koushik = "I'm " + firstName + ', a ' + (currentYear - birthYear) + 'years old ' + job + '!!';
console.log(koushik);

"I'm Koushik, a 27 years old programmer !!"
```

A template literal can assemble multiple pieces into one final string.

Let see how this is work.

```
const firstName = 'Koushik';
const job = 'programmer';
const birthYear = 1996;
const currentYear = 2023;


const koushikNew = `I'm ${firstName}, a ${currentYear - birthYear} years old ${job} !!`;
console.log(koushikNew);

"I'm Koushik, a 27 years old programmer !!"
```

We can create multiple line strings using template literals

```
console.log(`multiple
line
string.`);

multiple
line
string.
```

If else statements:

Conditional statements are used to perform different actions based on different conditions.

Use the if statement to specify a block of JavaScript code to be executed if a condition is true.

```
const age = 26;
const isOldEnough = age >= 18;

if (isOldEnough) {
   console.log('Koushik can start driving🚘')
}
Koushik can start driving🚘
```

Use the else statement to specify a block of code to be executed if the condition is false.

```
const age = 16;
const isOldEnough = age >= 18;
```

```
if (isOldEnough) {
    console.log('Koushik can start driving🚘')
} else {
    const yearsLeft = 18 - age;
    console.log(`Koushik is too young. Wait another ${yearsLeft} years.😢`)
}
Koushik is too young. Wait another 2 years.😢
```

Use else if to specify a new condition to test, if the first condition is false

```
const favourite = Number(prompt("What's your favourite number ?"));

if (favourite === 23) {
    console.log('Cool! 23 is an amazing number!')
} else if ( favourite === 7) {
    console.log('Cool! 7 is also a cool number!')
} else {
    console.log('Number is not 23 or 7')
}
```

Control structure *actually controls the flow of execution of a program*.

```
if () {

} else {

}
// it called if else control structure.
```

The switch statement:

Switch statement is an alternative way of writing if/else statement, when all we want to do is to compare one value to multiple different options, basically.

```
const day = 'monday';

switch(day){
    case 'monday': // day === 'monday'
        console.log('Workout day!');
        console.log('Practice JS!');
        break; // When JavaScript reaches a break keyword, it breaks out of the switch block.
    case 'tuesday':
         console.log('Learning CSS!');
        break;
    case 'wednesday':
    case 'thursday':
         console.log('Create projects');
        break;
    case 'friday':
         console.log('Watch coding videos!');
        break;
    case 'saturday':
```

```
    case 'sunday':
        console.log('Enjoy weekend!!');
}
Workout day!
Practice JS!
```

Type conversion and Coercion:

type conversion:

type conversion is when we manually convert from one type to another.

Converting Strings to Numbers:

The global method Number() converts a variable (or a value) into a number.

```
const inputYear = '1996';
console.log(Number(inputYear));
```

A numeric string (like "3.14") converts to a number (like 3.14).

An empty string (like " ") converts to 0.

```
// These will convert:
Number("3.14")
Number(Math.PI)
Number(" ")
Number("")

// These will not convert:
Number("99 88")
Number("John")
```

A non-numeric string (like 'koushik') converts to `NaN` (Not a Number).

```
const x = (Number('koushik')); // JavaScript gives us NAN not a number value whenever an operation that involves numbers fails to produce a new number.
```

Converting Numbers to Strings

The global method String() can convert numbers to strings.

It can be used on any type of numbers, literals, variables, or expressions:

```
console.log(String(23), 23);
'23' 23
```

type coercion:

type coercion is when JavaScript automatically converts types behind the scenes for us.

```
console.log('I am ' + 23 + ' years old');
'I am 23 years old'
```

```
console.log('10' + 5);
'105'
```

In JavaScript, the plus (+) operator that we used here triggers a coercion to strings. And so whenever there is an operation between a string and a number, the number will be converted to a string.

```
console.log('23' - '10' - 3);
10
```

The minus (-) operator that we used here triggers a coercion to numbers. So in this case strings are converted to numbers.

```
console.log('10' * 3, '30' / '3');
30 10
```

Both of them now converted to numbers, because that the only way that the multiplier (*) dividing (/)operator can work.

```
let n = '1' + 1; // n = '11'
n = n - 1;    // n = '11' - 1 = 10
console.log(n);
10
```

Type coercion in boolean values:

```
const money = 0;
if (money){
    console.log("Don't spend it all ;");
} else {
    console.log('You should get a job!')
}
You should get a job!
```

Here JavaScript will try to coerce any value into a Boolean.So, no matter what we put here, if it's not a Boolean, JavaScript will try to convert in to a Boolean. We know that the money here is 0, but 0 is a falsy value. And so , in this logical environment here in this condition, this number 0 will be converted to false. As a result the else block here is executed.

Truthy and Falsy Values:

In JavaScript, there are six falsy values.

- False
- 0
- '' the empty string.
- Undefined.
- Null
- NaN

These are not exactly false initially, but they will become when converted to a Boolean.

Everything else are truthy values.

```
console.log(Boolean(0));
False
console.log(Boolean(undefined));
False
console.log(Boolean('koushik'));
True
console.log(Boolean({})); // empty object is a truthy value

True

console.log(Boolean(''));
False
```

Expressions and Statements:

Expressions:

Expression is a piece of code that produces a value.

```
3 + 4
1996
true && false && !false
these are expressions.
```

Statements:

The statement is like a bigger piece of code that is executed and which does not produce a value on itself.

Basically, we write our whole programs as a sequence of actions. And these actions are statements. Like if/else statements.

```
if ( 23 > 10) {
    const str = '23 is bigger';
}
```

This statement here does not produce a value, all it does is, in this case, it simply declares a variable called str. It performs some actions, it does not produce a value.

Functions:

The fundamental building block of real-world JavaScript applications are functions.

So what actually are functions?

In the simplest form a function is simply a piece of code that we can reuse over and over again in our code. So it's a little bit like a variable but for whole chunks of code. So remember a variable holds value but a function can hold one or more complete lines of code.

A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses **( )**.

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas:
(*parameter1, parameter2, ...*)

The code to be executed, by the function, is placed inside curly brackets: **{ }**

```
function functionName () {
 console.log ('my name is Koushik');
};

// calling / running / invoking function

functionName();

my name is Koushik
```

Function Invocation

The code inside the function will execute when "something" **invokes** (calls) the function:

- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self-invoked)

Function Parameters and Arguments

Parameters are the name listed in the function's definition. (inside the parenthesis apples, oranges are the parameters.)

```
function fruitProcessor (apples, oranges) {

};
```

Function arguments are the real values passed to (and received by) the function.

```
function fruitProcessor(apples, oranges) {
    const juice = `Juice with ${apples} apples and ${oranges} oranges.`;
    return juice;
};

fruitProcessor(5, 0); // these actual values of the parameters are called the arguments.
```

Function Return:

So a function cannot only reuse a piece of code but it can also receive data and return data back.

The return statement ends function execution and specifies a value to be returned to the function caller.

```
function fruitProcessor(apples, oranges) {
```

```
    const juice = `Juice with ${apples} apples and ${oranges} oranges.`;
    return juice;
};

const appleJuice = fruitProcessor(5, 0);
console.log(appleJuice);

Juice with 5 apples and 0 oranges.
```

Function Declarations vs Expressions:

Function Declaration:

- A function declaration also known as a function statement declares a function with a function keyword. The function declaration must have a function name.
- Function declaration does not require a variable assignment as they are standalone constructs and they cannot be nested inside a functional block.
- These are executed before any other code.
- The function in function declaration can be accessed before and after the function definition.

```
// function declaration

function calAge (birthYear) {
    return 2023 - birthYear;
};

const age = calAge(1996);
```

Function Expression:

- A function Expression is similar to a function declaration without the function name.
- Function expressions can be stored in a variable assignment.
- Function expressions load and execute only when the program interpreter reaches the line of code.
- The function in function expression can be accessed only after the function definition.

```
// function expression

const calAge = function (birthYear) {
    return 2023 - birthYear;
};
```

Note: we can actually call function declarations before they are defined in the code. But function expression can not access before initialization or declaration.

Arrow function:

Arrow functions were introduced in ES6. An arrow function is simply a special form of function expression that is shorter and therefor faster to write.

```
// arrow function

const age = birthYear => 2023 - birthYear;

console.log(age(1996));

27
```

```
const yearUntilRetirment = (birthYear, firstName) => {
   const age = 2023 - birthYear;
   const retirement = 60 - age;
   return `${firstName} retires in ${retirement} years.`;
};

 console.log(yearUntilRetirment(1996, 'Koushik'));


Koushik retires in 33 years.

 console.log(yearUntilRetirment(1990, 'Atanu'));

Atanu retires in 27 years.
```

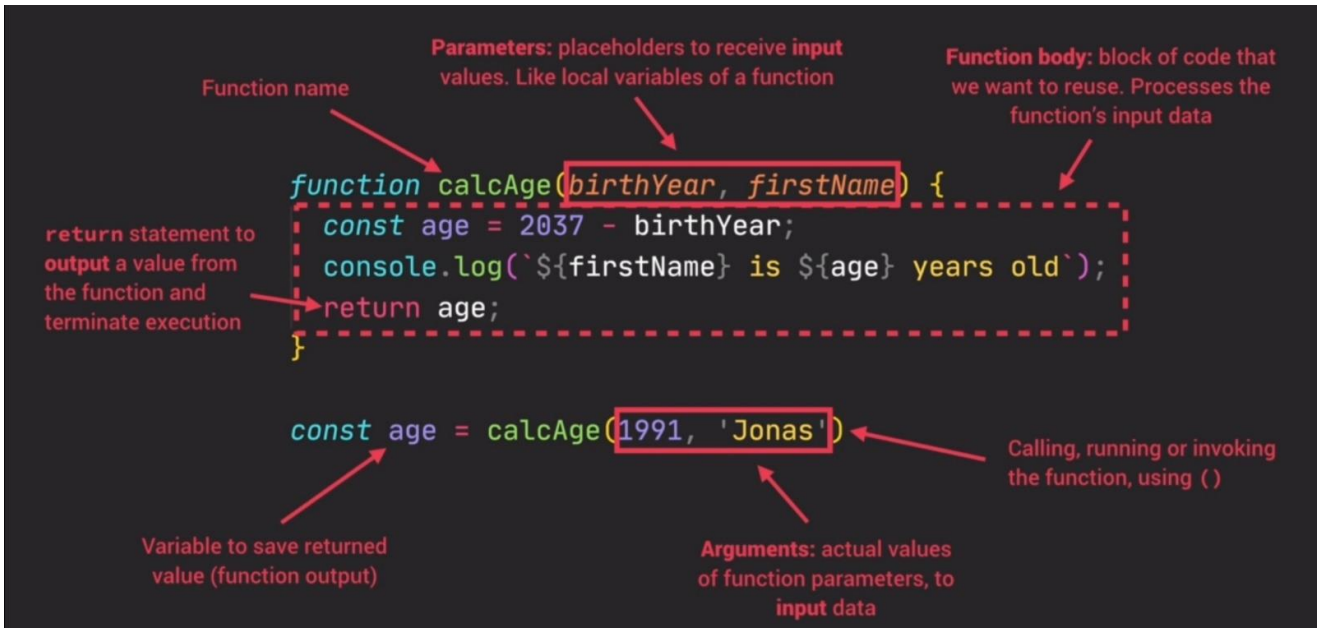
Functions calling other functions:

```
function cutFruitPieces(fruit) {
    return fruit * 4;
};

const fruitProcessor = function (apples, oranges) {
    const applePieces = cutFruitPieces(apples);
    const orangePieces = cutFruitPieces(oranges);

    const juice = `Juice with ${applePieces} pieces of apples and ${orangePieces} pieces of oranges.`;
    return juice;
}

console.log(fruitProcessor(2,3));

Juice with 8 pieces of apples and 12 pieces of oranges.
```

If else statements inside functions:

```
const calAge = function(birthYear) {
    return 2023 - birthYear;
};

const yearUntilRetirment = function (birthYear, firstName) {
    const age = calAge(birthYear);
    const retirement = 60 - age;

    if ( retirement > 0) {
        console.log(`${firstName} retires in ${retirement} years.`)

        return retirement;
    } else {
        console.log(`${firstName} has already retired.🎉`)
        return -1;
    };
};

yearUntilRetirment(1996, 'Koushik');

Koushik retires in 33 years.

yearUntilRetirment(1950, 'Mou');

Mou has already retired.🎉
```

Note: return keyword actually immediately exit the function.



Arrays:

Arrays is like a big container into which we can throw variables and then leter reference them. And that's super important. Because programming is most of the time, all about data. So we get data from somewhere we store and process data and then we give some data back. And that data, it has to go somewhere. So it has to be stored in some place. And for that, we use data structures, just like Arrays.

So in case we have more, than just a single value. The two most important data structures at least in JavaScript, are Arrays and Objects.

```
// How to create arrays

let arr = new Array();

const years = new Array(1991, 1984, 2008 , 2020);
console.log(years);
(4) [1991, 1984, 2008, 2020]

let arr = [];

const friends = ['Micheal', 'Steven', 'Peter'];
console.log(friends);
(3) ['Micheal', 'Steven', 'Peter']
```

Almost all the time, the second syntax is used. We can supply initial elements in the brackets:

```
const friends = ['Micheal', 'Steven', 'Peter'];
```

Array elements are numbered, starting with zero.

We can get an element by its number in square brackets:

```
const friends = ['Micheal', 'Steven', 'Peter'];

console.log(friends[0]); // Micheal
console.log(friends[1]); // Steven
console.log(friends[2]); // Peter
```

The total count of the elements in the array is its length:

```
const friends = ['Micheal', 'Steven', 'Peter'];

console.log(friends.length);

3
```

We can use array length to automatically get the last element of any Array. And that is useful so that we don’t have to count how many elements are in the Array.

```
const friends = ['Micheal', 'Steven', 'Peter'];
```

```
console.log(friends[friends.length - 1]);

Peter
```

We can replace an element:

```
friends[2] = 'Jay';

console.log(friends);

(3) ['Micheal', 'Steven', 'Jay']
```

variables declared with const, cannot be changed. And we did in fact declare the friends variable here with const, right? But I was still able to change one element of the Array here from Peter to Jay, right? So isn't that a contradiction?

Only primitive values, are immutable. But an Array is not a primitive value. And so we can actually always change it so we can mutate it.

Note: We can actually mutate Ararys even though they were declared with const. Now what we can not do is to actually replace the entire Array.

```
friends = ['Bob', 'Aliya'];

Uncaught SyntaxError: Unexpected identifier 'TypeError'
```

An Array can hold values with different types.

```
const firstName = 'Koushik';
const koushik = [firstName, 'Mahapatra', 2023 - 1996, 'student', friends];

console.log(koushik);

(5) ['Koushik', 'Mahapatra', 27, 'student', Array(3)]
```

Computing from an Arrays:

```
const calAge = function(birthYear){
    return 2023 - birthYear;
};

const years = [1990, 1967, 2002, 2010, 2018];

// Calculating age from years array

console.log(calAge(years[0]));

33
console.log(calAge(years[1]));

56
console.log(calAge(years[3]));
```

```
13
console.log(calAge(years[years.length - 1]));

5
```

Basic Array operations (Methods):

push :

- push adds an element to the end.
  push method returns the length of the new Array

```
const friends = ['Micheal', 'Steven', 'Peter'];

friends.push('Jay');

console.log(friends);

(4) ['Micheal', 'Steven', 'Peter', 'Jay']
```

unshift :

- unshift adds an element to the beginning.
  unshift method returns the length of the new Array.

```
const friends = ['Micheal', 'Steven', 'Peter'];

friends.unshift('Jay');

console.log(friends);

(4) ['Jay', 'Micheal', 'Steven', 'Peter']
```

pop :

- pop removes an element from the end.
  Pop returns the removed element.

```
const friends = ['Micheal', 'Steven', 'Peter'];

friends.pop(); // We don't need to pass an argument.

console.log(friends);

(2) ['Micheal', 'Steven']
```

shift :

- shift removes an element from the beginning.
  shift returns the removed element.

```
const friends = ['Micheal', 'Steven', 'Peter'];

friends.shift(); // We don't need to pass an argument.

console.log(friends);

(2) ['Steven', 'Peter']
```

indexOf :

- indexOf tells us in which position a certain element is in the Array.
  indexOf return the index at which that element is located.

```
const friends = ['Micheal', 'Steven', 'Peter'];

console.log(friends.indexOf('Steven')); // indexOf then we need to pass the element which we want to reference.

1
```

If we try indexOf for an element that is not in there, we will get minus one (- 1).

```
const friends = ['Micheal', 'Steven', 'Peter'];

console.log(friends.indexOf('Bob'));

-1
```

includes :

includes is an ES6 method. So includes, simply return true if the element is in the array and false if it is not.

This method actually uses strict equality for this check.

```
const friends = ['Micheal', 'Steven', 'Peter'];

console.log(friends.includes('Steven'));

true
console.log(friends.includes('Bob'));

false
```

we can use the include method to write conditionals.

```
const friends = ['Micheal', 'Steven', 'Peter'];

if (friends.includes('Peter')) {
    console.log('You have a friend called Steven.');
};

You have a friend called Steven.
```

Objects :

There are eight data types in JavaScript. Seven of them are called “primitive”, because their values contain only a single thing (be it a string or a number or whatever).

Objects are used to store keyed collections of various data and more complex entities.

An object can be created with figure brackets `{…}` with an optional list of *properties*. A property is a “key: value” pairs, where `key` is a string (also called a “property name”), and `value` can be anything.

```
const koushik = {   // an object
    firstName: 'Koushik', // by key "firstName" store value 'Koushik'
    lastName: 'Mahapatra', // by key "lastName" store value 'Mahapatra'
    age: 2023 - 1996,
    job: 'developer',
    friends: ['Micheal', 'Peter', 'Steven'] // an array
};
```

Dot notation (.) :

Property values are accessible using the dot notation:

```
const koushik = {
    firstName: 'Koushik',
    lastName: 'Mahapatra',
    age: 2023 - 1996,
    job: 'developer',
    friends: ['Micheal', 'Peter', 'Steven']
};

console.log(koushik.lastName); // if we want to get the lastName

Mahapatra
```

Bracket Notation [ ] :

Property values are accessible using bracket notation:

```
const koushik = {
    firstName: 'Koushik',
    lastName: 'Mahapatra',
    age: 2023 - 1996,
    job: 'developer',
    friends: ['Micheal', 'Peter', 'Steven']
};

console.log(koushik['lastName']); // if we want to get the lastName. have to use 'property name'
```

Note: the big deiffernce between Dot and Bracket notation is that in the bracket notation, we can put any expression that we would like.

```
const koushik = {
    firstName: 'Koushik',
    lastName: 'Mahapatra',
    age: 2023 - 1996,
    job: 'developer',
    friends: ['Micheal', 'Peter', 'Steven']
};

const nameKey = 'Name';

console.log(koushik['first' + nameKey]);

Koushik
console.log(koushik[`last${nameKey}`]);

Mahapatra
```

We can use square brackets in an object literal, when creating an object. That's called *computed properties*.

For instance:

```
const koushik = {
    firstName: 'Koushik',
    lastName: 'Mahapatra',
    age: 2023 - 1996,
    job: 'developer',
    friends: ['Micheal', 'Peter', 'Steven']
};

const interestedIn = prompt('What do you want to know about Koushik ? Choose between firstName, lastName, age, job, and friends');
```

The meaning of a computed property is simple: [interstedIn] means that the property name should be taken from interestedIn.

So, if a visitor enters ‘job’, interstedIn will become job: ‘developer’.

Add new properties to the object:

```
const koushik = {
    firstName: 'Koushik',
    lastName: 'Mahapatra',
    age: 2023 - 1996,
    job: 'developer',
    friends: ['Micheal', 'Peter', 'Steven']
};

koushik.location = 'India'; // using dot notation
koushik['twitter'] = '@kmp007'; // using bracket notation
```

Object Methods:

Any function that is attached to an object is called a method.

```
const koushik = {
    firstName: 'Koushik',
    lastName: 'Mahapatra',
    birthYear: 1996,
    job: 'developer',
    friends: ['Micheal', 'Peter', 'Steven'] ,
    hasDrivingLicense: true,
    calcAge: function(){
        return 2023 - this.birthYear;
    }
};
```

this :

In JavaScript, the this keyword refers to an object.

Which object depends on how this is being invoked (used or called).

The this keyword refers to different objects depending on how it is used:

In the example on top of this page, this refers to the koushik object.

Loops :

Loops are a fundamental aspect of every programming language, because they basically allow us to automate repetitive tasks. So, tasks that we have to perform over and over again.

for loop :

```
for (begin; condition; step) {
   // ... loop body ...
  };
```

```
//we need to use a let variable because this counter will later be updated by the for loop.

// for loop keeps running while condition is TRUE


for (let rep = 1; rep <= 3; rep++){
   console.log(`Lifting weights repetition ${rep} 🏋`);
  };

Lifting weights repetition 1 🏋
Lifting weights repetition 2 🏋
Lifting weights repetition 3 🏋
```

a traditional counter variable name has been i for a long time.

```
const koushik = [
   'Koushik',
   'Mahapatra',
   2023 - 1996,
   'developer',
   ['Micheal','Peter','Steven']
];

for (let i = 1; i <= 4 ; i++) {
   console.log(koushik[i]);
};
```

Looping Arrays

Array is zero based so we need to start it at zero.

```
const koushik = [
   'Koushik',
   'Mahapatra',
   2023 - 1996,
   'developer',
   ['Micheal','Peter','Steven']
];

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

```
};

Koushik
Mahapatra
27
developer
(3) ['Micheal', 'Peter', 'Steven']
```

How to create a new array based on the values of one original array.

So to do that we start by creating a new empty array outside of the loop.(empty array is basically create an array with the usual syntax, but without any element inside of it.) And so now we have to go here to the same loop because this new array types will be based on the koushik array. So it's gonna have the same length. And so we can use the exact same loop that we used to read data from the koushik loop also to construct this new types array.

```
const koushik = [
    'Koushik',
    'Mahapatra',
    2023 - 1996,
    'developer',
    ['Micheal','Peter','Steven']
];

const types = [];

for (let i = 0; i < koushik.length; i++) {
    // Filling types array

    types[i] = typeof koushik[i];

};

console.log(types);

(5) ['string', 'string', 'number', 'string', 'object']
```

We can also use push method to add typeof

```
const koushik = [
    'Koushik',
    'Mahapatra',
    2023 - 1996,
    'developer',
    ['Micheal','Peter','Steven']
];

const types = [];

for (let i = 0; i < koushik.length; i++) {
types.push(typeof koushik[i]);
};

console.log(types);
```

```
(5) ['string', 'string', 'number', 'string', 'object']
```

Calculate the ages for all these four birth years from an Array here using loop.

```
const years = [1991, 2007, 1969, 2020];
const ages = [];

for ( let i = 0; i < years.length; i++){
 ages.push(2023 - years[i]);
};

console.log(ages);

(4) [32, 16, 54, 3]
```

The Continue Statement:

Continue is to exit the current iteration of the loop and continue to the next one. The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
const koushik = [
    'Koushik',
    'Mahapatra',
    2023 - 1996,
    'developer',
    ['Micheal','Peter','Steven']
];


for (let i = 0; i < koushik.length; i++) {
 if (typeof koushik[i] !== 'string') continue;
 console.log(koushik[i]);
};

Koushik
Mahapatra
developer
```

we said here if the type of the current element (typeof koushik[i]), is not a ‘string’ then continue. So again, what we want to do here, is to only log ‘strings’ to the console. Which means that everything else should basically be skipped.

The Break Statement:

Break is used to completely terminate the whole loop not just the current iteration.

```
const koushik = [
    'Koushik',
    'Mahapatra',
    2023 - 1996,
    'developer',
```

```
    ['Micheal','Peter','Steven']
];


for (let i = 0; i < koushik.length; i++) {
 if (typeof koushik[i] === 'number') break;
 console.log(koushik[i]);
};

Koushik
Mahapatra
```

So, what we want to do now, is to log no other elements after we found a number. here the logic is as soon as a number is found, we want to break the loop. So, our if condition here is type of koushik[i] , if it's equal to a number, then break.

Looping backwards:

```
const koushik = [
    'Koushik',
    'Mahapatra',
    2023 - 1996,
    'developer',
    ['Micheal','Peter','Steven']
];

for ( let i = koushik.length - 1; i >= 0; i--) {
console.log(koushik[i]);
};

(3) ['Micheal', 'Peter', 'Steven']
developer
27
Mahapatra
Koushik
```

Here the counter i start from last which is koushik.length - 1. Then the condition when do we want the loop to stop? Well, it should stop after the zero. So, the condition we need to write remember is telling JavaScript in which condition the loop should keep running. And so basically, the loop should keep running as long as the counter is still above zero. So i should be greater or equally 0. So at koushik.length - 1 is in the beginning, i is greater equal zero, so the condition will be true and the next iteration will run, then we need to decrease the index of the counter. And so instead of using i++ , which we used before to increment the value we use minus minus (--) to decrement or to decrease the value by one.

Loop inside a loop:

```
for (let exercise = 1; exercise < 4; exercise++) {
    console.log(`----Starting exercise ${exercise}`);
```

```
    for ( let rep = 1; rep < 4; rep++){
        console.log(`Lifting weight repetition ${rep} 🏋`);
    };
};

----Starting exercise 1
Lifting weight repetition 1 🏋
Lifting weight repetition 2 🏋
Lifting weight repetition 3 🏋
----Starting exercise 2
Lifting weight repetition 1 🏋
Lifting weight repetition 2 🏋
Lifting weight repetition 3 🏋
----Starting exercise 3
Lifting weight repetition 1 🏋
Lifting weight repetition 2 🏋
Lifting weight repetition 3 🏋
```

So, we start exercise number one, which is what we have here and then inside of this iteration of exercise one, a new loop is created and executed. And so that's what then creates these lifting weights repetition from one to three, all right, then this loop here is finished, right? So, it run all the three time that then this first iteration is finished. And so, the exercise loop, so this outer loop here goes to its second iteration.

While Loop :

```
while (condition) {
    // code
    // so-called "loop body"
  };
```

For the while loop we can only specify a condition. So while, and then just a condition is the only thing that we can specify here. It called the while loop because it will run while this condition is true. But here we need to kind of manually, so more explicitly define the other two components of the for loop. So, the repetitions and the increasing of the counter. And so, we need to do that basically outside. So, we start at the beginning with rep equal one, then we have the condition, then let's put the code that we want to execute and then at the end of the iteration, we will then increase the counter.

```
let rep = 1;
while (rep <= 5) {
console.log(`Lifting weights repetition ${rep} 🏋`);
rep++;
};

Lifting weights repetition 1 🏋
Lifting weights repetition 2 🏋
Lifting weights repetition 3 🏋
```

Lifting weights repetition 4
Lifting weights repetition 5

While loop is more versatile than the for loop, which means that it can be used in a larger variety of situations. And that is because it doesn't

really need a counter. So we put the counter here because we need it for this specific use case. But all the while loop really needs is the condition which needs to stay true for it to keep running. And that condition can be any condition, it does not have to be related to any counter at all.

So, whenever we do need a loop without a counter, we can reach for the while loop. When we do know how many times the loop will run, that means we are gonna actually need a counter. For example, when we want to loop over an Array, we already know how many elements that Array has, and so we know how many iterations we will need. And therefore, the for loop is usually the right choice to loop over an Array.

Note: If the condition never becomes false, the loop will never end and this might crash the runtime.