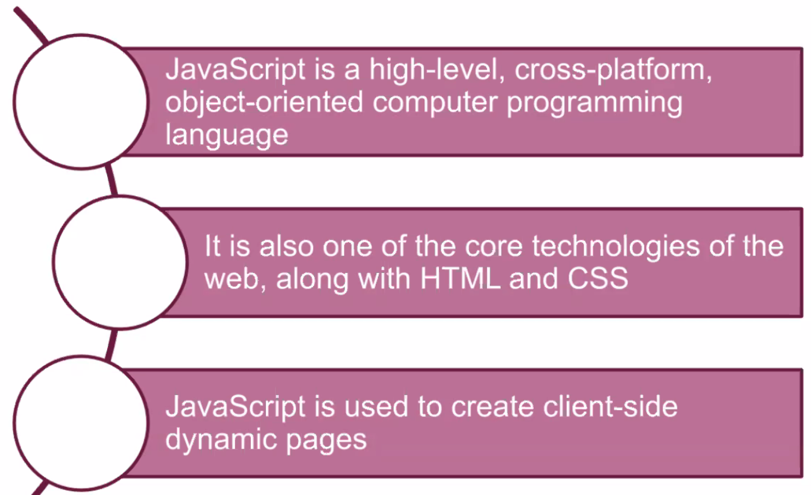


# What is Javascript?



**➤** JavaScript was created by Brendan Eich in 1995 during his time at Netscape Communications. It was inspired by Java, Scheme and Self.



**➤** **JavaScript** is a high-level, cross-platform, object-oriented computer programming language.

**➤** It is also one of the core technologies of the web, along with **HTML** and **CSS**.

**➤** **JavaScript** is the most commonly used to create client-side dynamic pages.

**➤** Today, **JavaScript** can execute not only in the browser, but also on the server, or actually on any device that has a special program called the JavaScript engine.

**⚜️ Why is it called *Java*Script?:**

When JavaScript was created, it initially had another name: **“LiveScript”**. But Java was very popular at that time, so it was decided that positioning a new language as a “younger brother” of Java would help.  
But as it evolved, JavaScript became a fully independent language with its own specification called **ECMAScript**, and now it has no relation to Java at all.

## Placement

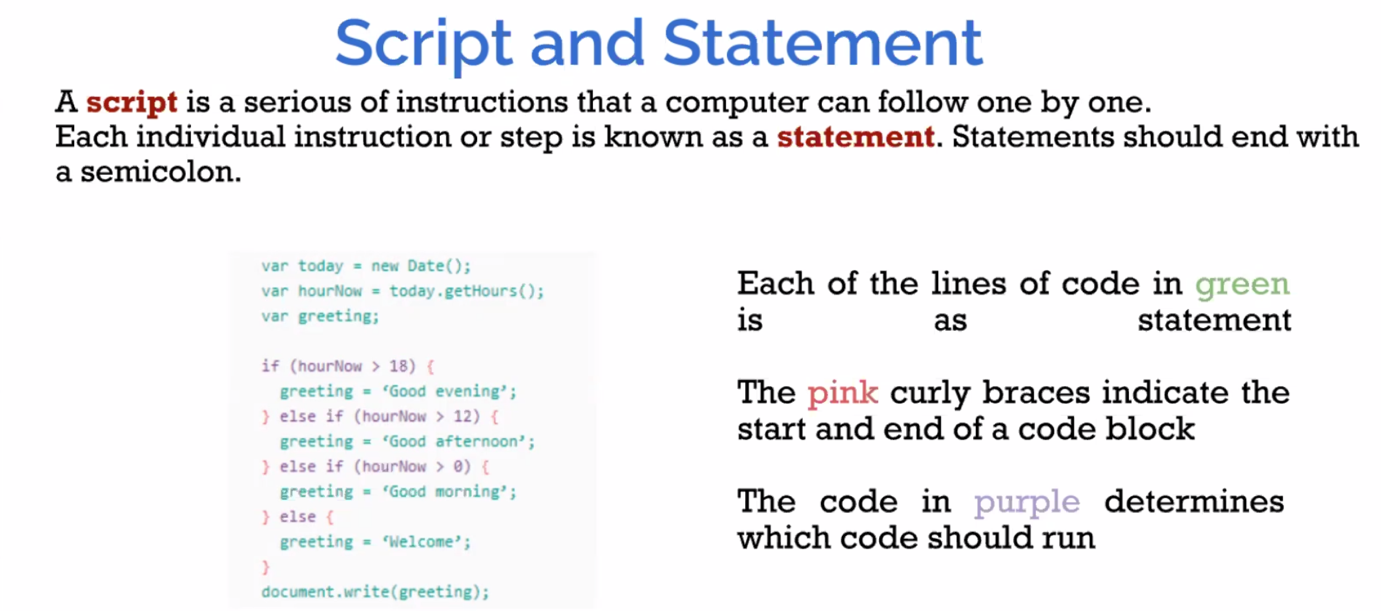
**➤** A **script** is a serious of instructions that a computer can follow one by one.

**➤** Each individual instruction or step is known as a **statement**.  **Statements** should end with a **semicolon**.

**➤** In **HTML**, **JavaScript** code must be inserted between <script> and </script> tags.

**➤** You can place any number of scripts in an **HTML** document.

**➤** **Script** tags can be inserted in the **HTML** page's <body>, or <head> sections, or in both.



## External JavaScript

**➤** **Scripts** can be located in **external files**.

**➤** External **scripts** are useful when the same code is used on many different web pages.

**➤** To include an external **JavaScript** file in your **HTML** code, it is necessary to put the name of the **script** file in the **src** attribute of a **<script>** tag.

**➤** An external **script** reference can be located in the **<head>** or **<body>** section of an **HTML** page.

**➤** The file extension of **JavaScript** files is **.js**.

<script src = "myScript.js" ></script>

## Output

There are different ways to **"display"** data in **JavaScript**.

**➤** document.write().  
**➤** window.alert().  
**➤** console.log().

## Comments

**●** a single line comments after a double slash //  
**●** and multi-line comments between /\* ....... \*/

## ECMAScript 2015 (ES6)

**➤** **ECMAScript (ES)** is a scripting language specification intended to standardize **JavaScript**.

**➤** **ECMAScript 2015**, which is also known as **ECMAScript 6** and **ES6**, is the sixth edition of the **ECMAScript Language Specification** standard.

**➤** **ES6** provides a new set of features and fixes to **JavaScript**.

**★** We’re going to dive into what' difference between ES6 and before its.

**⚡ Commonly Asked Questions:**

* How do I get JavaScript?
* Where can I download JavaScript?
* Is JavaScript Free?

You don't have to get or download JavaScript.  
JavaScript is already running in your browser on your computer, on your tablet, and on your smart-phone.  
Free to use for everyone.

# JS Variables

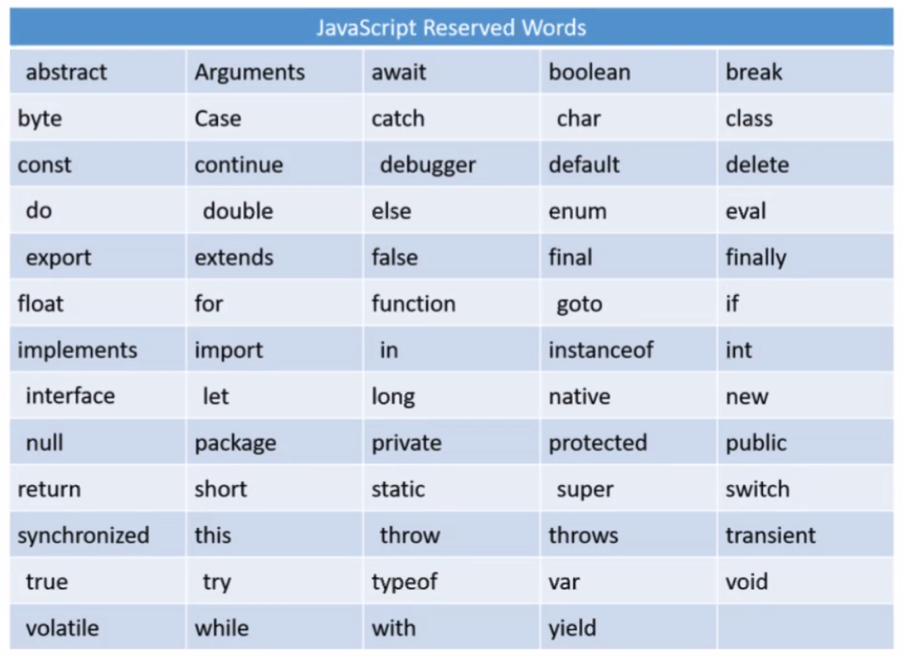
## Naming Rules

The general rules for defining names for **variables**:

**➤** Names can composed of letters, digits, underscores, and dollar signs.  
**➤** Numbers are not allowed as the first character.  
**➤** The first character must be;  
   **●**a letter  
   **●**an underscore ( **\_** )  
   **●**a dollar sign (**$**)

**➤** JavaScript names must not contain spaces, mathematical or logical operators.

**➤** Reserved words cannot be used as names



## JavaScript let and const

**➤** Before **ES6** we used to define a variable using the **var** keyboard.

**➤** **let** and **const** keyboards are added to **JavaScript** with **ES6**.

### let

**➤** The let statement enables you to declare a variable with **block scope**.

**➤** Scope is the fundamental concept that defines a variable's visibility in all programming languages.

Example

var a = 10;

{

let b = 3;

}

console.log ("a = " + a);

console.log ("b = " + b); //generates an error

Output:

a = 10  
ReferenceError: b is not defined

### **const**

**const** variables are similar to **let** variables, except that **const** variables are **immutable.** They are not allowed to be reassigned.

Example

const x = 5;

x = 7; //generates an error

Output:

TypeError: Assignment to constant variable

# JS Data Types

## Data Types

**➤** JavaScript variables can hold numerous data types, such as numbers, strings, booleans, and more.

**➤** It is required to declare a variable for creating.

**➤** In JavaScript, we can declare a variable with the var keyword.

var myNumber;

**➤** The latest ECMAScript standard defines [nine types:](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures)

**➤** Six Data Types that are primitives, checked by typeof operator:

* **Undefined** : typeof instance === "undefined"
* **Boolean** : typeof instance === "boolean"
* **Number**  : typeof instance === "number"
* **String**  : typeof instance === "string"
* **BigInt** : typeof instance === "bigint"
* **Symbol**  : typeof instance === "symbol"

**➤** **null** : typeof instance === "object". Special primitive type having additional usage for its value: if object is not inherited, then null is shown;

**➤** **Object** : typeof instance === "object". Special non-data but structural type for any constructed object instance also used as data structures: new Object, new Array, new Map, new Set, new WeakMap, new WeakSet, new Date and almost everything made with new keyword;

**➤** **Function** non data structure, though it also answers for typeof operator: typeof instance === "function".

This answer is done as a special shorthand for Functions, though every Function constructor is derived from Object constructor.

|  |  |
| --- | --- |
| **console.log ("1. " + typeof 0);**  **console.log ("2. " + typeof (3,14));**  **console.log ("3. " + typeof "hello");**  **console.log ("4. " + typeof (7+8));**  **console.log ("5. " + typeof "Oliver");**  **console.log ("6. " + typeof "");**  **console.log ("7. " + typeof true);**  **console.log ("8. " + typeof (3>=2));**  **var a; // Value is undefined.**  **console.log ("9. " + typeof a);** | 1. **number** 2. **number** 3. **string** 4. **number** 5. **string** 6. **string** 7. **boolean** 8. **Boolean** 9. **undefined** |

### Numbers

**➤** Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc. JavaScript has only one type of **number**.

**➤** **Numbers** can be written with decimals.

var x = 7;

var y = 33.33;

**Precision**

**➤** Integers (numbers without a period or exponent notation) are accurate up to 15 digits.

var x = 999999999999999;

var y = 9999999999999999;

console.log(x); // output = 999999999999999

console.log(y); // output = 10000000000000000

**➤** The maximum number of decimals is 17, but floating point arithmetic is not always 100% accurate:

var x = 0.2 + 0.1;

console.log(x); // output = 0.30000000000000004

**➤** To solve the problem above, it helps to multiply and divide:

var x = 0.2 + 0.1;

console.log(x); // output = 0.30000000000000004

var y = (0.2\*10 + 0.1\*10) / 10;

console.log(y); // output = 0.3

### BigInt

**➤** The **BigInt** type is a numeric primitive in JavaScript that can represent integers with arbitrary precision. With BigInts, you can safely store and operate on large integers even beyond the **safe integer** limit for Numbers.

**➤** A **BigInt** is created by appending n to the end of an integer or by calling the constructor.

**➤** You can obtain the safest value that can be incremented with Numbers by using the constant **Number.MAX\_SAFE\_INTEGER**. With the introduction of BigInts, you can operate with numbers beyond the **Number.MAX\_SAFE\_INTEGER**.

> const x = 2n \*\* 53n;

// 9007199254740992n

> const y = x + 1n;

// 9007199254740993n

**➤** You can use the operators +, \*, -, \*\*, and % with BigInts—just like with Numbers. A BigInt is not strictly equal to a Number, but it is loosely so.

**➤** A **BigInt** behaves like a Number in cases where it is converted to Boolean: if, ||, &&, Boolean, !.

**➤** **BigInts** cannot be operated on interchangeably with Numbers. Instead, a **TypeError** will be thrown.

### Symbol

**➤** **Symbol** is a primitive value.

**➤** A value having the data type Symbol can be referred to as a "Symbol value". In a JavaScript runtime environment, a symbol value is created by invoking the function Symbol, which dynamically produces an anonymous, unique value. A symbol may be used as an object property.

**➤** **Symbol** can have an optional description, but for debugging purposes only.

**➤** A Symbol value represents a unique identifier. For example:

// Here are two symbols with the same description:

let Sym1 = Symbol("Sym")

let Sym2 = Symbol("Sym")

console.log(Sym1 === Sym2) // returns "false"

**➤** **Symbol** type is a new feature in ECMAScript 2015. There is no ECMAScript 5 equivalent for Symbol.

**➤** In some programming languages, the **symbol** data type is referred to as an "**atom**."

### Strings

**➤** Scripts are defined as the sequence of characters.  
**➤** In **JavaScript**, strings are used for storing and manipulating text.  
**➤** A string can be any content that shows up inside **quotes**. We can use **single** or **double-quotes**.

var myName = "John";

var myAge = '36';

**➤** We can use quotes inside a string unless they don't match the quotes encompassing the string.

var text1 = "I'm John";

var text2 = 'She said, "Go ahead"'; // output : She said, "Go ahead"

**➤** If we need to use quotes inside the string, we can use

**backslash ( \)** escape character.

var text = "He said, \"I am a new programmer.\"";

// output : He said, "I am a new programmer."

**Breaking Long Code Lines**

**➤** For best readability, programmers often like to avoid code lines longer than 80 characters.  
**➤** If a JavaScript statement does not fit on one line, the best place to break it is after an operator:

document.getElementById("demo").innerHTML = // Break a code line after

"Hello Dolly!"; // an operator or comma

### 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;

**➤** In JavaScript, null is not a “reference to a non-existing object” or a “null pointer” like in some other languages.

**➤** It’s just a special value that represents “nothing”, “empty” or “value unknown”.

**➤** The code above states that **age** is unknown.

### Undefined

**➤** The special value **undefined** also stands apart. It makes a type of its own, just like null.

**➤** The meaning of **undefined** is “value is not assigned”.

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

let age;

console.log(age); // output: "undefined"

### Booleans

**➤** A **JavaScript Boolean** can have one of two values, either true or false.

**➤** **Boolean** data type is useful when we need to have one of two values, such as true/false, yes/no, etc.

**The Boolean() Function**

**➤** You can use the Boolean() function to find out if an expression (or a variable) is true:

Boolean (3 > 2); //return true

Boolean (2 < 3); //return true

Boolean (2 > 3); //return false

Boolean (2 == 3); //return false

**➤** **The Boolean value of everything with a "value" is true.**

Boolean("Hello World");

Boolean(5);

Boolean(-5);

Boolean("false");

Boolean(3.14 + 8); //All of these values return true

**➤** The Boolean value of everything without a "value" is false.

**➤** The Boolean value of 0, null, empty string, undefined, is false.

Boolean("");

Boolean(0);

Boolean(-0);

Boolean(null);

var x;

Boolean(x); //All of these values return false

### Objects

**➤** **Objects** in JavaScript, just as in many other programming languages, can be compared to objects in real life. The concept of objects in JavaScript can be understood with real life, tangible objects.

**➤** In JavaScript, an **object** is a standalone entity, with properties and type. Compare it with a cup, for example. A cup is an object, with properties. A cup has a color, a design, weight, a material it is made of, etc. In the same way, JavaScript objects can have properties, which define their characteristics.

**Object properties**

**➤** A **JavaScript object** has properties associated with it. A property of an object can be explained as a variable that is attached to the object.

var myCar = new Object();

myCar.make = 'Ford';

myCar.model = 'Mustang';

myCar.year = 1969;

**➤** The above example could also be written using an object initializer, which is a comma-delimited list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({}):

var myCar = {

make: 'Ford',

model: 'Mustang',

year: 1969

};

**➤** Properties of JavaScript objects can also be accessed or set using a bracket notation

myCar['make'] = 'Ford';

myCar.model = "Mustang";

# JS Strings

## String Methods and Properties

**➤** Primitive values, like "Edward Benedict", cannot have properties or **methods** (because they are not objects).

**➤** But with **JavaScript**, **methods** and **properties** are also available to primitive values, because JavaScript treats primitive values as objects when executing methods and properties.

### length Property

**➤** length property returns the count of the total number of characters.

**➤** The length of an empty string is 0

var x = "Clarusway";

var y = "";

console.log (x.length); //output: 9

console.log (y.length); //output: 0

### concat() Method

**➤** The concat() method is used to combine two or more strings.

**➤** This method **does not modify** the current strings but returns a new string that contains the joined strings text.

var s1 = "Hello ";

var s2 = "World!";

var s3 = s1.concat(s2);

console.log(s3) ; //output: Hello World!

**➤** The concat() method can be used instead of the **plus**(+) operator.

### charAt() Method

**➤** The charAt() method returns the char value at the specified index in a string.

var x = "Clarusway";

console.log (x.charAt(0)); //output: C

console.log (x.charAt(3) ); //output: r

**❗ Warning :** The index of the first character is 0 and the index of the last character is string.length-1.

### includes() Method

**➤** The includes() method specifies whether a string includes the characters of a specified string.

**➤** This method returns true if the characters are in the string, and if not false.

var s = "Hello John, welcome to Clarusway.";

var n = s.includes("welcome");

console.log (n); // output: true

**❗ Warning :** The includes() method is **case sensitive**.

### indexOf() and lastIIndexOf() Method

**➤** The indexOf() method returns the **index** of (the position of) the **first** occurrence of a specified text in a string:

**➤** This method returns **-1** if the value is **not found**.

var s = "Hello John, welcome to Clarusway.";

var n = s.indexOf("welcome");

console.log (n); // output: 12

**➤** You can find second or other letters in the string with second parameter of indexOf() method. Find the first occurrence of the letter "e" in a string, **starting the search at position 5**:

var s = "Hello John, welcome to Clarusway.";

var n = s.indexOf("e", 5);

console.log (n); //output : 13

### lastIndexOf() Method

**➤** The lastIndexOf() method returns the **index** of the **last** occurrence of a specified text in a string:

var s = "Hello John, welcome to Clarusway. How are you John?";

var n = s.lastIndexOf("John");

console.log (n); //output : 46

### replace() Method

**➤** The replace() method looks for a string for a **given value** and returns a **new string** to **replace** the specified values.

var s = "Hello John, welcome to Clarusway.";

var rep = s.replace("Clarusway", "Course")

console.log (rep); output : Hello John, welcome to Course.

**❗ Warning :**  
The replace() method does not change the string it is called on. It returns a **new string**.  
  
By default, the replace() method replaces **only** the **first match** and the replace() method is **case sensitive**.

**➤** To replace **case insensitive**, use a **regular expression** with an /i flag (insensitive):

var str = "Hello John, welcome to Clarusway.";

var rep = str.replace(/JOHN/i, "Edward")

console.log (rep); //output: Hello Edward, welcome to Clarusway.

### search() Method

**➤** The search() method searches a string for a given value and returns the **position** of the match.

**➤** This method returns -1 if the value is not found.

var s = "Hello John, welcome to Clarusway.";

console.log (s.search("Clarusway"));// output: 23

**❗ Warning :**  
The two methods, indexOf() and search(), are **equal**? **NOT** equal.

* The search() method cannot take a second start position argument.
* The indexOf() method cannot take powerful search values (regular expressions).

### slice() Method

**➤** The slice() method **extracts a section** of a string and returns it as a **new string**.

**➤** We can use a **negative** number to select from the **end** of the string.

|  |  |
| --- | --- |
| **var s = "Hello John, welcome to Clarusway.";**  **console.log (s.slice(0, 5));**  **console.log (s.slice(6, 10));**  **console.log (s.slice(12));**  **console.log (s.slice(12, -10));** | **Hello John welcome to Clarusway. welcome to** |

**➤** If you dont use the **second parameter**, the method will slice out the **rest** of the string:

|  |  |
| --- | --- |
| **var s = "Hello John, welcome to Clarusway.";**  **var rest = s.slice(12);**  **console.log(rest)** | **welcome to Clarusway.** |

### split() Method

**➤** The split() method **divides** a string into an **array** of substrings, and returns the **new array**.

**➤** The split() method **does not alter** the original string.

var s = "Hello John, welcome to Clarusway."

var arr = s.split(" ");

console.log (arr); // output : ["Hello", "John,", "welcome", "to", "Clarusway."]

**➤** If the **separator** is omitted, the returned array will contain the whole string in **index [0]**.

**➤** If the separator is "", the returned **array** will be an array of **single characters**.

### substr() Method

**➤** The substr() method returns the **parts** of a string, **beginning** at the character at the specified position and a **number** of characters after it.

**➤** The substr() method **does not alter** the original string.

var s = "Hello John, welcome to Clarusway."

console.log (s.substr(23, 9)) ;// output: Clarusway

### substring() Method

**➤** The substring() method returns the **parts** of a string between "start" and "end", **not including "end"** itself.

**➤** The substring() method **does not alter** the original string.

var s = "Welcome to Clarusway."

console.log (s.substring(5, 10)); // output : me to

console.log (s.substring(5, 1)); // output : elco

**➤** If you omit the second parameter, substring() will slice out the **rest** of the string.

Differences between slice() and substring()

|  |  |
| --- | --- |
| **What they have in common:**   1. If start equals stop: returns an empty string 2. If stop is omitted: extracts characters to the end of the string   If either argument is greater than the string's length, the string's length will be used instead. | |
| **Distinctions of** [substring()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/substring)**:**   1. If start > stop, then substring will swap those 2 arguments. 2. If either argument is negative or is NaN, it is treated as if it were 0. | **Distinctions of** [slice()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/slice)**:**   1. If start > stop, slice() will return the empty string. ("") 2. If start is negative: sets char from the end of string, exactly like substr() in Firefox. This behavior is observed in both Firefox and IE. |

### toLowerCase() Method

**➤** The toLowerCase() method transforms a string to **lowercase** letters.

**➤** The toLowerCase() method **does not alter** the original string.

var s = "WELCOME TO CLARUSWAY";

console.log(s.toLowerCase()); // output : welcome to clarusway

### toUpperCase() Method

**➤** The toUpperCase() method transforms a string to **uppercase** letters.

**➤** The toUpperCase() method **does not alter** the original string.

var s = "Welcome to Clarusway"

console.log(s.toUpperCase());// output : WELCOME TO CLARUSWAY

### trim() Method

**➤** The trim() method **eliminates** whitespace from both sides of a string.

**➤** The trim() method **does not alter** the original string.

var s = " Welcome to Clarusway "

console.log (s. trim()); // output : Welcome to Clarusway

## Template literals ``[[1]](#footnote-1)

Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them.

They were called "template strings" in prior editions of the ES2015 specification.

`string text`

`string text line 1

string text line 2`

`string text ${*expression*} string text`

*tag*`string text ${*expression*} string text`

Template literals can contain placeholders. These are indicated by the dollar sign and curly braces (${*expression*}). The expressions in the placeholders and the text between the backticks (` `) get passed to a function.

[Click for more info](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals)

## JavaScript String Escape / Unescape

Escapes or unescapes a JavaScript string removing traces of offending characters that could prevent interpretation.

The following characters are reserved in JavaScript and must be properly escaped to be used in strings:

* **Horizontal Tab** is replaced with **\t**
* **Vertical Tab** is replaced with **\v**
* **Nul char** is replaced with **\0**
* **Backspace** is replaced with **\b**
* **Form feed** is replaced with **\f**
* **Newline** is replaced with **\n**
* **Carriage return** is replaced with **\r**
* **Single quote** is replaced with **\'**
* **Double quote** is replaced with **\"**
* **Backslash** is replaced with **\\**

# Operators

**➤** Let's take a simple 3 + 2 phrase equals 5. Number 3 and 2 are **operands** and ‘+’ is the **operator**.

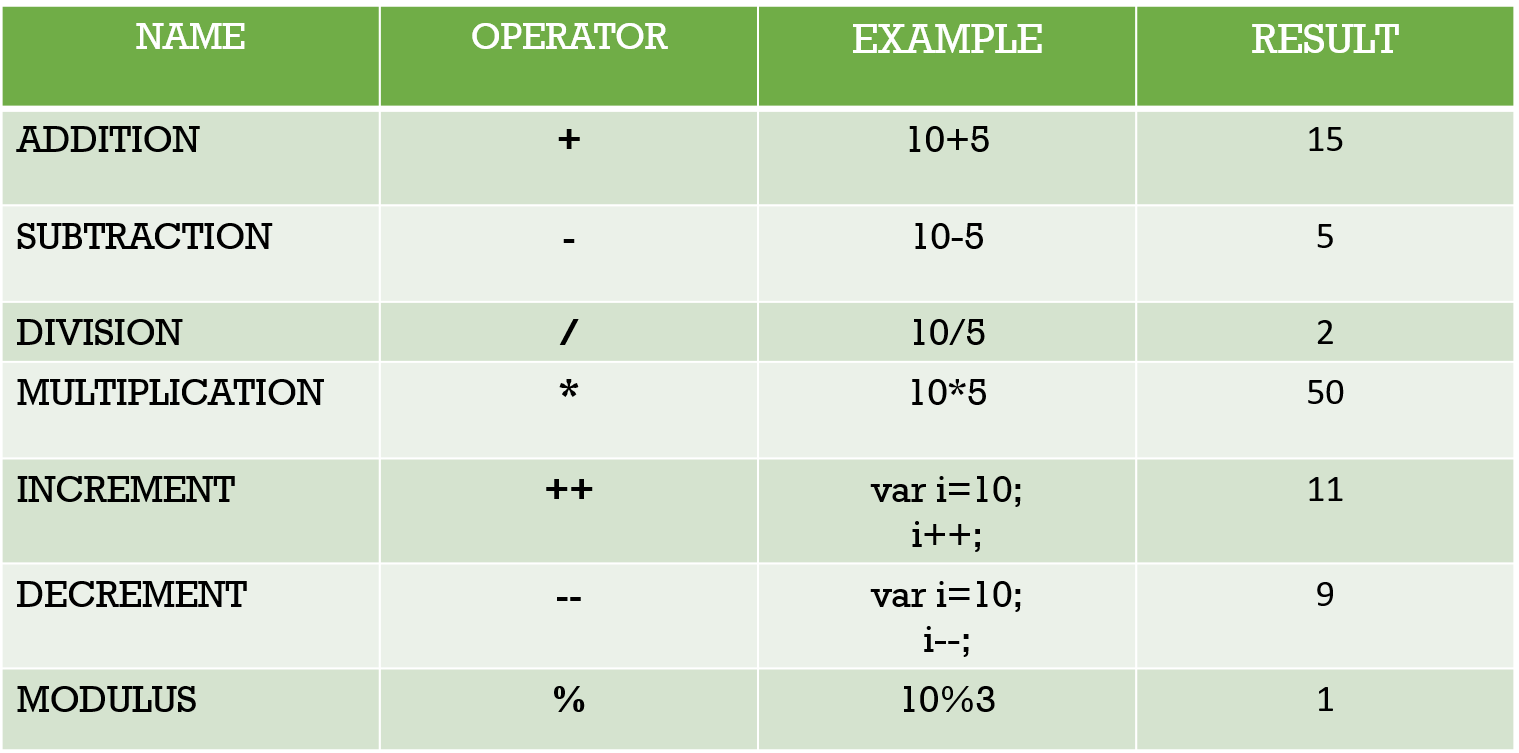
**➤** Expressions rely on **operators** to create a single value from one or more values.

**➤** JavaScript supports the **operators** of the following types.

**●** Arithmetic Operators  
**●** Assignment Operators  
**●** Comparison Operators  
**●** Logical Operators

## Arithmetic Operators

**➤** **Arithmetic operators** execute arithmetic functions on numbers (literals or variables).

****

var a = 20;

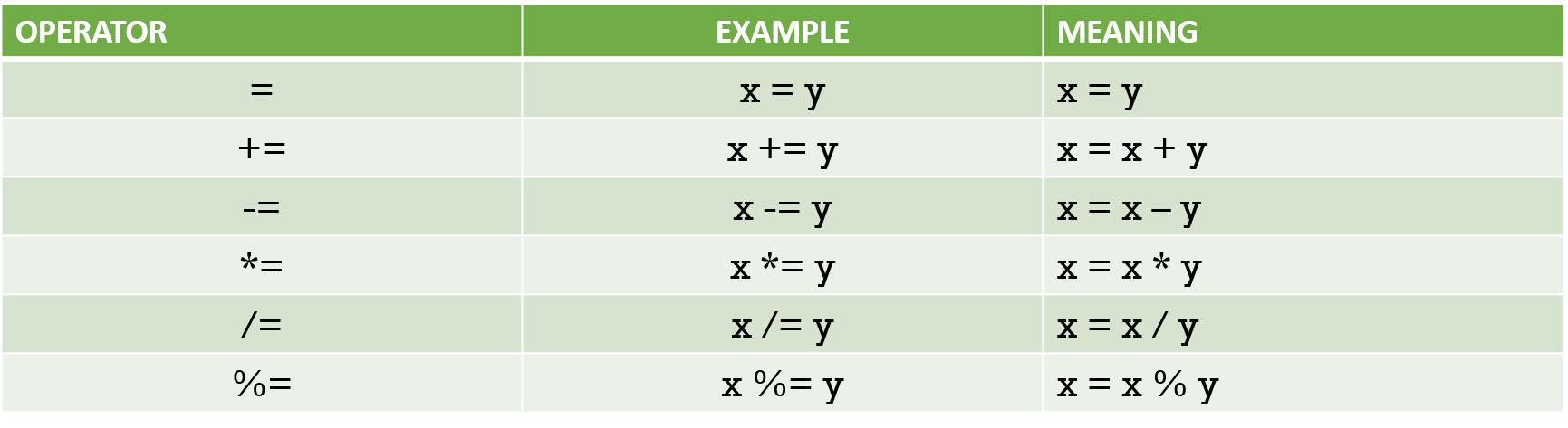
var b = 3;

var c = a \* b;

console.log (c); //output: 60

## Assignment Operators

**➤** **Assignment operators** assign values to JavaScript variables.



var a = 20;

a -= 5;

console.log (a); // 15

var a = 20;

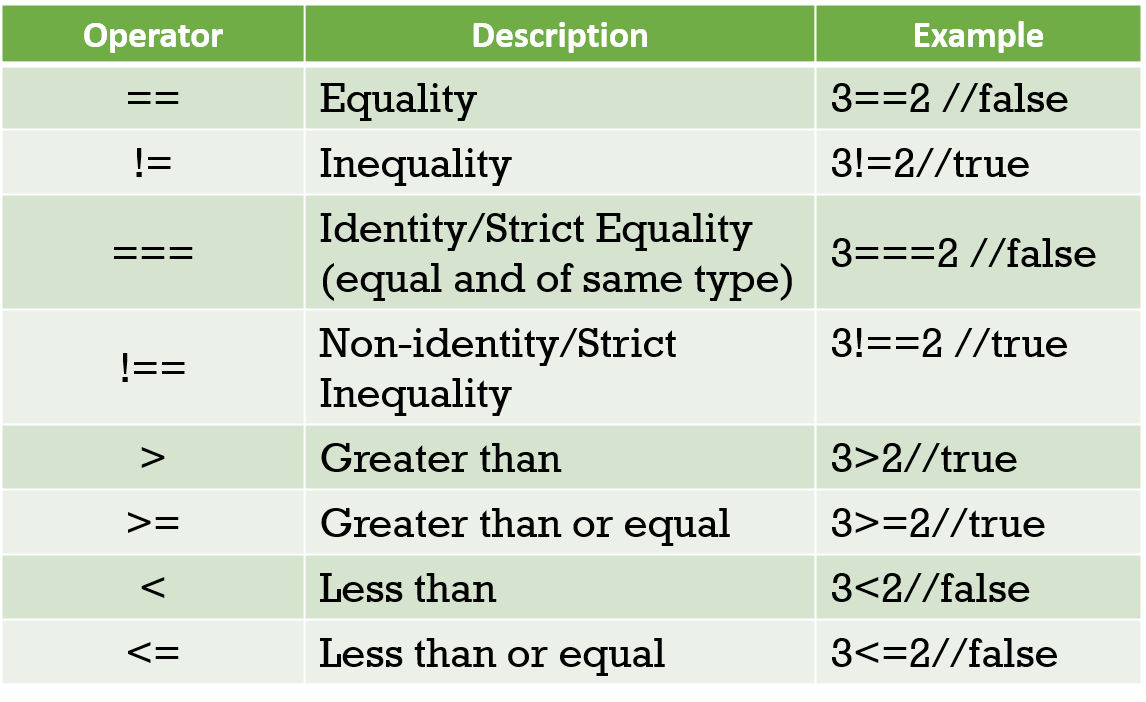
a %= 3;

console.log (a); //2

## Comparison Operators

**➤** **Comparison operators** are used to determine equality or difference between variables or values in logical statements.

**➤** All comparison operators return **Boolean** (true or false).



var a = "10";

console.log (a == 10);

// true

var a = "10";

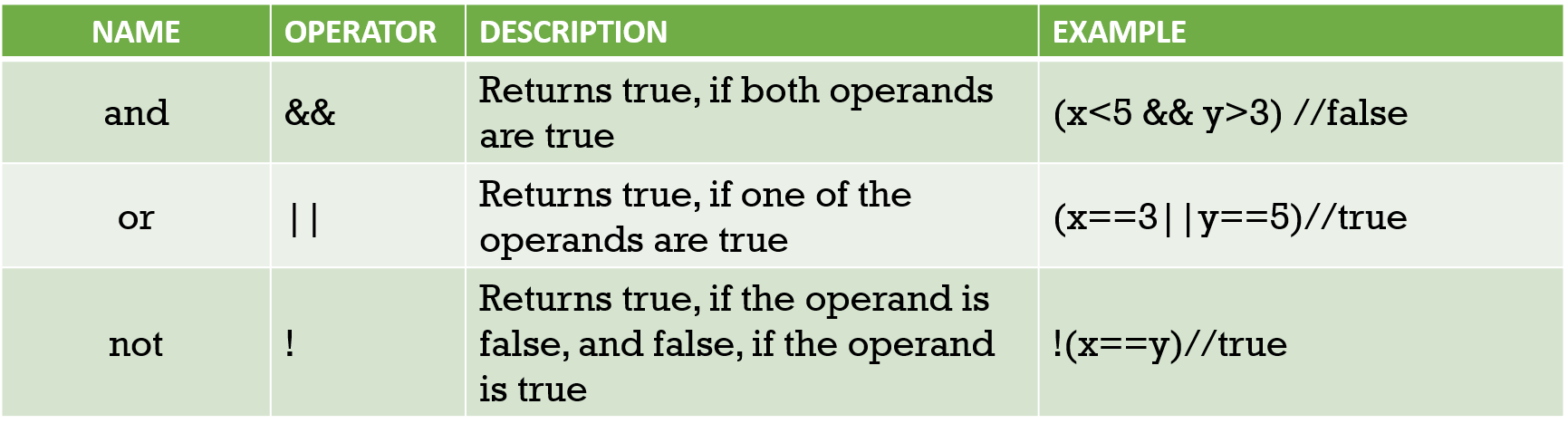
console.log (a === 10);

// false

## Logical Operators

**➤** Logical operators, also known as **Boolean Operators**, are used to determine the logic between variables or values and return true or false.

**➤** Seeing as x = 3 and y = 2, logical operators are explained in the table below:



var a = 10;

var b= 5;

console.log (a>=b && b<=20 );

// true

var a = 10;

var b= 5;

console.log (a==b || b<=20 );

// true

var a = 10;

var b= 5;

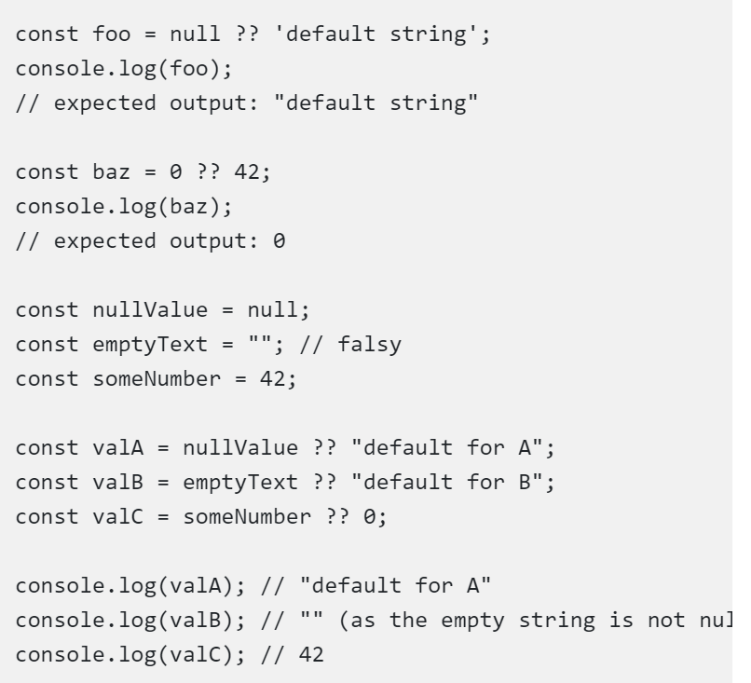
console.log (!(a>=b ));

// false

### Nullish coalescing operator (??)

**➤** The **nullish coalescing operator (??)** is a logical operator that returns its right-hand side operand when its left-hand side operand is **null** or **undefined**, and otherwise returns its left-hand side operand.

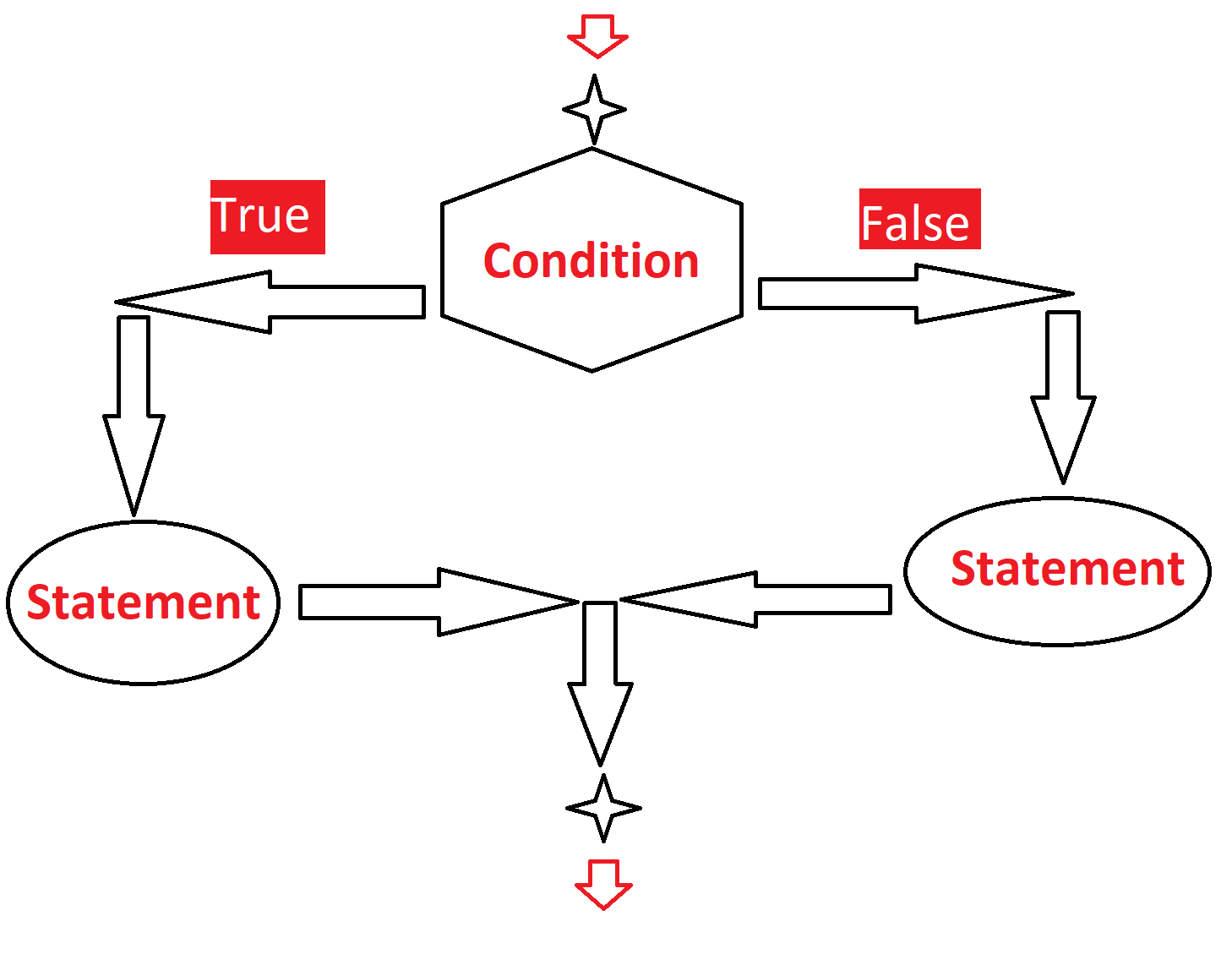
**➤** Contrary to the logical **OR** (**||**) operator, the left operand is returned if it is a falsy value that is not **null** or **undefined**. In other words, if you use **||** to provide some default value to another variable **foo**, you may encounter unexpected behaviors if you consider some falsy values as usable (eg. **''** or **0**). See below for more examples.



# Conditionals

## Conditional Statements

**➤** Based on different conditions, **conditional statements** are used to decide the **execution flow**. If a condition is true, one action can be performed and you can perform another action if the condition is false.



**➤** You want to perform different actions for different decisions very often when you write code.

**➤** You can use the code's **conditional statements** to do this.

**➤** The following **conditional statements** are available in **JavaScript**:

1. **If Statement:** If a specified condition is true, Use if to specify a code block to be executed
2. **else:** If the same condition is false, Use else to specify a code block to be executed
3. **else if:** If the first condition is false, Use else if to specify a new condition to test
4. **switch:** compare a value with multiple variants.

## If Statement

 If a condition is valid, use the **if statement** to define a block of JavaScript code to execute.

if (condition) {

// Statement (if the condition is true, statement to be execute)

}

var x = 3;

if (x == 3) {

console.log(x); //(the condition is true and statement to be execute)

}

// 3

## Else Statement

**➤** If the condition is false, use the other statement to define a block of code to be executed.

var time = 12;

if (time < 8) {

console.log("Good morning.");

} else {

console.log("Good day.");

}

## if…else if…else Statement

**➤** If the first condition is false, use the other if the statement defines a new condition.

var time = 21.00;

if (time < 8) {

console.log("Good morning.");

} else if (time < 18) {

console.log("Good afternoon.");

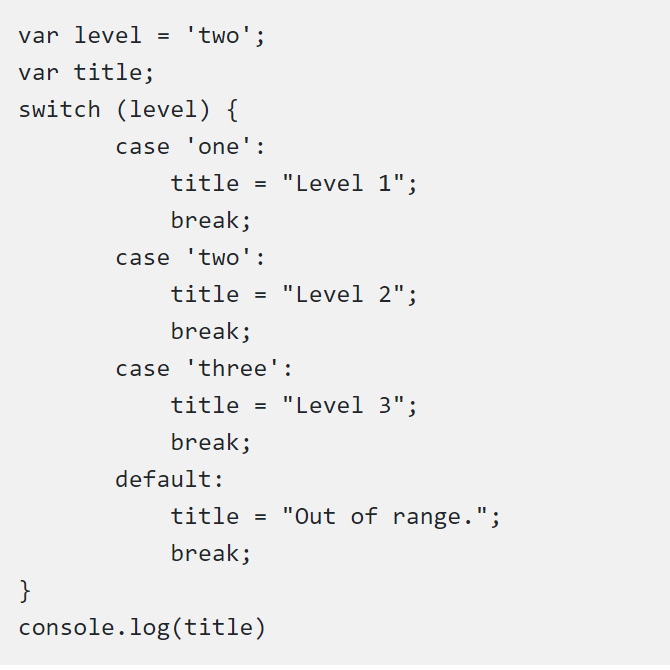
} else {

console.log("Good evening.");

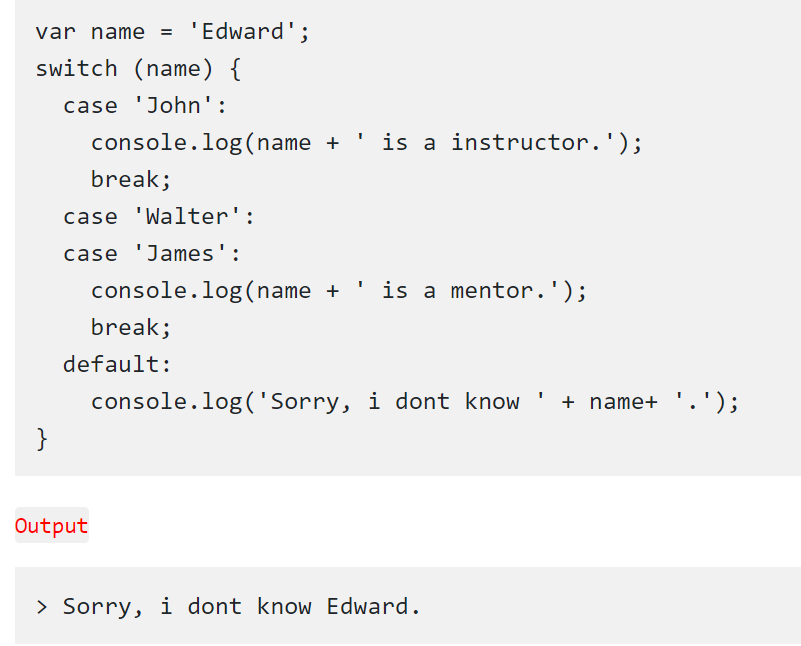
} // Good evening

## Switch Statements

**➤** **Switch case statement** is used to compare the value of a variable with multiple values and execute some statements based on the match.



Level 2



**⚜️ Tips:**

If you omit the break statement, the next case will be executed even if the evaluation does not match the case. But the last default break is **not necessary** to break the last case in a **switch block**. The block breaks (ends) there anyway.

**⚜️ Tips:**

Switch cases use **strict** comparison (===). Because of that, the values must be of the **same type** to match.

## Ternary

It is a one-line if statement

*condition* ? *exprIfTrue* : *exprIfFalse*

const point = 55;

point < 50 ? console.log("BAD") : console.log("GOOD");

Good

const result = point < 50 ? "BAD" : "GOOD";

console.log(result);

Good

## Nullish

It is for checking whether a variable is Null or not.

*leftExpr* ?? *rightExpr*

const variable\_1 = 60;

// const result\_1 = variable\_1 != null ? variable\_1 : 80;

const result\_1 = variable\_1 ?? 80;

console.log(result\_1);

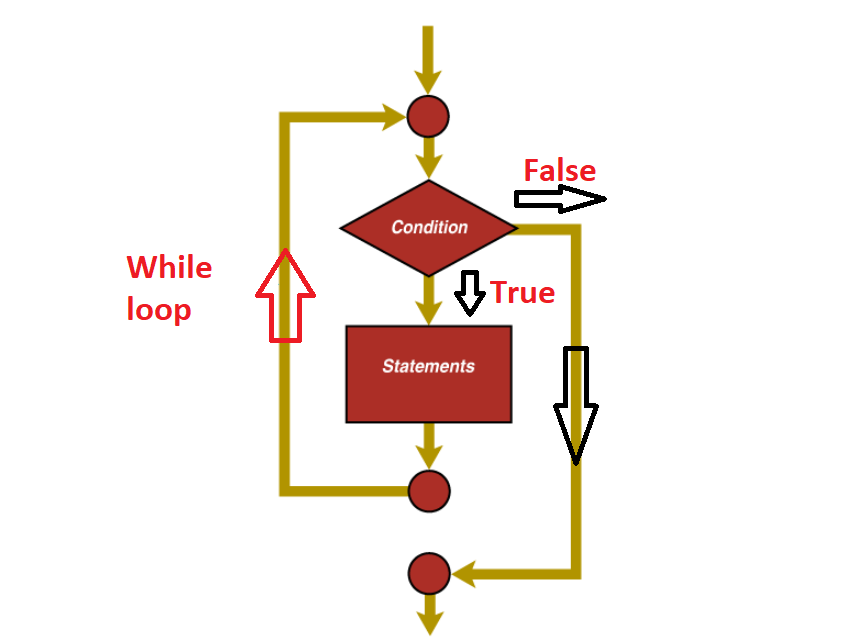
# Loops

**➤** **JavaScript Loop** provides a **fast** and **easy** way to perform repetitive tasks. They offer only a few lines of code to perform iterations. Iteration is the number of times the function is to be performed (this number may even be **zero**). Make sure you understand all of them so that the best **loop** statement can be applied for a given situation.

**➤** The following **loop statements** are given by **JavaScript** to accomplish this:

**●** while statement  
**●** for statement  
**●** for…in statement  
**●** for…of statement  
**●** do..while statement  
**●** break statement  
**●** continue statement

## while Statement



**➤** A while statement executes its statements as long as it is true for a specified condition.

while (condition) {

// statement

}

**➤** If the condition is **incorrect**, the statement will stop executing within the **loop** and the power will pass to the statement after the loop.

**➤** The condition check takes place before the assertion is executed in the loop. If the result is **correct**, the assertion will be executed and the condition will be re-tested. If the condition returns false, the execution will stop and the control will be passed on to the following statement while.

var i = 0;

var sum = 0;

while (i < 4) {

var sum = sum + i;

i++;

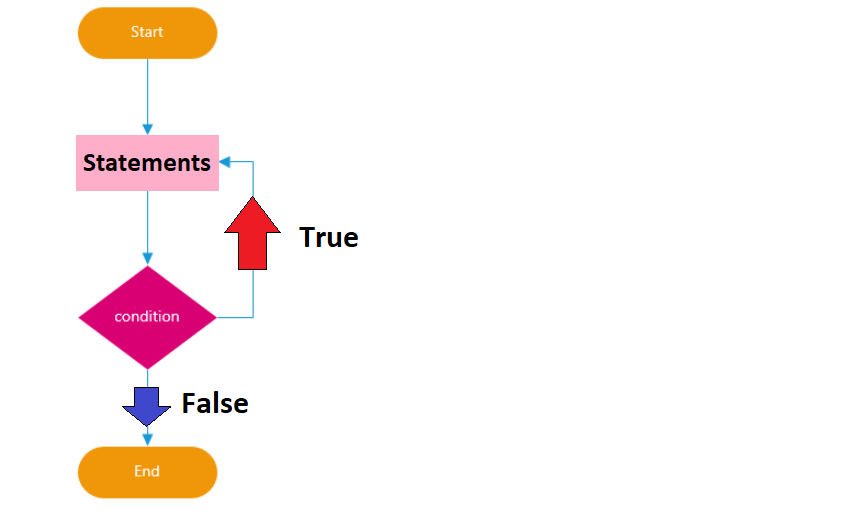
}

console.log ("The total is: " + sum); //6



## do...while Statement

**➤** The do...while statement repeats until false is evaluated by the specified condition.  
**➤** The difference between while and do-while loops is that code will execute **at least once** in the **do-while loop**.



do {

statement

} while (condition);

var i = 6;

var sum=0;

do {

var sum = sum + i;

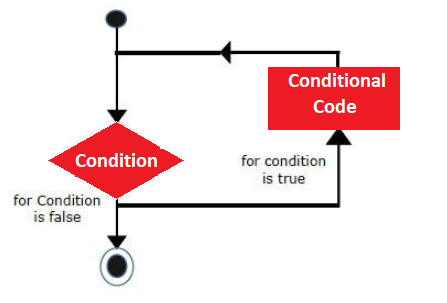
i++;

}

while (i<6);

console.log ("The total is " + sum); //6

## for Statement



var sum=0;

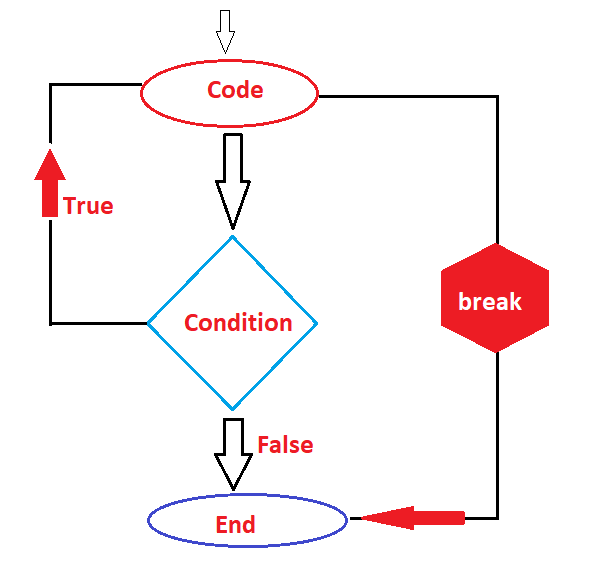
for (var i = 0; i < 6; i++) {

sum += i;

}

console.log ("The total is " + sum); //15

## break Statement



**➤** To terminate a loop, switch or in conjunction with a label statement, use the break statement. ("**jumps out**" of a loop.)

var i = 0;

while (i < 10) {

if (i === 3) {

break;

}

console.log(i)

i = i + 1;

}

console.log(i);

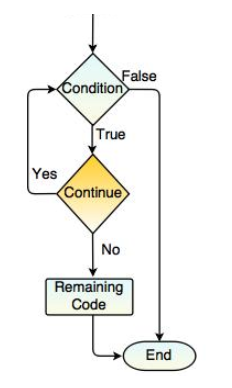
0

1

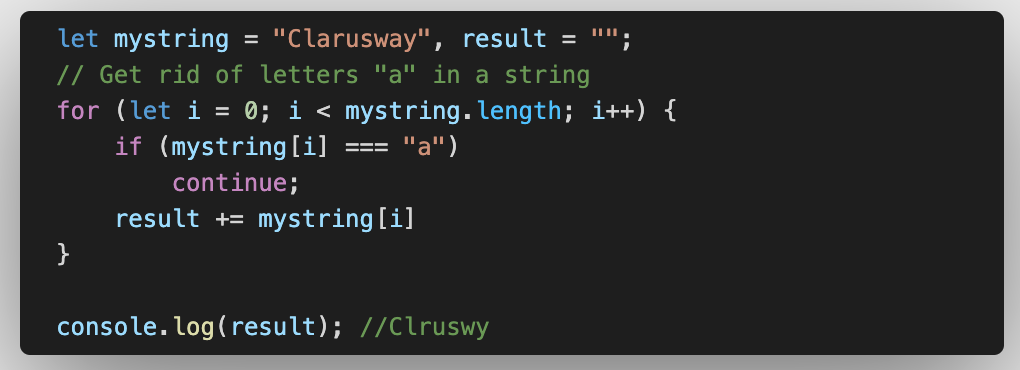
2

3

## continue Statement



**➤** If a specified condition occurs, the continuation clause **breaks** one iteration (in the loop) and **continues** with the next iteration in the loop.



# Arrays

**➤** The **array** is a single variable in JavaScript that is used to store various elements. It is often used when a list of elements is stored and accessed by a single variable. Unlike most languages where the array is a multiple variable relations, there is a single variable in JavaScript array that stores multiple elements.

var colors = ["Red", "Blue", "Yellow"];

**Spaces and line breaks are not important**. A declaration can span multiple lines.

var colors = new Array("Red", "Blue", "Yellow");

* The two examples above do exactly the same. There is no need to use new Array(). For simplicity, readability and execution speed, use the first one (the array literal method).

**➤** **Arrays** are a particular type of object. The JavaScript type operator returns **"object"** for arrays.

**➤** However, it is better to define **JavaScript** arrays like arrays.

**➤** **Objects** are using names to control their **"members"**.

var name = {firstName:"John", lastName:"James", age:24};

## length property

**➤** The length property returns the **number** of elements.

var name = ["John" , "James" , "Oliver" , "Aaron" , "Scott"];

var x = name.length; // the length of name is 5

## concat() method

**➤** The concat() method is used to combine **two or more** arrays.

**➤** This method **does not modify** the current arrays but returns a **new array** that contains the elements of the joined arrays.

var dogs = ["Bulldog" , "Beagle" , "Rottweiler"];

var cats = ["Ragdoll" , "Sphynx" , "Birman"];

var pets = dogs.concat(cats);

console.log (pets);

// ["Bulldog", "Beagle", "Rottweiler", "Ragdoll", "Sphynx", "Birman"]

## sort() method

**➤** The sort() method **sorts** arrays **alphabetically**.

//Changes the original array but not the ones compised of numbers

{

let myArray = ["izmir" , "istanbul" , "ankara" , "Corum" , "Kars"];

const result = myArray.sort();

console.log(myArray);

console.log(result);

}

let num = ["23","198","34","3","9"];

let result = num.sort();

console.log(num);

console.log(result);

//['198', '23', '3', '34', '9']

**❗ Warning :**  
By default, the sort() function sorts values as **strings**. If numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1". Because of this, the sort() method will produce **incorrect** result when sorting numbers.

## push() method

**➤** The push() method **adds** a new element to the **end** of an array.

let myArray = ["izmir" , "istanbul" , "ankara" , "Corum" , "Kars"];

let result = myArray.push("Van");

console.log(myArray);

console.log(result);

console.log(myArray);

['izmir', 'istanbul', 'ankara', 'Corum', 'Kars', 'Van']

6

['izmir', 'istanbul', 'ankara', 'Corum', 'Kars', 'Van']

## pop() method

**➤** The pop() method allows you to **remove** the **last** element from an array.

//removes an alement from the end

{

let myArray = ["izmir" , "istanbul" , "ankara" , "Corum" , "Kars"];

console.log(myArray);

let result = myArray.pop();

console.log(result);

console.log(myArray);

}

['izmir', 'istanbul', 'ankara', 'Corum', 'Kars']

Kars

['izmir', 'istanbul', 'ankara', 'Corum']

## shift method()

**➤** The shift() method **removes** the **first** element from an array and returns the string that removed element.

//removes an alement from the beginning

{

let myArray = ["izmir" , "istanbul" , "ankara" , "Corum" , "Kars"];

console.log(myArray);

let result = myArray.shift();

console.log(result);

console.log(myArray);

}

['izmir', 'istanbul', 'ankara', 'Corum', 'Kars']

izmir

['istanbul', 'ankara', 'Corum', 'Kars']

## unshift method()

**➤** The unshift() method **adds** a new element to the **beginning** of an array and returns the new array length.

//Adds an alement to the beginning

{

let myArray = ["izmir" , "istanbul" , "ankara" , "Corum" , "Kars"];

console.log(myArray);

let result = myArray.unshift("Rize");

console.log(result);

console.log(myArray);

}

['izmir', 'istanbul', 'ankara', 'Corum', 'Kars']

6

['Rize', 'izmir', 'istanbul', 'ankara', 'Corum', 'Kars']

## splice() Method

**➤** The splice() method is used to **add** new elements to an array and **delete** elements from an array.

**➤** The **first** parameter (1) determines the position of the first element to **delete** and starting position to **insert**.

**➤** The **second** parameter (2) determines the number of elements to **delete**.

**➤** The remaining parameters ("White", "Pink") determines the **new** elements to be **added**.

## slice() Method

**➤** The slice() method **slices** a piece of an array into a **new array**.

The original array will **not** be modified.

var colors = ["Red" ,"Yellow", "Green", "Blue"];

var newColors = colors.slice(1, 3);

console.log(colors); //["Red" ,"Yellow", "Green", "Blue"]

console.log(newColors); // ["Yellow", "Green"]

## indexOf() Method

**➤** The indexOf() method returns the **position** of a specified element in an array at the **first** occurrence

**➤** This method returns -1 if the element is **not found**.

## lastIndexOf() Method

**➤** The lastIndexOf() method returns the **last index** of a specified element in an array and returns -1 if the element is **not found**.  
**➤** In this method, the search **begins** at the started position, or at the **end** if no starting position is specified, and the search ends at the beginning of the array.

let myArray = ["ankara", "izmir" , "ankara", "istanbul" , "corum" , "ankara", "kars"];

let first\_index = myArray.indexOf("ankara");

console.log(first\_index); //0

let declared\_index = myArray.indexOf("ankara", 1); //find "ankara"s index after 1st index

console.log(declared\_index); //2

let last\_index = myArray.lastIndexOf("ankara");

console.log(last\_index); //5

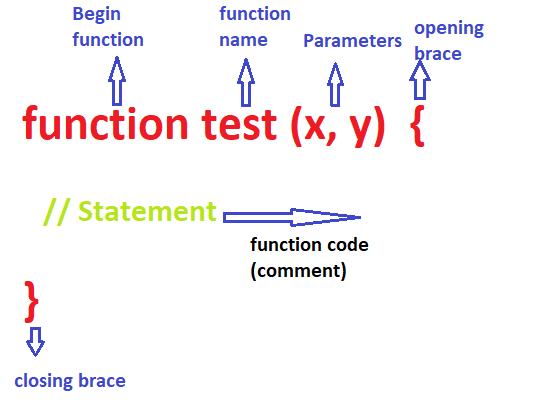
# Functions

**➤** A JavaScript feature is a block of code that is programmed to perform a specific task.

function name(parameters){

statements

}



function square(number) {

return number \* number;

}

**➤** The function square takes one parameter, called number. The function consists of one statement that says to return the parameter of the function (that is, number) multiplied by itself. The statement return specifies the value returned by the **function**.

**➤** Primitive parameters (such as a number) are transferred to functions by value; the value is passed to the function, but this adjustment is **not expressed globally** or in the calling function if the function changes the value of the parameter.

## Calling a Function

**➤** The code within the function will be executed when the function is **invoked** with just one line of code.

function square(number) {

return number \* number;

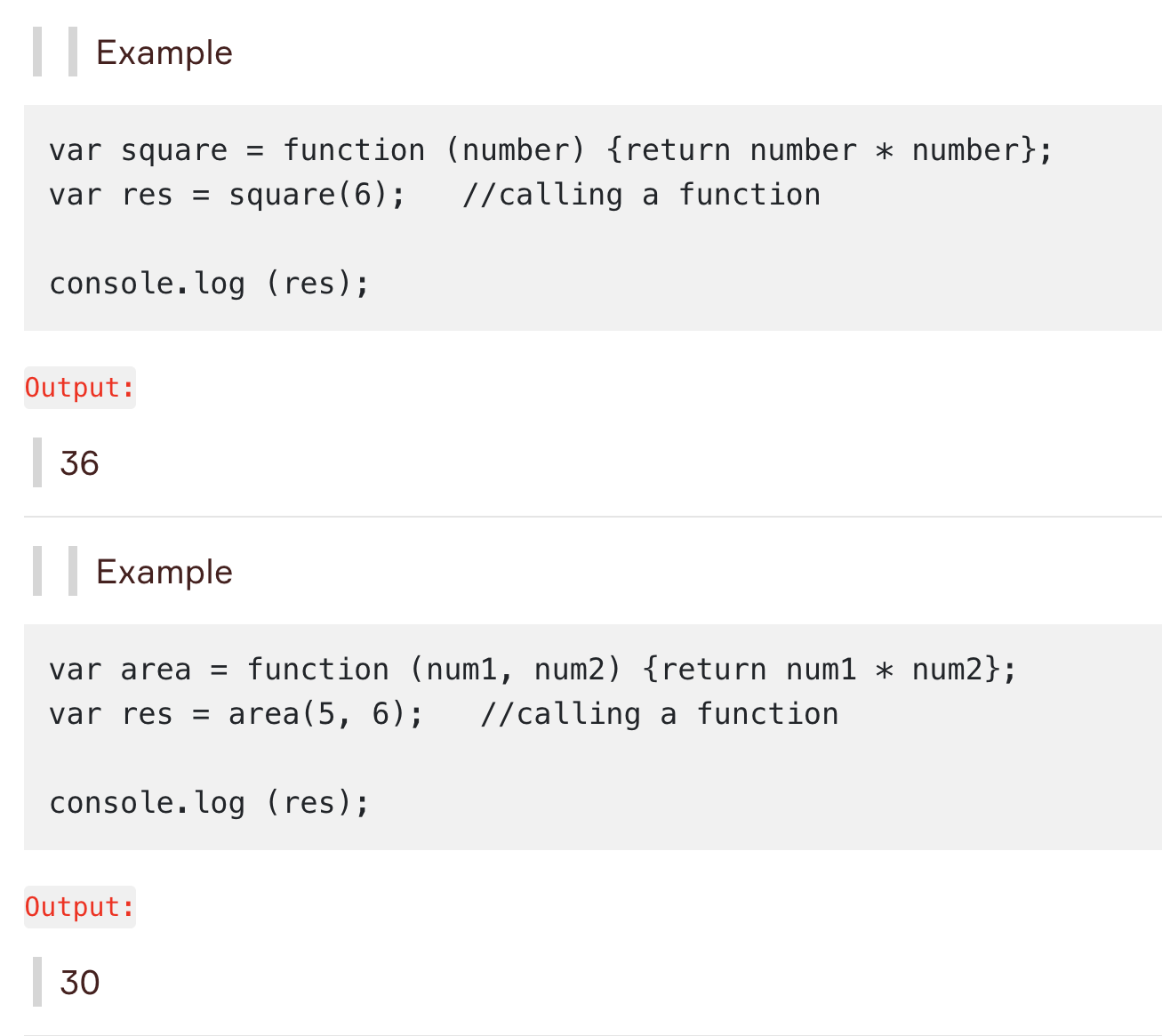
}

var res = square(5); //calling a function

console.log (res); //25

## Function Expression

**➤** We can also describe a JavaScript function using a function expression.  
**➤** A function expression is very similar to a function statement and has almost the same syntax.  
**➤** The main **difference** between a function expression and a function statement is the function name that can be omitted to create anonymous functions in function expressions.



## Function Constructor

**➤** We can also describe a JavaScript function using a function **constructor**.

const square = new Function("number", "return number \* number");

console.log(square(3)); //9

**❗ Warning :**  
You actually don't have to use the function constructor. Most of the time, you should avoid using the **new** keyword in JavaScript for your code speed and unexpected errors.

## Arrow Functions

**➤** **Arrow function**, which is added to JavaScript with **ES6**, is a **short** syntax for writing function expressions.

const myfunc3 = (p3) => {

console.log("my\_arrow\_func", p3)

}

myfunc3("lorem")

-----------------------------

const myfunc4 = p1 => p1 + 6;

console.log(myfunc4(10));

-----------------------------

const myfunc5 = (p1, p2) => "Welcome " + p1 + " it's " + p2;

//if we use return we have to use {}

// const myfunc5 = (p1, p2) => return {"Welcome " + p1 + " it's " + p2};

console.log(myfunc5("Bill", "Sunday"));

-----------------------------

const myfunc6 = () => console.log("heyyyo");

myfunc6();

## IIFE Functions

(Immediately Invobale Function Expression)

(function (p1, p2) {

console.log("IIFE example: ", p1, p2);

})("JS", "Sunday");

((p1,p2)=>{

console.log("IIFE example: ", p1, p2);

})("Another", "Version");

## Callback Functions

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Here is a quick example:

function greeting(*name*) {

  alert('Hello ' + name);

}

function processUserInput(*callback*) {

  var name = prompt('Please enter your name.');

  callback(name);

}

processUserInput(greeting); //

let my\_array = [6, 7, 11, 2, 123, 3, 55];

-------------------------

function sortElements\_1(a,b){

return a - b

}

const sortElements\_2 = (a,b)=>{return a - b}

const num1 = my\_array.sort(function (a,b) {return a - b});

const num2 = my\_array.sort((a,b)=> a - b);

const num3 = my\_array.sort(sortElements\_1);

const num4 = my\_array.sort(sortElements\_2);

console.log(num1);

console.log(num2);

console.log(num3);

console.log(num4);

//All give the same result

// [2, 3, 6, 7, 11, 55, 123]

# Objects & Classes

**➤** In previous lessons, we learned that **JavaScript** variables are containers for data values.

**➤** **Objects** are also variables. Unlike other variables, objects can contain many values.

**➤** **Objects** are useful for storing data in an organized manner.

**➤** **Objects** are composed of properties. A property is a “name: value” pair.

var dog = {

name: "Rocky",

legs: 4,

tail: 1,

qualities: ["loyalty","companionship"]

};

**➤** dog is an **object** and has properties just like the real world.

| **Property** | **Property Value** |
| --- | --- |
| name | Rocky |
| legs | 4 |
| tail | 1 |
| qualities | loyalty, companionship |

## Accessing Object Properties

**➤** There are 3 ways to access object properties.

1. objectName.propertyName
2. objectName['propertyName']
3. objectName[variable]

var dog = {

name: "Rocky",

legs: 4,

tail: 1,

qualities: ["loyalty","companionship"]

};

var myDog=dog.name;

var yourDog=dog["name"] ;

const propName = "name"

var ourDog=dog[propName] ;

## Object Methods

**➤** **Objects** can also have **functions**.

**➤** An **object method** is a **function** stored as a property.

**➤** We can use the following syntax to access an **object method**.

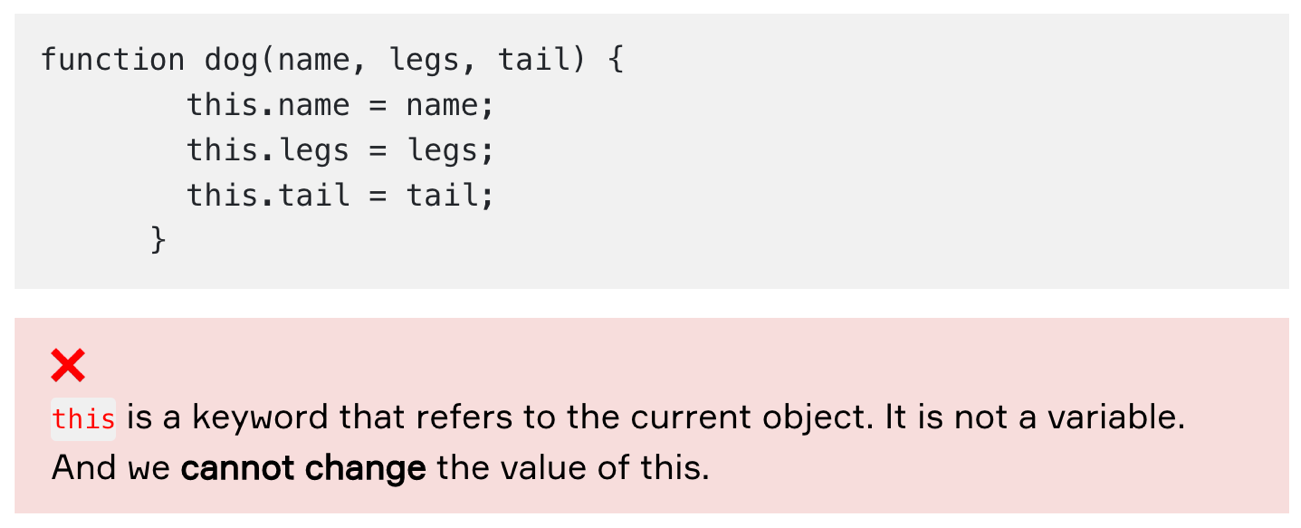
objectName.methodName()



## The Object Constructor

**➤** If we need a number of objects of a single type, we need to set an **object type**.

**➤** We can use an **object constructor function** to create an **object type**.



**➤** After we have an **object constructor**, we can create new objects of the same type with the new keyword.



## JavaScript Classes

**➤** A class is a type of function, but we use the keyword class and assign the properties within a constructor() method instead of using the keyword function to initiate it.

### Class Definition

**➤** **Classes** are "special functions," and the class syntax has two components just as you can describe function expressions and function declarations: class expressions and class declarations.  
**➤** The standard way to define a class is to use a declaration of class. You use the class keyword with the class name to designate a class, and always add the constructor() method.



### Methods

**➤** The constructor method is unique, it is where you initialize property, it is named automatically when a class is initialized, and it must have the exact name "constructor," also, if you don't have a constructor method, JavaScript can add an invisible and empty constructor method.



### Static Methods

**➤** Static methods are defined on the class itself, and not on the prototype.

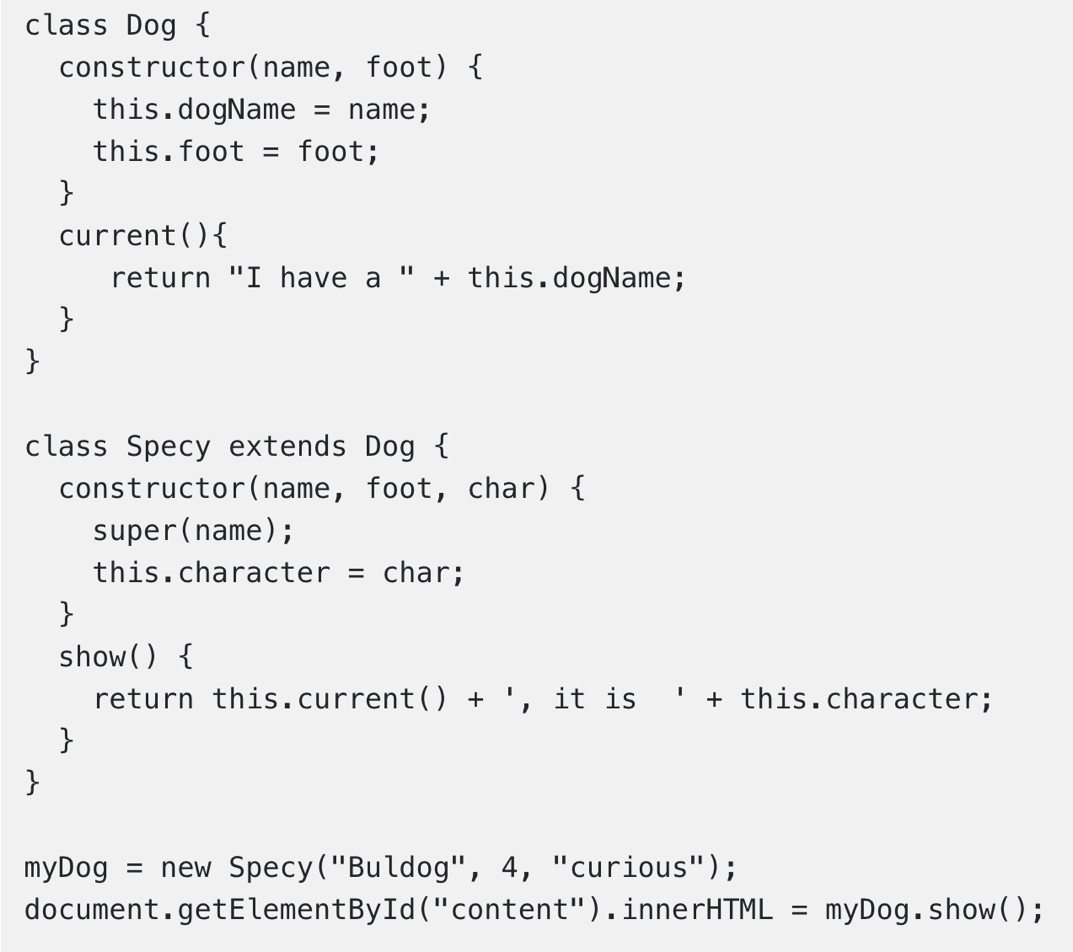
**➤** That means you cannot call a **static method** on the object, but on the class:



### Inheritance

**➤** To create a **class inheritance**, use the extends keyword.

**➤** A class created with a **class inheritance** inherits all the methods from another class:



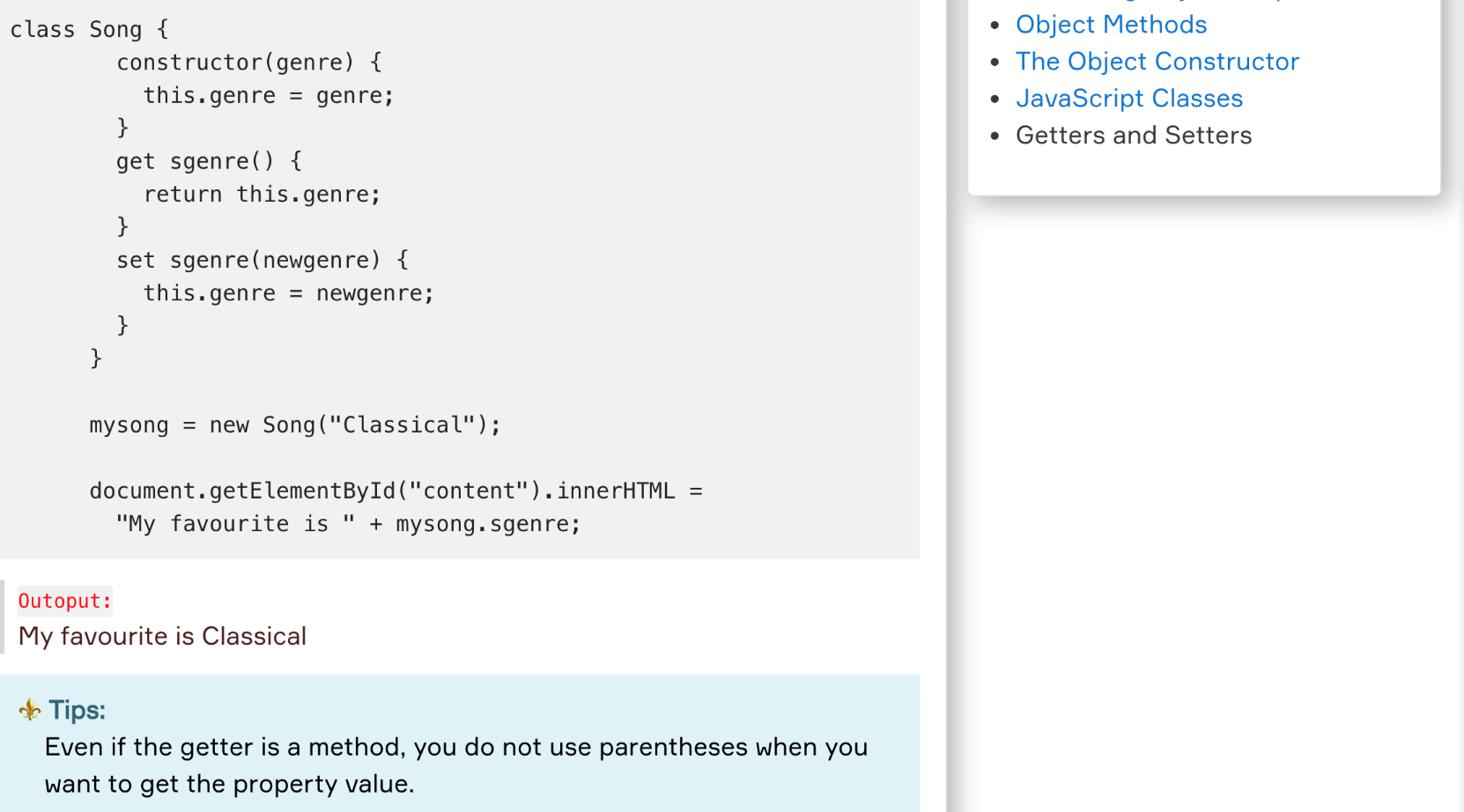
**➤** The super() method refers to the **parent class**.

**➤** By calling the super() method in the constructor method, we call the parent's constructor method and get access to the parent's properties and methods.

## Getters and Setters

**➤** Classes also allows us to use **getters** and **setters**.

**➤** To add **getters** and **setters** in the class, use the get and set keywords.



Outoput:  
My favourite is Classical

**⚜️ Tips:**

Even if the getter is a method, you do not use parentheses when you want to get the property value.

**➤** To use a setter, use the same syntax as when you set a property value, **without parentheses**.



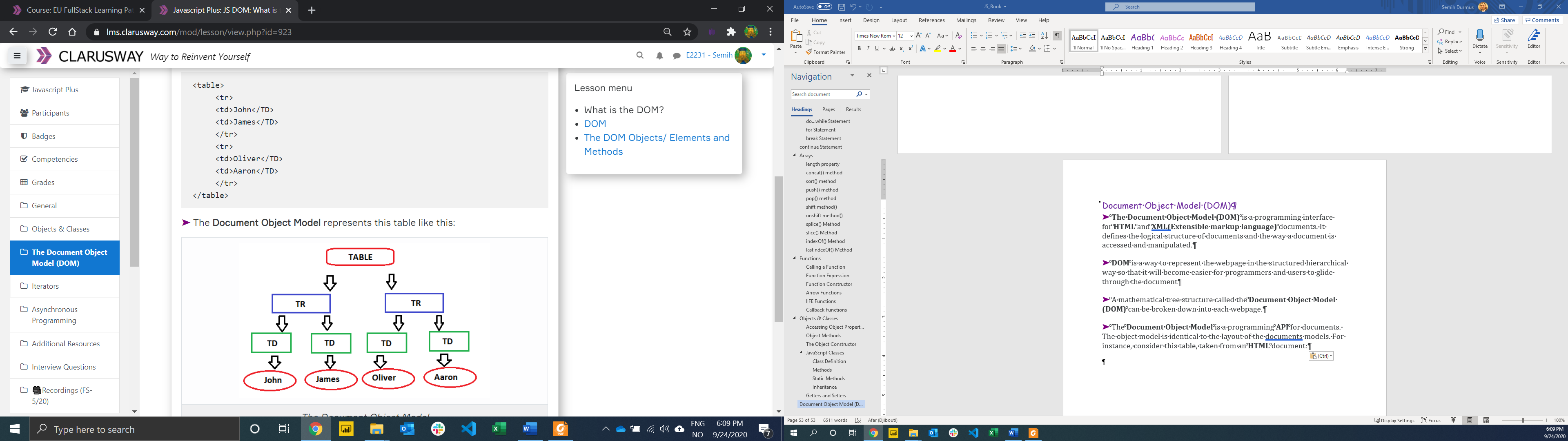
# Document Object Model (DOM)

**➤** **The Document Object Model (DOM)** is a programming interface for **HTML** and **XML(Extensible markup language)** documents. It defines the logical structure of documents and the way a document is accessed and manipulated.

**➤** **DOM** is a way to represent the webpage in the structured hierarchical way so that it will become easier for programmers and users to glide through the document

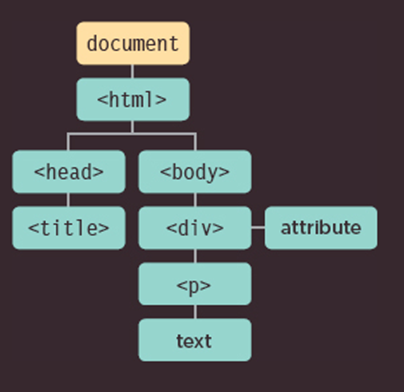
**➤** A mathematical tree structure called the **Document Object Model (DOM)** can be broken down into each webpage.

**➤** The **Document Object Model** is a programming **API** for documents. The object model is identical to the layout of the documents models. For instance, consider this table, taken from an **HTML** document:



## Node

**➤** The nodes in the node tree have a **hierarchical** relationship to each other.



**●** In a node tree, the top node is called the **root**.  
**●** Every node, except the root, has exactly one **parent node**.  
**●** A node can have any number of **children**.  
**●** A **leaf** is a node with no children.  
**●** **Siblings** are nodes with the same parent

**➤** Most element nodes have child nodes:

**●** The <html> node has two child nodes; <head> and <body>  
**●** The <head> node has one child node; the <title> node  
**●** The <title> node also has one child node; the text node "DOM Tutorial"  
**●** The <h1> and <p> nodes are siblings, and both child nodes of <body>

<!doctype html>

<html lang="en">

<head>

<title>Clarusway</title>

</head>

<body>

<h1>Hello, world!</h1>

<p>Clarusway</p>

</body>

</html>

**➤** The root node in the HTML above is <html>  
**➤** All other nodes in the document are contained within <html>.

The <html> node has two child nodes:

**●** <head>  
**●** <body>

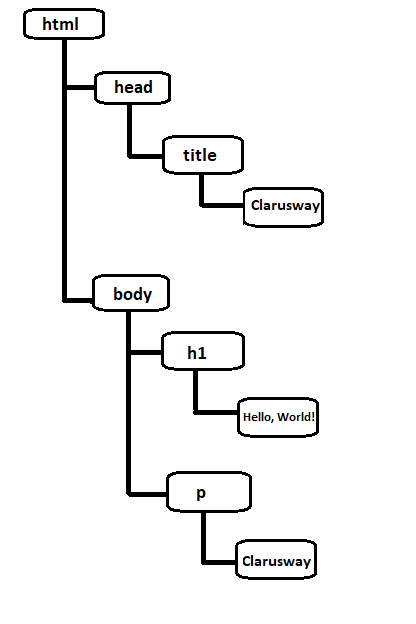
The <head> node holds a <title> node.

The <body> node holds:

**●** <h1>

**●** <p>

**➤** This document can be represented as the following node tree:



## The DOM Objects/ Elements and Methods

* **document :** The root of the page.

document.links, document.height

* **elements :** A Node in tree.
* **nodelist :** A group of elements

document.getElementByTagName('h1')

* **atribute :** A node in the DOM.

### Methods

| **Code** | **Function** |
| --- | --- |
| write(“string”) | writes the given string on the document |
| getElementById() | returns the element having the given id value. Find an element by element id |
| getElementsByName() | returns all the elements having the given name value |
| getElementsByTagName() | returns all the elements having the given tag name |
| getElementsByClassName() | returns all the elements having the given class name |
| innerHTML | a way of accessing, for the purpose of writing to, the content inside of an HTML element |
| document.forms | basically acessing the "forms" collection of the document object |

**Finding Elements**

* document.getElementById(id) : Find an element by element id
* document.getElementsByTagName(name): Find elements by tag name
* document.getElementsByClassName(name): Find elements by class name

**Changing Elements**

* element.innerHTML: Change the inner HTML of an element
* element.attribute: Change the attribute value of an HTML element

**Adding and Deleting Elements**

* document.createElement(element) : Create an HTML element
* document.removeChild(element): Remove an HTML element
* document.appendChild(element): Add an HTML element
* document.replaceChild(new, old) : Replace an HTML element
* document.write(text) : Write into the HTML output stream

# Iterators

**➤** **Iterators** are a new way to loop over any collection in JavaScript. They were introduced in **ES6** and have become really popular since they are widely useful and are used in various places.

## for/in Statement

**➤** The for ... in statement iterates over all enumerable properties of an object that are keyed by strings (ignoring ones keyed by Symbols), including inherited enumerable properties.

let obj = {x: 1, y: 2, z: 3};

for (let a in obj) {

console.log(a);

}

// x

// y

// z

**❗ Warning :**  
The for ... in loop should not be used to iterate over arrays where index order is important, because, depending on the JavaScript engine, it could iterate in an arbitrary order. Also, the iterating variable is a string, not a number, so if you try to do any math with the variable, you'll be performing string concatenation instead of addition.

## for/of Statement

**➤** The for ... of statement creates a loop iterating over iterable objects, including: built-in String, Array, array-like objects (e.g., arguments or NodeList), TypedArray, Map, Set, and user-defined iterables. It invokes a custom iteration hook with statements to be executed for the value of each distinct property of the object.

var names = ['Aaron', 'James', 'Oliver'];

var a;

for (a of names) {

console.log(a);

}

//Aaron

//James

//Oliver

# Asynchronous Programming

### **Synchronous Code**

In synchronous programs, if you have two lines of code (L1 followed by L2), then L2 cannot begin running until L1 has finished executing.

You can imagine this as if you are in a line of people waiting to buy train tickets. You can't begin to buy a train ticket until all the people in front of you have finished buying theirs. Similarly, the people behind you can't start buying their tickets until you have bought yours.

### **Asynchronous Code**

In asynchronous programs, you can have two lines of code (L1 followed by L2), where L1 schedules some tasks to be run in the future, but L2 runs before that task completes.

You can imagine as if you are eating at a sit-down restaurant. Other people order their food. You can also order your food. You don't have to wait for them to receive their food and finish eating before you order. Similarly, other people don't have to wait for you to get your food and finish eating before they can order. Everybody will get their food as soon as it is finished cooking.

console.log("Hello.");

setTimeout(function() {

console.log("Goodbye!");

}, 2000);

console.log("Hello again!");

//Say "Hello."

//Say "Hello again!"

//Do nothing for two seconds.

//Say "Goodbye!"

Note that asynchronous does not mean the same thing as concurrent or multi-threaded. JavaScript can have asynchronous code, but it is generally single-threaded. This is like a restaurant with a single worker who does all of the waiting and cooking. But if this worker works quickly enough and can switch between tasks efficiently enough, then the restaurant seemingly has multiple workers.

## Async callbacks

Async callbacks are functions that are specified as arguments when calling a function which will start executing code in the background.

When the background code finishes running, it calls the callback function to let you know the work is done or to let you know that something of interest has happened.

Using callbacks is slightly old-fashioned now, but you'll still see them in use in a number of older-but-still-commonly-used APIs.

An example of an async callback is the second parameter of the addEventListener() method (as we saw in action above):

btn.addEventListener('click', () => {

alert('You clicked me!');

let pElem = document.createElement('p');

pElem.textContent = 'This is a newly-added paragraph.';

document.body.appendChild(pElem);

});

The first parameter is the type of event to be listened for, and the second parameter is a callback function that is invoked when the event is fired.

## Promises

**Promises** are the new style of async code that you'll see used in **modern** Web APIs.

A good example is the **fetch() API**, which is basically very efficient. Let's look at a quick example, from our Fetching data from the server article:

fetch('products.json')

.then(function(response) {

return response.json();

})

.then(function(json) {

products = json;

initialize();

})

.catch(function(err) {

console.log('Fetch problem: ' + err.message);

});

Two **then()** blocks. Both contain a callback function that will run if the previous operation is successful, and each callback receives as input the result of the previous successful operation, so you can go forward and do something else to it.

The **catch()** block at the end runs if any of the .then() blocks fail — in a similar way to synchronous try...catch blocks, an error object is made available inside the catch(), which can be used to report the kind of error that has occurred

## Async/await

The **async** keyword can be placed before a function. The word “async” before a function means one simple thing: a function **always returns a promise**. Other values are wrapped in a resolved promise automatically.

async function f() {

return 1;

}

For instance, this function returns a resolved promise with the result of 1; let’s test it:

async function f() {

return 1;

}

f().then(console.log); // 1

The keyword **await** makes JavaScript wait until that promise settles and returns its result.

// works only inside async functions

let value = await promise;

Here’s an example with a promise that resolves in 1 second:

async function f() {

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("done!"), 1000)

});

let result = await promise; // wait until the promise resolves (\*)

console.log(result); // "done!"

}

f();

1. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template\_literals [↑](#footnote-ref-1)