

# M02 - JavaScript Fundamentals

...

Functions

# Functions

## What are functions?

- We often need to perform a similar action in many places in the script
- Example: show a message for login, logout...
- Functions are the main “building blocks” of a JavaScript program
- They allow the code to be called many times without repetition
- We have already seen examples of integrated functions: **alert**, **prompt** and **confirm**
- But we can also create our own functions!

# Functions

1. Function declaration
2. Function naming
3. Local and global variables
4. Parameters
5. Function return
6. Function expressions
7. Arrow functions

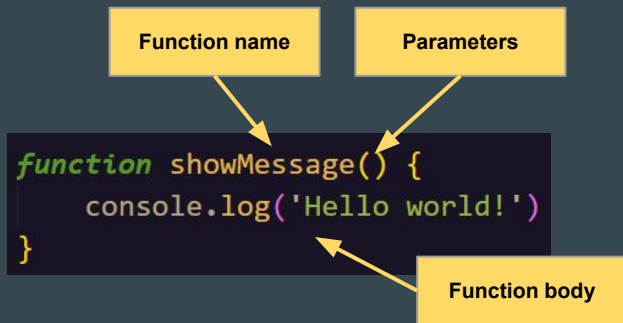
# Functions

## 1. Function declaration

# Functions

## 1. Function declaration

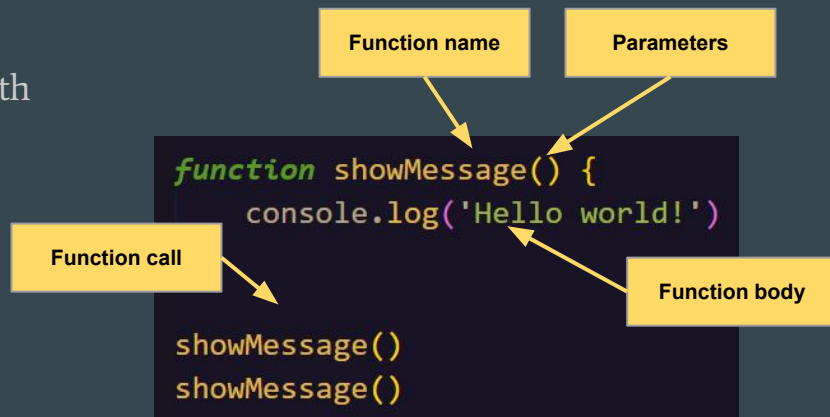
- To create a function, you can use a function declaration



# MO2 - Functions

## 1. Function declaration

- Invoking a Function
  - To **invoke a function**, use the function name with parentheses and parameters (if any)
  - This example clearly demonstrates one of the main objectives of the functions: **avoid duplication of code**
  - If we need to change the message or the way it is displayed, just modify the code in one place: the function that generates the message!



# Functions

## 2. Function naming

# Functions

## 2. Function naming

- Functions are actions. So their name is usually a verb
- It should be brief, as precise as possible and describe what the function does, so that someone reading the code receives an indication of what the function does
- It is a general practice to start a function with a verbal prefix that vaguely describes the action. There must be an agreement within the development team on the meaning of the prefixes
- For example, functions that start with "show" usually show something



# Functions

## 2. Function naming

- Functions that begin with...
  - “get..” - returns a value
  - “calc...” - calculates something
  - “create...” - creates something
  - “check...” - checks something and returns a boolean, etc.
- Examples:

```
showMessage(...) // shows a message
getAge(...)      // returns the age
calcSum(...)     // calculates a sum and returns the result
createForm(...)  // creates a form
checkPermission(...) // checks a permission, returns true or false
```

# Functions

## 2. Function naming

- A function must do exactly what its name suggests, not anymore
- Two independent actions generally deserve two functions, even though they are usually called together (in this case, we can do a third function that calls these two)
- The functions must be short and do exactly one thing. If this is large, it may be worth splitting the function into some smaller functions. Sometimes following this rule may not be so easy, but it is definitely a good strategy:
  - a separate function is easier to test and debug
  - its own existence is a great comment!

# Functions

## 3. Local and global variables

# Functions

## 3. Local and global variables

- A variable declared within a function is only visible within that function
- It is said to be a **local** variable

Local variable

```
function showMessage() {  
  let message = "Hello, I'm JavaScript!" // local variable  
  console.log(message)  
}
```

Error: variable does not  
exist here

```
showMessage() // Hello, I'm JavaScript!  
console.log(message) // Error! variable is local to the function
```

# M02 - Functions

## 3. Local and global variables

- A function can also access an external (**global**) variable, for example:



The diagram illustrates how a function can access a global variable. A yellow box labeled "Global variable" has an arrow pointing to the `let userName = 'John'` line. Another yellow box labeled "Access to global variable" has an arrow pointing to the `userName` property access within the function's `message` string.

```
let userName = 'John'

function showMessage() {
  let message = 'Hello, ' + userName
  console.log(message)
}

showMessage() // Hello, John
```

# M02 - Functions

## 3. Local and global variables

- The function has full access to the external variable and can modify it

Modify global  
variable

```
let userName = 'John'

function showMessage() {
  userName = 'Bob' // modify the global variable userName
  // ...
}

console.log(userName) // John (before the function call)
showMessage()
console.log(userName) // Bob (value was modified by the function)
```

- The external variable is used only if there is no local one with the same name
- So, an occasional change can happen if we don't use **let**

# Functions

## 3. Local and global variables

- If a variable with the same name is declared inside the function, it is used instead of the external one

```
let userName = 'John'

function showMessage() {
  let userName = 'Bob' // declares a local variable
  console.log(`Hello, ${userName}`) // Hello, Bob
}

showMessage() // the function will create and use it's local userName variable
console.log(userName) // John (value is not changed, the function did not changed the global variable)
```

Access to local  
variable

Unchanged  
global variable

# Functions

## 3. Local and global variables

- Summary
  - Global variables
    - Declared outside of any function
    - Visible to any function (unless overlaid by local variables)
    - Only store project-level data and accessible from anywhere
  - Normally, a function declares all variables specific to its task
  - The modern code has few if any global. Most variables reside in functions



# Functions

## 4. Parameters

# Functions

## 4. Parameters

- We can pass arbitrary data to functions
  - Function **parameters** are the names listed in the function definition
  - Function **arguments** are the actual values passed to (and received by) the function

```
function showMessage(from, text) { // arguments: from, text
  console.log(`${from}: ${text}`)
}

showMessage('Ann', 'Hello!') // Ann: Hello! (*)
showMessage('Ann', "What's up?") // Ann: What's up? (**)
```

- When the function is called on the lines (\*) and (\*\*), the given values are copied to local variables (**from** and **text**). From there the function uses these local variables

# Functions

## 4. Parameters

- See another example: we have the **from** variable and we pass it to the function
- The function changes the **from** variable, but the change is not seen from the outside, because the function always receives a copy of the value:

```
function showMessage(from, text) {  
  from = `* ${from} *` // modify the value of the local variable: from  
  console.log(`${from}: ${text}`)  
}  
  
let from = 'Ann'  
  
showMessage(from, 'Hello') // *Ann*: Hello  
  
// the value of "from" is the same, the function modified the local variable  
console.log(from) // Ann
```

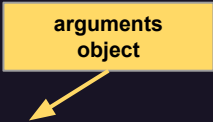
# Functions

## 4. Parameters

- The **arguments** object
  - JS functions have an internal object called **arguments**
  - Contains an array of the arguments used when the function was called (invoked)

```
let x = findMax(1, 123, 500, 115, 44, 88)
console.log(x) // 500
```

```
function findMax() {
  let i
  let max = -Infinity
  for (i = 0; i < arguments.length; i++) {
    if (arguments[i] > max) {
      max = arguments[i]
    }
  }
  return max
}
```



A yellow box labeled "arguments object" with an arrow pointing to the `arguments` property in the `for` loop of the `findMax` function.

# Functions

## 4. Parameters

- **Rest** parameters
  - A function can be called with any number of arguments, no matter how it is defined

```
function sum(a, b) {  
  return a + b  
}  
console.log(sum(1, 2, 3, 4, 5)) // 3
```

- There is no error for the "excessive" arguments. But in the result only the first two will be counted
- The **rest** parameters
  - mean "to collect the remaining parameters in an array"
  - can be mentioned in a function definition with three points ...
  - should always be the last to be mentioned

# Functions

## 4. Parameters

- Rest parameters
  - For example, to gather all (or some) arguments in an array:

Rest parameters for  
all the arguments

```
function sumAll(...args) { // args is the name of the array
  let sum = 0
  for (let arg of args) sum += arg
  return sum
}

console.log(sumAll(1)) // 1
console.log(sumAll(1, 2)) // 3
console.log(sumAll(1, 2, 3)) // 6
```

Rest parameters  
for some  
arguments

```
function showName(firstName, lastName, ...titles) {
  console.log(`${firstName} ${lastName}`) // Julius Caesar
  // rest of the arguments go to the array titles
  // i.e. titles = ['Consul', 'Imperator']
  console.log(titles[0]) // Consul
  console.log(titles[1]) // Imperator
  console.log(titles.length) // 2
}

showName('Julius', 'Caesar', 'Consul', 'Imperator')
```

# Functions

## 4. Parameters

- Default values
  - If a parameter is not provided, its value will be undefined

```
function showMessage(from, text) {  
  console.log(`${from}: ${text}`)  
}  
  
showMessage('John') // John: undefined
```

# Functions

## 4. Parameters

- Default values
  - If we want to use a default text we can specify it with =

```
function showMessage(from, text = 'No text given') {  
  console.log(`${from}: ${text}`)  
}  
  
showMessage('John') // John: No text given
```

- It can be a more complex expression, which is only evaluated and assigned if parameter is missing

```
function showMessage(from, text = anotherFunction()) {  
  // anotherFunction is only executed if no value for text is given  
  // variable text gets the return of the function  
}
```



# Functions

## 5. Function return

# Functions

## 5. Function return

- A function can return a value back to the calling code as a result
- The simplest example would be a function that adds two values:

```
function sum(a, b) {  
    return a + b  
}  
  
let result = sum(1, 2)  
console.log(result) // 3
```

- The **return** directive can be anywhere in the function. When execution reaches it, the function stops and the value is returned to the calling code (assigned to the result above).

# Functions

## 5. Function return

- There may be multiple instances of return in a single function. For example:

```
function checkAge(age) {  
  if (age > 18) {  
    return true  
  } else {  
    return confirm('Do you have your parents permission?')  
  }  
}  
  
let age = prompt('How old are you?', 18)  
  
if (checkAge(age)) {  
  alert('Access granted')  
} else {  
  alert('Access denied')  
}
```

# Functions

## 5. Function return

- It is possible to use the return without a single function value
- Causes the function to exit immediately

```
function showMovie(age) {  
  if (!checkAge(age)) {  
    return  
  }  
  console.log("Showing the movie...")  
}
```

- Uma função com um retorno vazio ou sem ele retorna **undefined**

# Functions

## 6. Function expressions

# Functions

## 6. Function expressions

- Definition

- It is very similar to a function declaration and has almost the same syntax
- The main difference between them is the name of the function, which can be omitted in expressions to create anonymous functions



The diagram shows two code snippets side-by-side. The first snippet is a function declaration: `function showMessage() { console.log('Hello world!') }`. A yellow box labeled "Function declaration" has an arrow pointing to the `function` keyword. The second snippet is a function expression: `let showMessage = function () { console.log('Hello world!') }`. A yellow box labeled "Function expressions" has an arrow pointing to the `function` keyword in this snippet.

```
function showMessage() {  
    console.log('Hello world!')  
}
```

Function declaration

```
let showMessage = function () {  
    console.log('Hello world!')  
}
```

Function expressions

- The function call is identical in both approaches

# Functions

## 6. Function expressions

- Differences between function declarations and expressions
  - Function expressions are created when execution arrives and is ONLY usable from then on
  - Function declarations are different:
    - A function declaration is usable throughout the all script/code block
    - In other words, when JavaScript prepares to execute the script or a block of code, it first looks for function declarations in it and creates functions. We can think of it as a "startup stage"
    - And after all function declarations are processed, the execution continues
    - As a result, a function declared as a function declaration can be called before it is defined

# Functions

## 6. Function expressions

- Differences between function declarations and expressions

It works!

showMessage()

```
function showMessage() {  
  console.log('Hello world!')  
}
```

Function  
declaration

It does not work!

showMessage()

```
let showMessage = function () {  
  console.log('Hello world!')  
}
```

Function  
expression



# Functions

## 6. Function expressions

- When should I use Functions declarations and expressions?
  - Consider function declaration syntax
  - It gives more freedom in how to organize our code, because we can call some functions before they are declared
  - It is also easier to search for the function `f (...) {...}` in the code than `let f = function (...) {...}`

# Functions

## 6. Function expressions

- Immediately invoked function expression (IIFE)
  - Immediately create and invoke the function
  - Just add the function inside parentheses and invoke it with new parentheses

```
(function () {  
    let message = 'Hello'  
    console.log(message) // Hello  
})();
```

# Functions

## 7. Arrow functions

# Functions

## 7. Arrow functions

- There is a simpler and more concise syntax for creating functions expressions
- They are called arrow functions
- Syntax:

`let func = (arg1, arg2, ...argN) => expression`

- Explaining the example:
  - Creates a `func` function that has arguments `arg1..argN`
  - Evaluates the `expression` on the right side
  - Returns the result

# Functions

## 7. Arrow functions

- Example of an arrow function and a similar function expression:

Arrow function

```
let sum = (a, b) => a + b  
console.log(sum(1, 2)) // 3
```

Function  
expression

```
let sum = function (a, b) {  
  return a + b  
}  
  
console.log(sum(1, 2)) // 3
```

# Functions

## 7. Arrow functions

- Removals:
  - `function` word
  - curly braces
  - `return` word
- Additions:
  - arrow (`=>`)

Arrow function

```
let sum = (a, b) => a + b
console.log(sum(1, 2)) // 3
```

Function expression

```
let sum = function (a, b) {
  return a + b
}
console.log(sum(1, 2)) // 3
```

# Functions

## 7. Arrow functions

- If we have only one argument, then parentheses can be omitted, making the writing of the function even shorter:

```
// the same as:  
// let double = function(n) { return n * 2 }  
let double = n => n * 2  
  
console.log(double(3)) // 6
```

# Functions

## 7. Arrow functions

- If there are no arguments, parentheses must be empty (but must be present)

```
let sayHi = () => console.log('Hello!')  
  
sayHi()
```



# Functions

## 7. Arrow functions

- Previous examples received arguments from the left of `=>` and evaluated simple expressions
- Sometimes we have several expressions or statements
- To do this, wrap everything in curly braces and use the word **return**

```
let sum = (a, b) => { // curly braces opens a multi-line function
  let result = a + b
  return result // when using curly braces we must use return declaration
}

console.log(sum(1, 2)) // 3
```

# Functions

## 7. Arrow functions

- Arrow functions may look strange and barely readable at first, but this changes quickly as you get used to the structure
- They are very convenient for simple one-line actions, when we don't want to write too much code