**JavaScript Basics**

Table of Contents

[What is JavaScript 4](#_Toc26222929)

[Dynamic, interpreted Programming Language 4](#_Toc26222930)

[Weakly Type Programming Language 4](#_Toc26222931)

[What will we be covering? 4](#_Toc26222932)

[Basics 4](#_Toc26222933)

[Building a strong foundation 4](#_Toc26222934)

[Advanced Concepts 4](#_Toc26222935)

[Setting up Development Environment 5](#_Toc26222936)

[Visual Studio Code Editor 5](#_Toc26222937)

[Chrome Browser 5](#_Toc26222938)

[Important Principles 5](#_Toc26222939)

[Some Great External Resources 6](#_Toc26222940)

[Debugging JavaScript 6](#_Toc26222941)

[Referencing the JavaScript file in HTML 6](#_Toc26222942)

[Defer Importing 7](#_Toc26222943)

[Async Importing 7](#_Toc26222944)

[var/let/const 7](#_Toc26222945)

[Strict Mode 7](#_Toc26222946)

[Invoking Strict Mode 8](#_Toc26222947)

[Variables & Constants 8](#_Toc26222948)

[Variables 8](#_Toc26222949)

[Constants 8](#_Toc26222950)

[Operators 8](#_Toc26222951)

[Operator examples 8](#_Toc26222952)

[typeof Keyword 8](#_Toc26222953)

[Data Types and Objects 9](#_Toc26222954)

[Data Types 9](#_Toc26222955)

[Using Back Ticks 9](#_Toc26222956)

[Converting Data Types 10](#_Toc26222957)

[Mixing Number and String 10](#_Toc26222958)

[Control Structures 10](#_Toc26222959)

[Boolean Operators 10](#_Toc26222960)

[Conditional If/Else/Else If Statement 11](#_Toc26222961)

[Switch 11](#_Toc26222962)

[If statement 11](#_Toc26222963)

[If and else 11](#_Toc26222964)

[If, else if and else 11](#_Toc26222965)

[Ternary Operator – Single line If 11](#_Toc26222966)

[The logical AND or OR Operators 11](#_Toc26222967)

[Operator Precedence 12](#_Toc26222968)

[Logical Operators Short cuts 12](#_Toc26222969)

[Double Bang Operator !! 12](#_Toc26222970)

[Setting up Defaults 12](#_Toc26222971)

[Loops 13](#_Toc26222972)

[For Loop 13](#_Toc26222973)

[For Of Loop 13](#_Toc26222974)

[For In Loop 13](#_Toc26222975)

[While Loop 14](#_Toc26222976)

[Do While Loop 14](#_Toc26222977)

[Continue & Break 14](#_Toc26222978)

[Error Handling Try/Catch 14](#_Toc26222979)

[Try/Catch Basics 14](#_Toc26222980)

[Throwing a Custom Error 14](#_Toc26222981)

[Rethrowing an Error 15](#_Toc26222982)

[Try/Catch/Finally 16](#_Toc26222983)

[Try/Finally 16](#_Toc26222984)

[Functions 17](#_Toc26222985)

[Global Variables 17](#_Toc26222986)

[Parameters Vs Arguments 17](#_Toc26222987)

[Functions Vs Methods 18](#_Toc26222988)

[Function Expressions Vs Function Declarations 19](#_Toc26222989)

[Shadowing 19](#_Toc26222990)

[Anonymous Functions 19](#_Toc26222991)

[Self-Invoking function 20](#_Toc26222992)

[Arrow Function 20](#_Toc26222993)

[Some Important Concepts Regarding Functions 21](#_Toc26222994)

# What is JavaScript

* JavaScript was created to make webpages dynamic, changing content on the pages from inside the page.
* It is compiled at runtime. It can be executed as part of a webpage in a browser or directly on any machine.
* JavaScript has nothing in common with Java, it is totally independent.
* It is an interpreted language, gets compiled on the fly.
* It’s a hosted language, runs in different environments e.g. in the browser.
* JavaScript engine is built into the browser.
  + Chrome 🡺 V8
    - Was extracted to run JavaScript any where called “Node.js”
  + Firefox 🡺 SpiderMonkey
  + Job of the engine is to
    - Parse the code
    - Then on the fly compile it to machine code
    - And then it executes the machine code
  + **Important:** all of this happens on a single thread
* Cannot read the file system, it is kind of running in a sand box.
* JavaScript running in Node.js can access the file system

# Dynamic, interpreted Programming Language

You may store some string in a var and then later can store a number in it. Means code can change at run time.

# Weakly Type Programming Language

Unlike other programming languages, you don’t define that some variable must hold a certain value. Data types can change, are assumed.

# What will we be covering?

## Basics

Language basics, base syntax, efficient development, debugging, loops, functions, DOM basics, arrays and iterables objects.

## Building a strong foundation

Classes and OOP, constructor functions and prototypes, DOM and browser APIs, Events, Asynchronous and http (ajax)

## Advanced Concepts

Work with 3rd party libraries, modules, tooling (webpack) browser storage, browser support, frameworks, meta-programming, nodes introduction, security, memory leaks etc.

# Setting up Development Environment

## Visual Studio Code Editor

There are a lot of editors available and can use NotePad++ but here will be using **Visual Studio code editor**. It is free, is under active development, modern and customisable. We can install a lot of extensions as well.

* Download and install VS Code from <https://code.visualstudio.com/download>.
* Resources
  + Take a look at this post for short cuts etc: <https://itplate.blogspot.com/2019/11/visual-studio-code-shortcuts-extensions.html>
  + User and Workspace settings: <https://code.visualstudio.com/docs/getstarted/settings>
  + VS Code Docs: <https://code.visualstudio.com/docs>
  + VS Code Keybindings: <https://code.visualstudio.com/docs/getstarted/keybindings>
  + VS Code Extensions Docs: <https://code.visualstudio.com/docs/editor/extension-gallery>
  + Google Chrome DevTools Docs: <https://developers.google.com/web/tools/chrome-devtools/>
* If you don’t like the color theme, go to File 🡺 preferences 🡺 Color Theme 🡺 and select a different theme of your choice.
* Helpful Extensions, go to the extensions tab and add
  + Material Icon Theme by Philipp Kief
  + Prettier by Esben Petersen
  + Bracket Pair Colorizer CoenraadS
  + Chrome debug by Microsoft (Debugger for Chrome)
* To autoformat the code, go to File 🡺 Preferences 🡺 Keyboard shortcut 🡺 search for Format Document 🡺 Find it and then look at the short cut applied to it. On windows the shortcut would be **Shift+Alt+F**. Where ever you need to auto format, press this combination keys.
* Changing the User or Workspace settings: Go to File 🡺 Preferences 🡺 Settings 🡺 Search for lets say Prettier under User and make sure that “Prettier: Semi” and “Prettier: Single Quote” are selected.

## Chrome Browser

For testing we will be needing a browser, we’ll use Chrome but can use any browser.

* We will be using Chrome developer tools to debug the code.
* Press F12 to bring up the developer tools
* Click on Console tab and then the gear icon, make sure that Preserve Log check box is not checked.

# Important Principles

* javascript is case sensitive
* lines should end with a ;. Now this is best practice. But when you have two expressions on the same line then it is must to separate these with a ;.
* variable name cannot start with a number but can have a number in it
* starting a variable name with $ and \_ is allowed. No other special characters are allowed.
* \_ can appear any where in the variable name, it is allowed but is bad practice. Always use camel case.

## Some Great External Resources

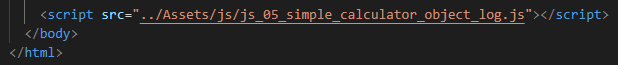
* [W3Schools](https://www.w3schools.com/js/default.asp)
* [MDN => JavaScript Basics](https://developer.mozilla.org/en-US/docs/Web/JavaScript)
* [MDN => Variables](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Variables)
* [MDN => Operators](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Math)
* [MDN => Functions](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Functions)
* [MDN => Arrays](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Arrays)
* [MDN => Objects](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics)
* [Popup Boxes](https://www.w3schools.com/js/js_popup.asp)
* [Continue & Breaking out of loops](https://www.w3schools.com/js/js_break.asp)

# Debugging JavaScript

* Read and utilize error messages, most probably others have encountered it as well so google it with keywords
* Use console.log(…) to gain insights into code, great for logical errors. You can print multiple variables separated with comma.
* Use the (Chrome or browser of your choice) debugging capabilities
* Syntax errors like forgetting to close the bracket, the IDE will yell at us with red squiggly lines. The file itself in the explorer will turn red as well. Hove over the problem, read the message and fix it.
* We may have misspelled parseInt as parse. The IDE will not yell at us in this case, but the browser will not run your code. Open the developer tools with F12 and look at the console tab. Read the message, it will give you the file name, the method name, click on it and it will take you to the source code.
* In the developer tools, the files that have been loaded show up under the Sources tab. Find the file you are interested in and can put the break points here. The controls to step into etc will be just under the window. Hover over the variables just like the IDE.
* Test code changes directly in the dev tools before implementing it. When you save the code here, it doesn’t save your file, only saves in the browser.
* Install Chrome Debug extension to debug code in the IDE as well.
  + Put the break point in the IDE and rather then opening in the browser directly, press F5 or click Debug 🡺 Start Debugging. It will ask you to pick the environment to run in, choose Chrome. A launch.json file will get added to the .vscode folder. Open launch.json and change the URL to the one showing in the browser. We’ll dive into local server later.

# Referencing the JavaScript file in HTML

We usually put the javascript file reference towards the end of the HTML file:



This way the html has already loaded.

We can also write inline javascript inside <script></script> tags but try to avoid this.

## Defer Importing

We can move the importing the scripts into the <head> section but then it will pose an issue. If we are referencing any thing from the dom, it will error out. We can use the defer attribute



The browser will download the script early but it will not execute the script early, it will execute after the HTML has been parsed and the scripts have downloaded.

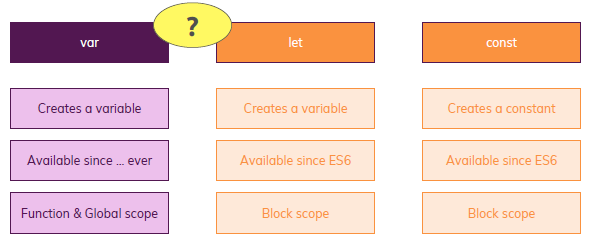
## Async Importing

These are scripts that we want to load early and want to execute early. In this case we don’t care if the parsing of the HTML has finished or not.



The order of the scripts download is not guaranteed if we are loading more than one script file.

# var/let/const



Important point to note is the scope.

# Strict Mode

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode#Changes_in_strict_mode>

Strict mode makes several changes to normal JavaScript semantics:

1. Eliminates some JavaScript silent errors by changing them to throw errors.
2. Fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode.
3. Prohibits some syntax likely to be defined in future versions of ECMAScript.

## Invoking Strict Mode

To invoke strict mode for an entire script, put the exact statement "use strict"; (or 'use strict';) before any other statements. The same goes for functions.

// Whole-script strict mode syntax

'use strict';

var v = "Hi! I'm a strict mode script!";

# Variables & Constants

|  |  |
| --- | --- |
| Variables We define variables with “let” keyword and when we change the value, then do it without the “let”.  var lastName = "Smith";  lastName = "Yodder"; | Constants Constants cannot be changed once defined. A keyword “const” is used for these.  const totalUsers = 15;  totalUsers = 20; **Cannot be changed** |

# Operators

|  |  |
| --- | --- |
| * + to add two numbers or concat strings * - subtract two numbers * \* multiply numbers * / divide two numbers * typeOf | * % modulus, divide two numbers, yield remainder * \*\* Exponentiation (e.g. 2\*\*3=8) * = assignment operator |

## Operator examples

Rater than doing

currentResult = currentResult + enteredNumber;

we can do

currentResult += enteredNumber;

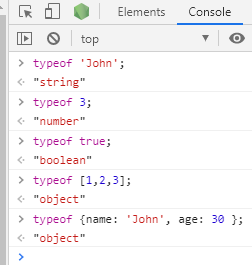
The same goes for -= ; /= ; \*=

alert(++currentResult) will be after the result value

alert(currentResult++) will be before the result value

## typeof Keyword

Lets you evaluate the type of the variable at run time.



# Data Types and Objects

For detail reading check

<https://www.w3schools.com/js/js_type_conversion.asp>

## Data Types

* numbers : 2, -3, 22.956
* string (Text): 'Hi', "Hi", `Hi` can use single quote, double quote and back ticks
* boolean: true/false; important for conditional code and situations where only have 2 options
* object: the most important type, allows to group data. Key value pair like {name: 'John', age: '21 }
* function
* arrays: is a list of data like [1, 2, 3]. Could be array of int, string, objects, array and can have mixed data as well. Arrays have some built in functions, you can view the list and description here: <https://www.w3schools.com/jsref/jsref_obj_array.asp> To read an array element, you’ll do it like myArray[0], 0 here is called index and is 0 based. The first element has an index of 0.
* date
* null 🡺 explicitly assigned null value.
* undefined 🡺 default value of uninitialized variables
* NaN 🡺 Not a type (not a number), technically it is of type number and can therefore be used in calculations.

Variable in JavaScript are not tied to data types, so following is perfectly legal.

let name = “John”;

name = 0;

### Using Back Ticks

const totalUsers = 15;

let myUsers = `Total Users: ${totalUsers}`;

Now the value of the totalUsers will be put in the string rather than the string literal.

We can also create multiple line strings with back ticks.

## Converting Data Types

* parseFloat(someVar)
* parseInt(someVar)
  + Here we can also do +someVar
* To convert to string 🡺 someVar.toString();

## Mixing Number and String

You saw the example with a number and a "text number" being added in JavaScript.

3 + '3' => '33'

That happens because the + operator also supports strings (for string concatenation).

It's the only arithmetic operator that supports strings though. For example, this will not work:

'hi' - 'i' => NaN

You can't generate a string of 'h' with the above code. Only + supports both strings and numbers.

Thankfully, JavaScript is pretty smart and therefore is actually able to handle this code:

3 \* '3' => 9

Please note: It yields the number (!) 9, NOT a string '9'!

Similarly, these operations also all work:

3 - '3' => 0

3 / '3' => 1

Just 3 + '3' yields '33' because here JavaScript uses the "I can combine text" mode of the + operator and generates a string instead of a number.

# Control Structures

## Boolean Operators

Conditional operators return true or false

|  |  |  |
| --- | --- | --- |
| == | Check for value equality (no type check) | a ==b |
| != | Check for value inequality | a != b |
| === and !== [prefer this] | Check for value and type (in)equality | a ===b OR a !== b |
| > and < | Check for value greater or smaller | a > b OR a < b |
| >= and <= | Check for value greater or equal / smaller or equal | a >= b OR a <= b |
| ! | Check if NOT true | !a |

JavaScript compares strings based on standard lexicographical ordering, using Unicode values. That means that b is greater than a for example.

## Conditional If/Else/Else If Statement

* Use if to specify a block of code to be executed, if a specified condition is true
* Use else to specify a block of code to be executed, if the same condition is false
* Use else if to specify a new condition to test, if the first condition is false
* Use switch to specify many alternative blocks of code to be executed

### Switch

To read more about the **switch** statement.

<https://www.w3schools.com/js/js_switch.asp>

### If statement

if (hour < 18) {

greeting = "Good day";

}

### If and else

if (condition) {

// block of code to be executed if the condition is true

} else {

// block of code to be executed if the condition is false

}

### If, else if and else

if (condition1) {

// block of code to be executed if condition1 is true

} else if (condition2) {

// block of code to be executed if the condition1 is false and condition2 is true

} else {

// block of code to be executed if the condition1 is false and condition2 is false

}

### Ternary Operator – Single line If

It is a single line if statement, check <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator> for more details.

let someValue = isMember ? "$2.00" : "$10.00";

## The logical AND or OR Operators

Some times when writing the condition, we need to check multiple conditions. This we can do with logical AND (&&) or OR (||) operators. Check mozilla documentation for more details:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators>

For &&: both conditions need to be true to return true

For ||: either condition could be true to return true

## Operator Precedence

The order in which the operators execute. Look at the following for more details:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence>

## Logical Operators Short cuts

You can quickly write and test your JavaScript in <https://playcode.io/> or <https://jsfiddle.net/>

[Ternary Operator](#_Ternary_Operator_–) – Single line If statement

### Double Bang Operator !!

It is used with two !!

const userInput = '';

//with ternary operator

const isValid = userInput ? true : false;

//with double bang

const isValidInput = !!userInput;

//both are false

console.log(isValid + ' ' + isValidInput);

'false false'

Only a single ! will negate it only.

### Setting up Defaults

const userInput = '';

const name = userInput || 'John';

console.log(name);

'John'

const userName = 'John';

const altName = '';

console.log(userName === 'John'); // generates and prints a boolean => true

console.log(userName); // wasn't touched, still is a string => 'John'

console.log(userName || null); // userName is truthy and therefore returned by || => 'John'

console.log(altName || 'John'); // altName is falsy (empty string), hence 'John' is returned => 'John'

console.log(altName || ''); // both altName and '' are falsy but if the first operand is falsy, the second one is always returned => ''

console.log(altName || null || 'Smith'); // altName and null are falsy, 'Smith' is returned => 'Smith'

console.log(userName && 'Smith'); // userName is truthy, hence second (!) value is returned => 'Smith'

console.log(altName && 'Smith'); // altName is falsy, hence first value is returned => ''

console.log(userName && ''); // us

true

'John'

'John'

'John'

''

'Smith'

'Smith'

''

''

## Loops

Executes code multiple times.

### For Loop

Execute code a certain amount of times (with counter variable)

for (let i = 0; i < 3; i++)

{

console.log(i);

}

### For Of Loop

Execute for every element in an array

for (const el of array)

{

console.log(el);

}

### For In Loop

Execute for every key in an object

for (const key in obj) {

console.log(key);

console.log(obj[key]);

}

### While Loop

Execute code as long as a condition is true

while (isLoggedIn) {

…

}

### Do While Loop

Execute code as long as a condition is true

do {

…

}

while (isLoggedIn)

### Continue & Break

* The break statement "jumps out" of a loop.
* The continue statement "jumps over" one iteration in the loop.

<https://www.w3schools.com/js/js_break.asp>

# Error Handling Try/Catch

## Try/Catch Basics

Good code is code that pre-emptively prevents errors in a program from crashing the program and instead handles those errors gracefully by doing something like recovering from the error, informing the user about it or logging the error somewhere. This is called error handling and the primary mechanism for error handling in JavaScript is the try…catch construct.

let myVariable = 2;

try {

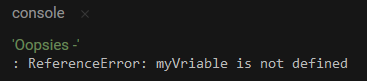
console.log(myVriable + 77); //wrong variable name used

} catch (e) {

console.log('Oopsies -', e);

}

Will result in



## Throwing a Custom Error

With the below example, if we don’t enter a value that can be coerced to a number we throw a custom error that’s then caught by our catch clause.

let myVariable = prompt('Give me a number');

try {

if (isNaN(+myVariable)) {

// throws if the value provided can't be coerced to a number

throw new Error('Not a number!');

}

console.log('Good choice', myVariable);

} catch (e) {

console.log(e);

}

Will result in



## Rethrowing an Error

A good coding practice is to catch and deal only errors that we expect and then to rethrow other errors to be handled by a potential parent try…catch construct:

let myVariable = prompt('Give me a number');

try {

if (isNaN(+myVariable)) {

throw new TypeError('Not a number!');

}

console.log('Good choice', myVariable);

} catch (e) {

if (e.name === 'TypeError') {

console.log(e);

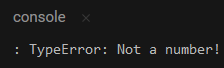
} else {

throw e;

}

}

Will result in



## Try/Catch/Finally

You can also tack-on a finally block to ensure that some code runs no matter if the code in the try blocks errors-out or not. The finally block can be useful to clean-up your code.

let myVariable = 2;

try {

console.log(myVriable + 77);

} catch (e) {

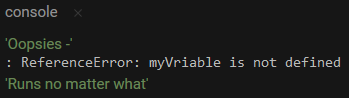
console.log('Oopsies -', e);

} finally {

console.log('Runs no matter what');

}

Will result in



## Try/Finally

Like finally, the catch clause is optional and, if you want you can use try…finally to ensures that some code runs upon encountering an error.

Here for example we try to set the background color on an element that doesn’t exist. First, using a try…catch…finally:

try {

document.querySelector('.here2').style.backgroundColor = 'pink';

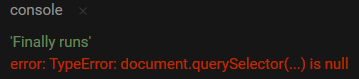
} finally {

console.log('Finally runs');

}

console.log('After try block');

Will result in



Notice how, when the catch clause is missing, the error is uncaught and the program crashes, but only after running the code in our finally clause.

# Functions

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure—a set of statements that performs a task or calculates a value. To use a function, you must define it somewhere in the scope from which you wish to call it.

A function definition (also called a function declaration, or function statement) consists of the function keyword, followed by:

* The name of the function.
* A list of parameters to the function, enclosed in parentheses and separated by commas.
* The JavaScript statements that define the function, enclosed in curly brackets, { }.

For example, the following code defines a simple function named square:

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.

## Global Variables

Never use global variables in the functions, these are variables that are declared outside the function and then used inside the function. These are perfectly legal to use but not the best practice.

## Parameters Vs Arguments

When talking about functions, the terms parameters and arguments are often interchangeably used as if it were one and the same thing but there is a very subtle difference.

* Parameters are variables listed as a part of the function definition.
* Arguments are values passed to the function when it is invoked.
* JavaScript does not throw an error if the number of arguments passed during a function invocation are different than the number of parameters listed during function definition.

// Basic function with three parameters that logs the sum of all the parameters

function argCheck(parameter1, parameter2, parameter3){

console.log(parameter1 + parameter2 + parameter3);

}

// Function with extra arguments

argCheck(1,2,3,4);

// Logs 6 (1 + 2 + 3, ignores 4)

// Function with missing arguments

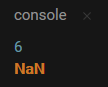
argCheck(1,2);

// Logs NaN because by default if a corresponding argument is missing, it is set to undefined.

// parameter3 is assigned undefined and so 1+2+undefined = NaN

// Note that, no error is thrown

Results in



## Functions Vs Methods

//add numbers is a function

function addNumbers(num1, num2){

console.log(num1+num2);

}

//Greet is a method

let contact = {

Name: "John",

Greet: function(toPrint){

console.log(toPrint);

}

}

//calling a function

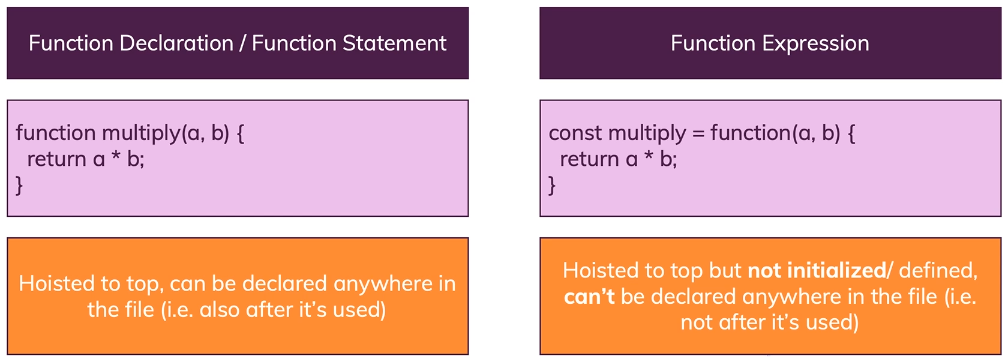
addNumbers(2,3)

//calling a method

contact.Greet(contact.Name);



## Function Expressions Vs Function Declarations



## Shadowing

JavaScript creates a new variable on a different scope - this variables does not overwrite or remove the global variable by the way - both co-exist. In the following example alert will display John since that is being passed into the function.

let userName = 'John';

function greetUser(name) {

//shadowing, same variable name as in the global scope

  let userName = name;

  console.log(userName);

}

userName = 'Smith';

greetUser('John');



## Anonymous Functions

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are invoked (called upon).

function myFunction(a, b) {

return a \* b;

}

myFunction(2, 3);

A JavaScript function can also be defined using an expression. A function expression can be stored in a variable. After a function expression has been stored in a variable, the variable can be used as a function:

var myExpFunction = function (a, b) {

return a \* b

};

var result = myExpFunction(4, 3);

console.log(result);

The result is 12. The above is an anonymous function. Important thing to note is that function expressions are not “hoisted” like normal functions. You have to first write the function and then call it.

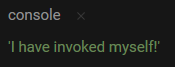
### Self-Invoking function

Function expressions can be made "self-invoking". A self-invoking expression is invoked (started) automatically, without being called. Function expressions will execute automatically if the expression is followed by (). You cannot self-invoke a function declaration. You have to add parentheses around the function to indicate that it is a function expression:

(function () {

console.log('I have invoked myself!');

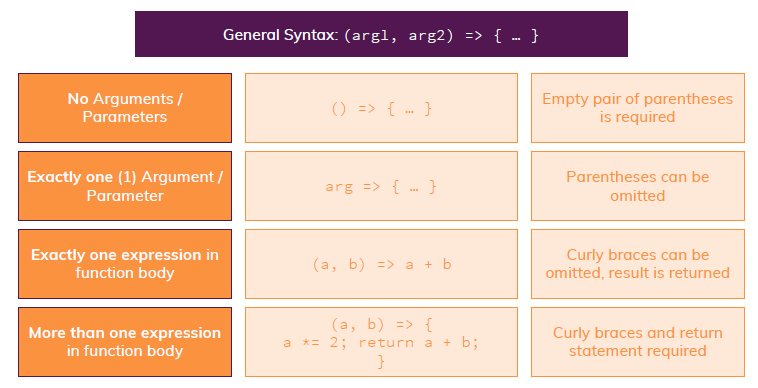
})();



## Arrow Function

Arrow functions allows a short syntax for writing function expressions. You don't need the function keyword, the return keyword, and the curly brackets.

Arrow functions do not have their own this. They are not well suited for defining object methods. Arrow functions are not hoisted. They must be defined before they are used. Using const is safer than using var, because a function expression is always constant value. You can only omit the return keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always keep them.



//ES5 function expression

var es5 = function(x, y) {

return x \* y;

}

//ES6 arrow function, incase of a single statement, no return is needed

const es6\_1 = (x, y) => x \* y;

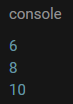
//arrow function with a return

const es6\_2 = (x, y) => { return x \* y };

console.log(es5(2,3));

console.log(es6\_1(2,4));

console.log(es6\_2(2,5));



## Some Important Concepts Regarding Functions

* <https://stackoverflow.com/questions/59150018/proper-use-of-this-in-javascript-objects?noredirect=1#comment104525531_59150018>
* <https://stackoverflow.com/questions/4616202/self-references-in-object-literals-initializers>

Arrow function examples

//arrow functions syntax

//1. default

const add = (a, b) => {

const result = a + b;

return result; // like in "normal" functions, parameters and return statement are OPTIONAL!

};

//Noteworthy: Semi-colon at end, no function keyword, parentheses around parameters/ arguments.

//2) Shorter parameter syntax, if exactly one parameter is received:

const log = message => {

console.log(message); // could also return something of course - this example just doesn't

};

//Noteworthy: Parentheses around parameter list can be omitted (for exactly one argument).

//3) Empty parameter parentheses if NO arguments are received:

const greet = () => {

console.log('Hi there!');

};

//Noteworthy: Parentheses have to be added (can't be omitted)

//4) Shorter function body, if exactly one expression is used:

const add2 = (a, b) => a + b;

//Noteworthy: Curly braces and return statement can be omitted, expression result is always returned automatically

//5) Function returns an object (with shortened syntax as shown in 4)):

const loadPerson = pName => ({name: pName });

//Noteworthy: Extra parentheses are required around the object, since the curly braces would otherwise be interpreted as the function body delimiters (and hence a syntax error would be thrown here).