M01 - JavaScript Fundamentals

•••

Loops

LOOPS

- Sometimes, certain instructions require repeated execution
- Loops are the ideal way to reproduce this effect
- A loop represents a set of instructions that must be repeated
- In the context of a loop, a repetition is referred to as an iteration
- Loop types:

    - while - the condition is checked before each iteration
    - do…while - the condition is checked after each iteration
    - for(;;) - the condition is checked before each iteration, additional settings available.

Loops

1. WHILE statement

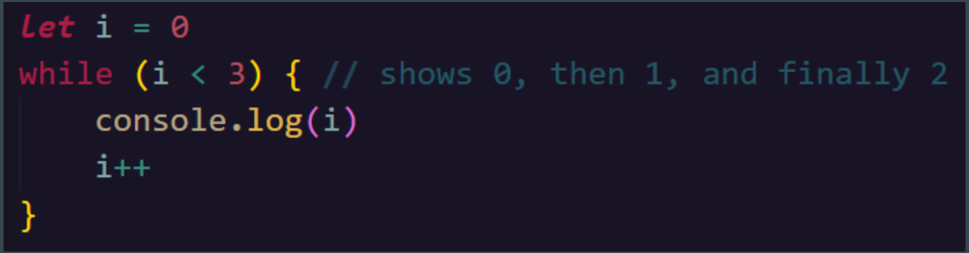
Loops

1. WHILE statement

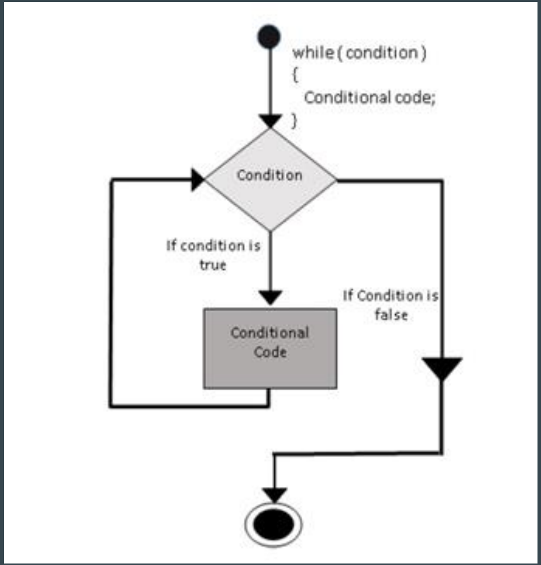
- The while loop has the following syntax:

  while(condition) {… loop body …}

- As long as the condition is true, the loop body is executed
- For example, the cycle below shows i while i<3:



```
let i = 0
while (i < 3) { // shows 0, then 1, and finally 2
    console.log(i)
    i++
}
```



Loops

2. DO…WHILE statement

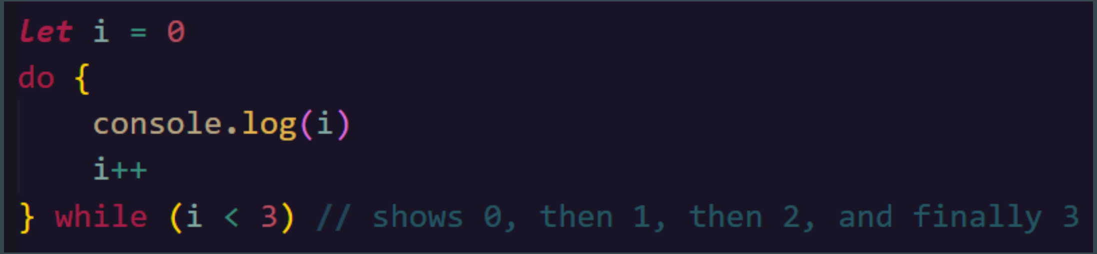
Loops

2. DO...WHILE statement

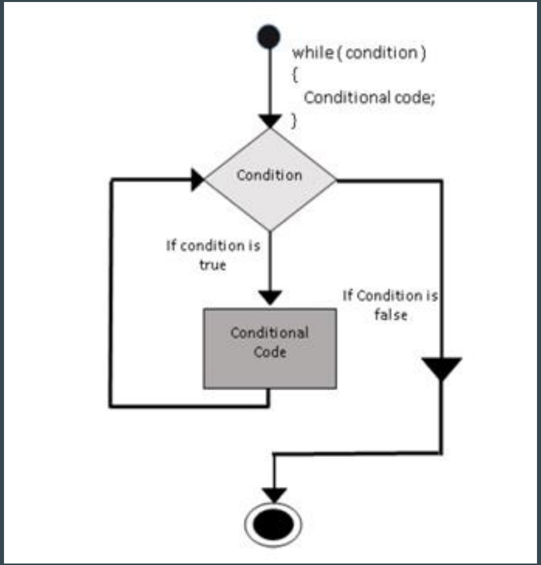
- The condition check can be moved below the loop body using the syntax:

  do {... loop body ...}while(condition)

- The loop first executes the body and then checks the condition. As long as it is true, it runs again.



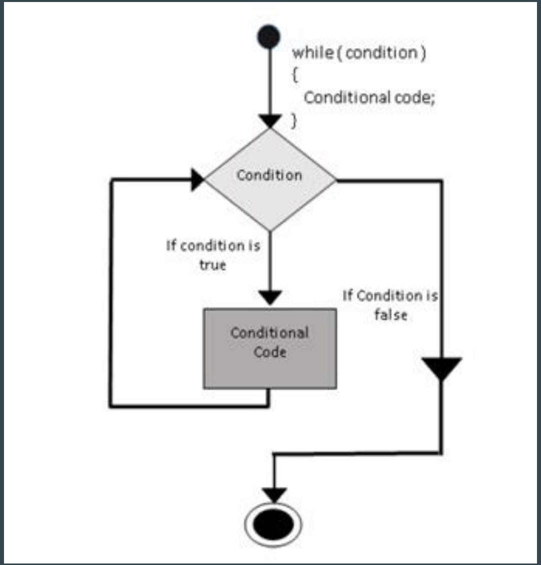
```
let i = 0
do {
    console.log(i)
    i++
} while (i < 3) // shows 0, then 1, then 2, and finally 3
```



Loops

2. DO...WHILE statement

- This form of syntax should only be used when you want the loop body to be executed at least once, regardless of the condition in effect.
- Usually, the other way is preferred: while (...) {...}



Loops

3. FOR statement

Loops

3. FOR statement

- for - loops through a block of code a number of times

- for/in - loops through the values of an iterable object
  (when we use arrays)

- for/of - loops through the properties of an object
  (when we use objects)

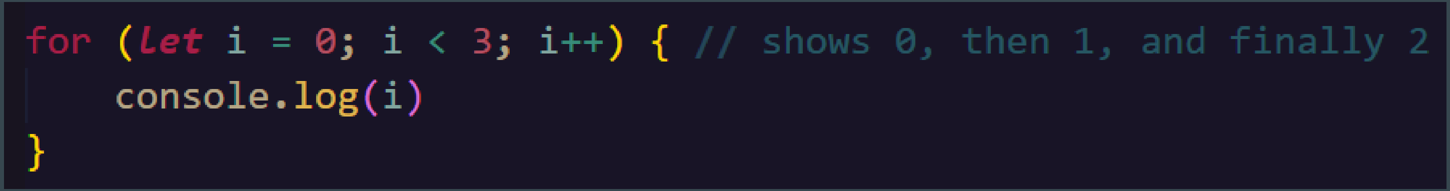
Loops

3. FOR statement

- The for loop is the most commonly used loop
- Syntax:

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

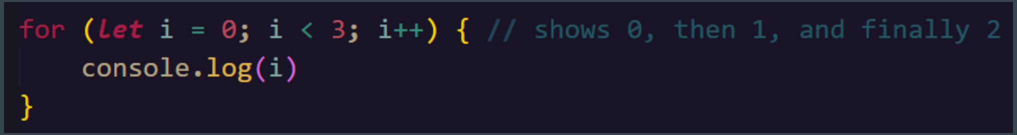
- Example:



```
for (let i = 0; i < 3; i++) { // shows 0, then 1, and finally 2
    console.log(i)
}
```

Loops

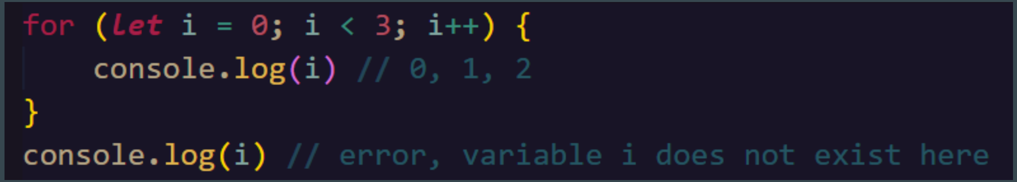
3. FOR statement



```
for (let i = 0; i < 3; i++) { // shows 0, then 1, and finally 2
    console.log(i)
}
```

- Explanation:
  - begin i = 0 executes once upon entering the loop
  - condition i < 3 checked before each iteration of the cycle. If false, the cycle stops.
  - step i++ executes after the body in each iteration, but before checking the condition.
  - body console.log(i) runs repeatedly while condition is true

- Use of inline variable
  - Variable i only exists within the block where it was defined

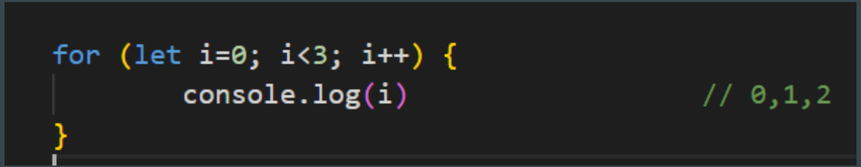


```
for (let i = 0; i < 3; i++) {
    console.log(i) // 0, 1, 2
}
console.log(i) // error, variable i does not exist here
```

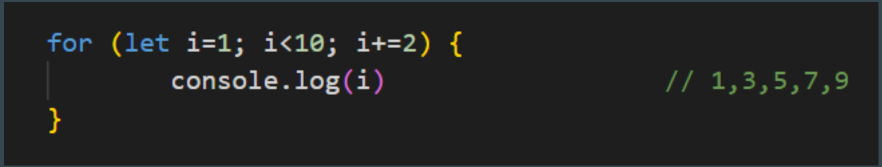
Loops

3. FOR statement

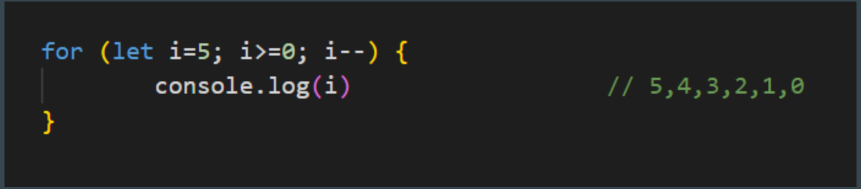
- Step argument



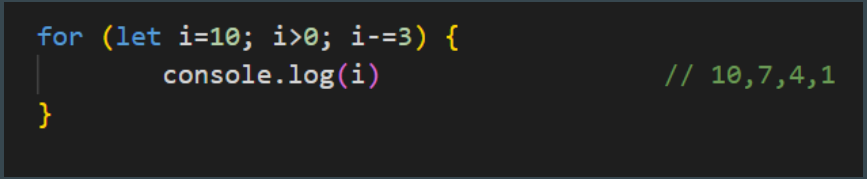
```
for (let i=0; i<3; i++) {
        console.log(i)                  // 0,1,2
}
```



```
for (let i=1; i<10; i+=2) {
        console.log(i)                  // 1,3,5,7,9
}
```



```
for (let i=5; i>=0; i--) {
        console.log(i)                  // 5,4,3,2,1,0
}
```

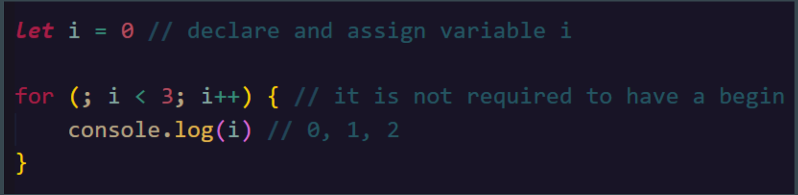


```
for (let i=10; i>0; i-=3) {
        console.log(i)                  // 10,7,4,1
}
```

Loops

3. FOR statement

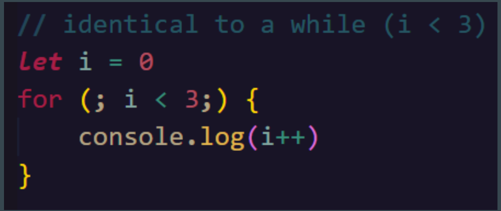
- Skip parts
  - Any part of the for cycle can be ignored
  - Remove the begin



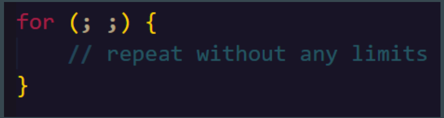
```
let i = 0 // declare and assign variable i

for (; i < 3; i++) { // it is not required to have a begin
    console.log(i) // 0, 1, 2
}
```

  - Remove the step

    (default step is 1)



```
// identical to a while (i < 3)
let i = 0
for (; i < 3;) {
    console.log(i++)
}
```



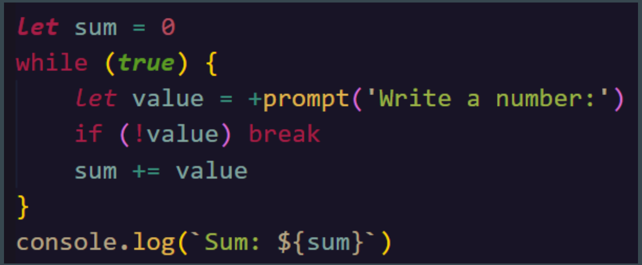
```
for (; ;) {
    // repeat without any limits
}
```

  - Remove all

Loops

3. FOR statement

- Loop break
  - Normally, a cycle ends when its condition becomes false
  - We can force the exit at any time using the special interrupt directive: break



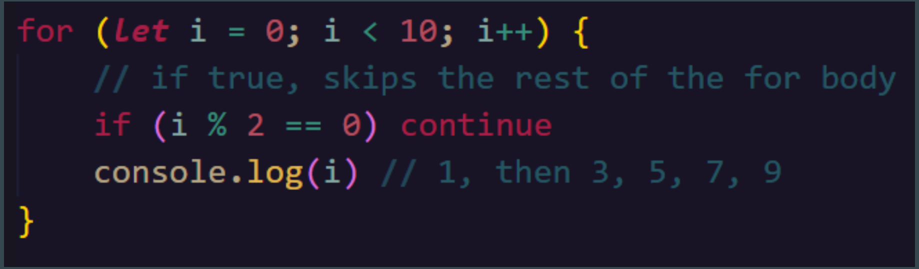
```
let sum = 0
while (true) {
    let value = +prompt('Write a number:')
    if (!value) break
    sum += value
}
console.log(`Sum: ${sum}`)
```

  - The combination of infinite cycle + break is great for situations where the condition of a loop must be checked not at the beginning or end of the cycle, but in the middle or even at various places in the loop body

Loops

3. FOR statement

- Skip to the next iteration
  - The continue directive is a lighter version of the break. Not for the whole cycle. Instead, it interrupts the current iteration and forces the loop to start a new one (if the condition allows).
  - We can use it if we finish the current iteration and want to move on to the next one
  - Example:



```
for (let i = 0; i < 10; i++) {
    // if true, skips the rest of the for body
    if (i % 2 == 0) continue
    console.log(i) // 1, then 3, 5, 7, 9
}
```

M01 - 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

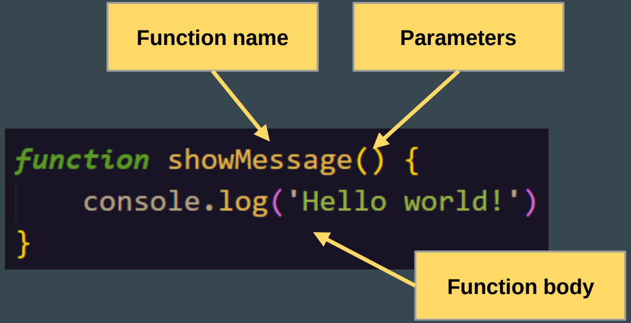
Functions

1. Function declaration

Functions

1. Function declaration

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

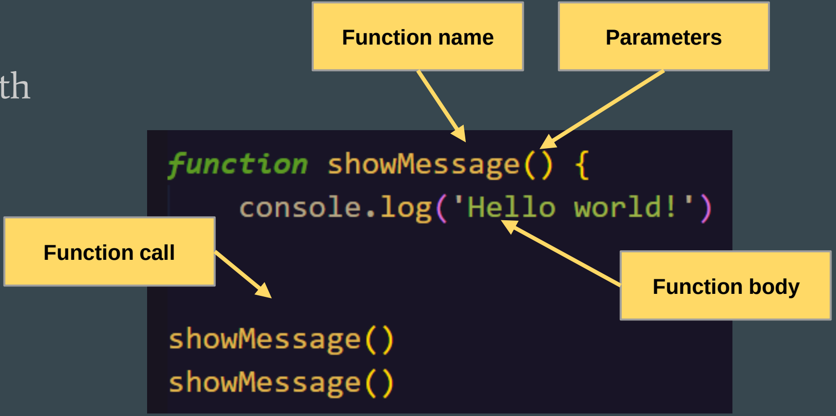


```
function showMessage() {
    console.log('Hello world!')
}
```

M02 - 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!



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

showMessage()
showMessage()
```

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

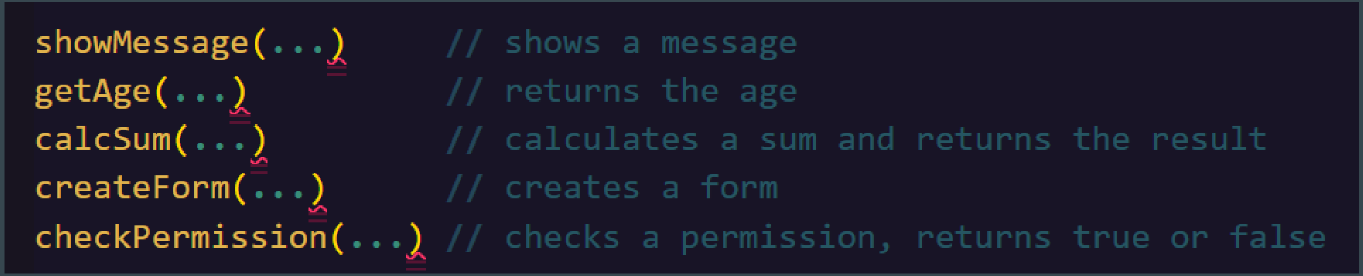
- It is a general practice to start a function with a verbal prefix that vaguely describes the action. There must be an agreement within the development team on the meaning of the prefixes

- 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

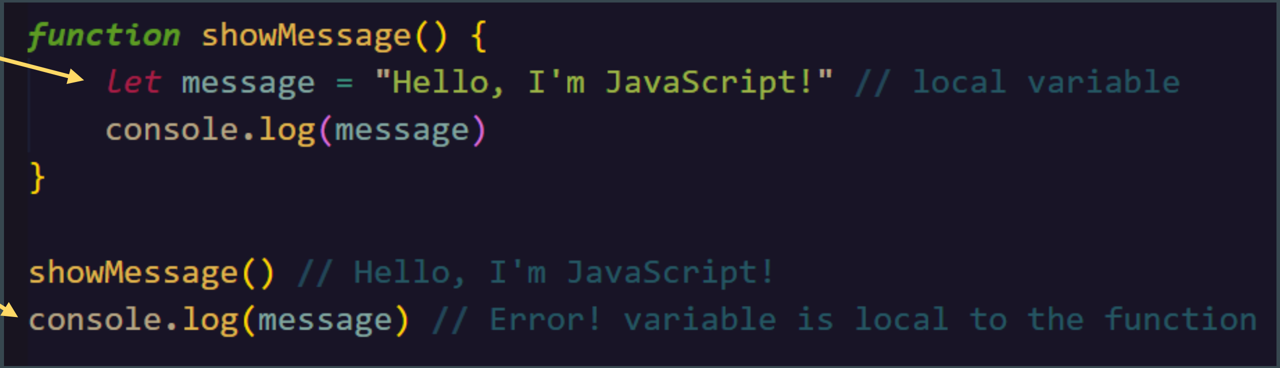
Functions

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

Error: variable does not exist here

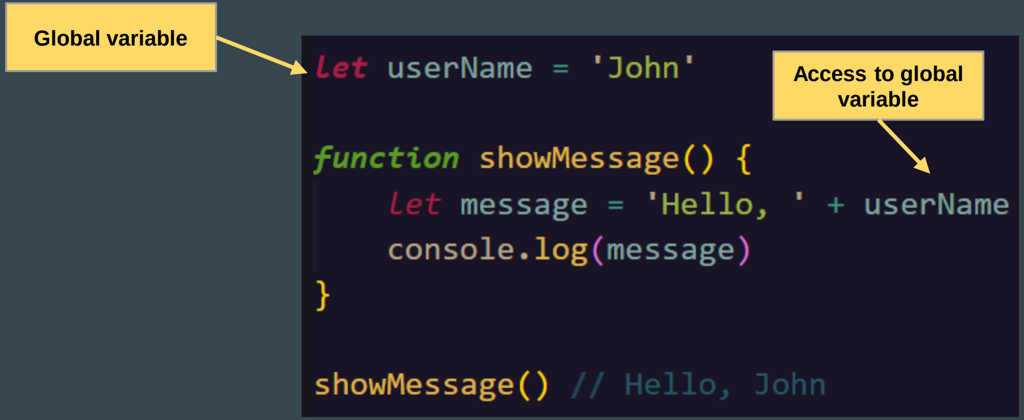


```
function showMessage() {
    let message = "Hello, I'm JavaScript!" // local variable
    console.log(message)
}

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:

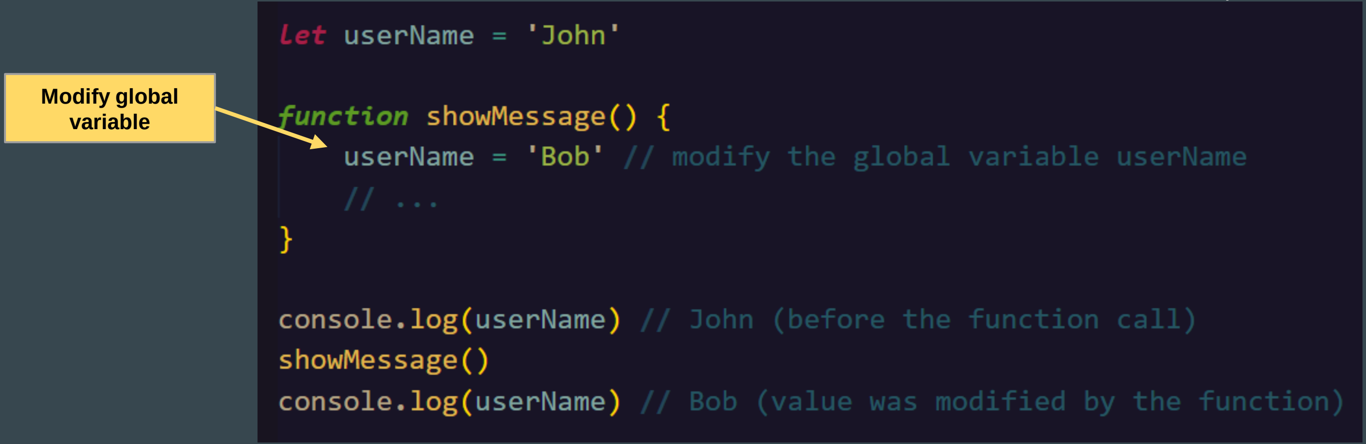


```
let userName = 'John'

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



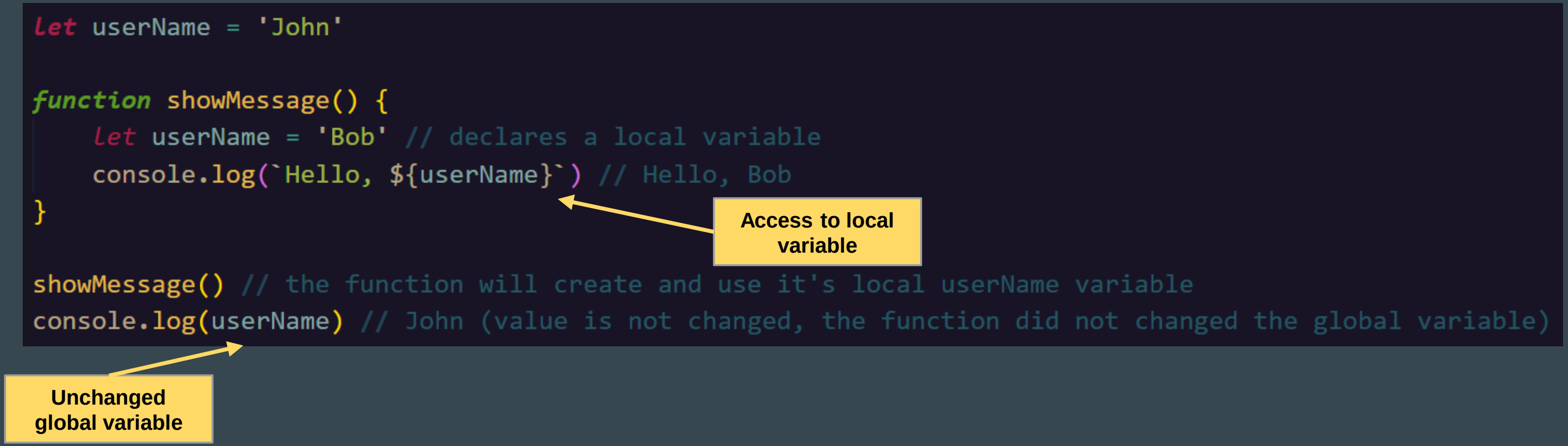
```
let userName = 'John'

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)
```

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

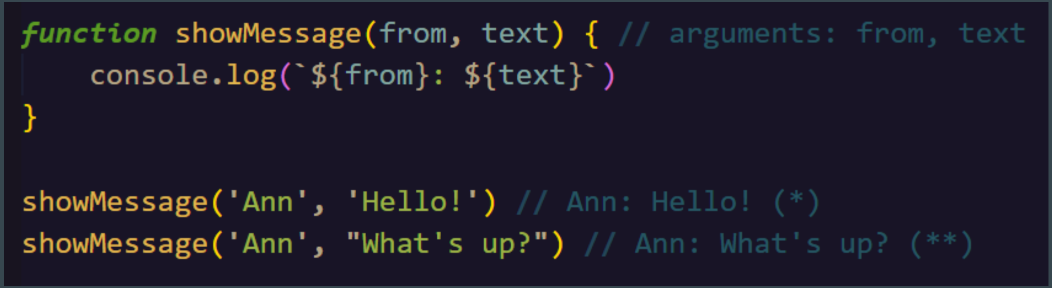
Functions

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



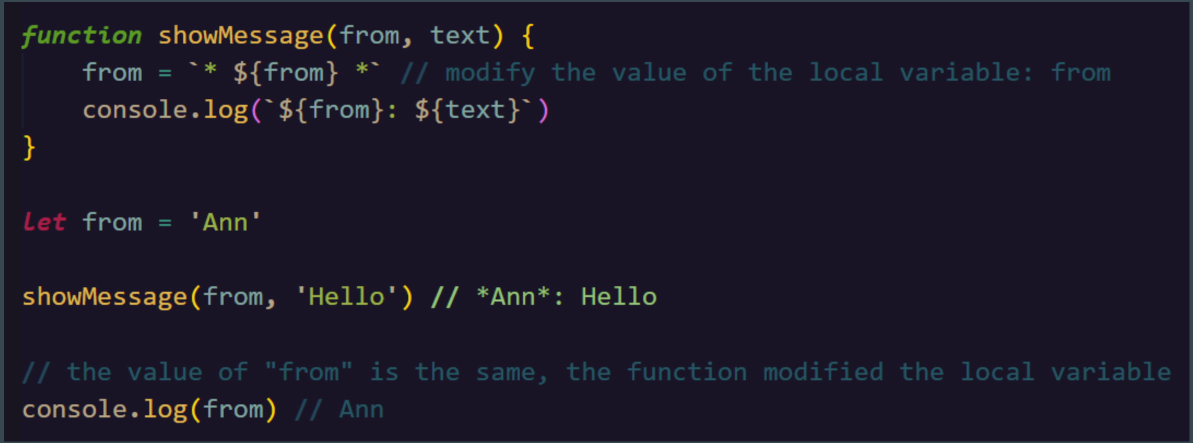
```
function showMessage(from, text) { // arguments: from, text
    console.log(`${from}: ${text}`)
}

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:

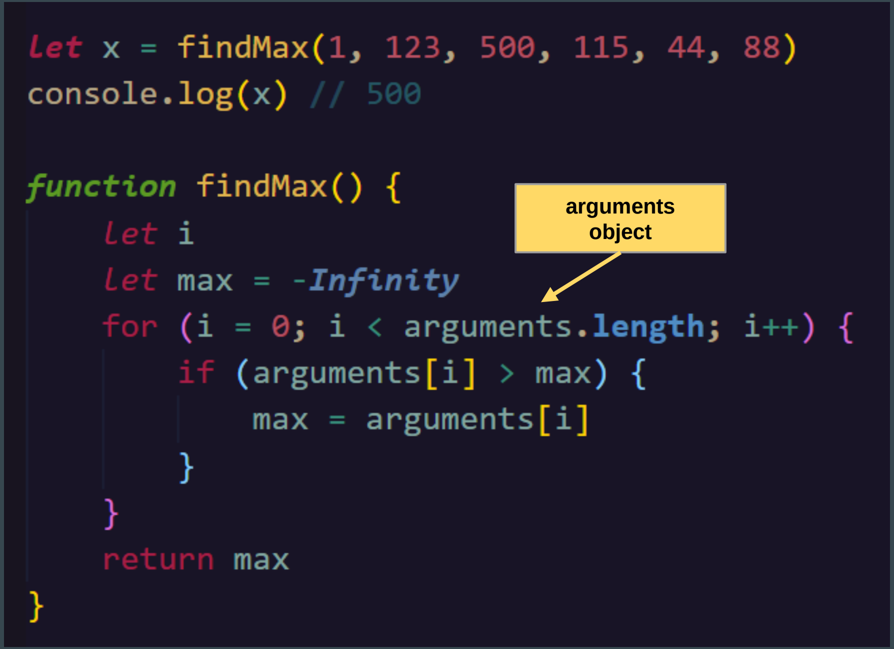


```javascript
function showMessage(from, text) {
    from = `* ${from} *` // modify the value of the local variable: from
    console.log(`${from}: ${text}`)
}

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)

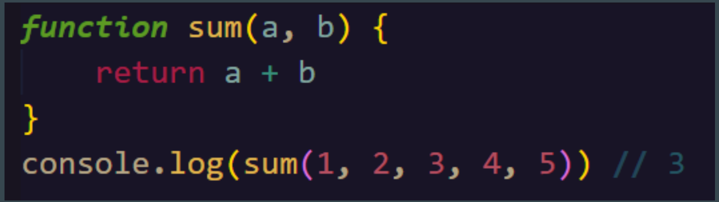


```
let x = findMax(1, 123, 500, 115, 44, 88)
console.log(x) // 500

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

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

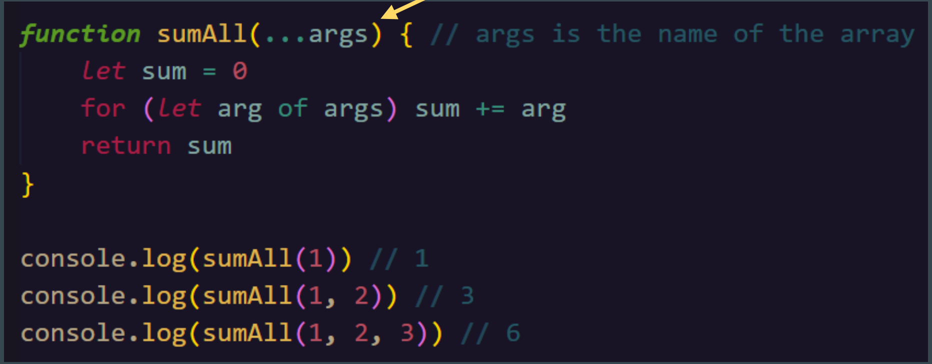
Functions

4. Parameters

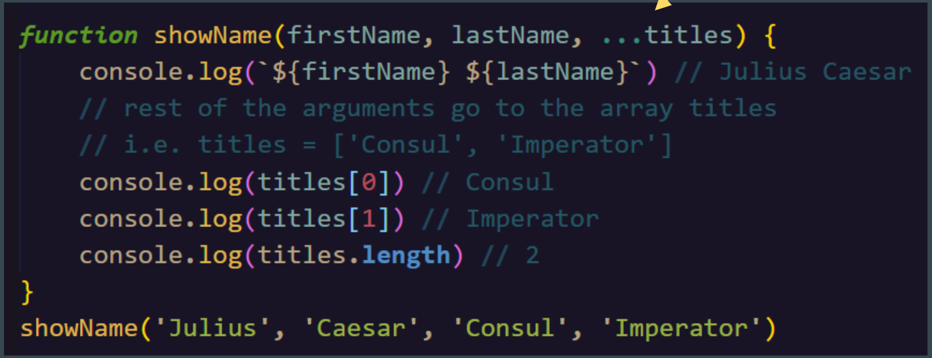
- Rest parameters
  - For example, to gather all (or some) arguments in an array:

Rest parameters for all the arguments

Rest parameters for some arguments



```javascript
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
```

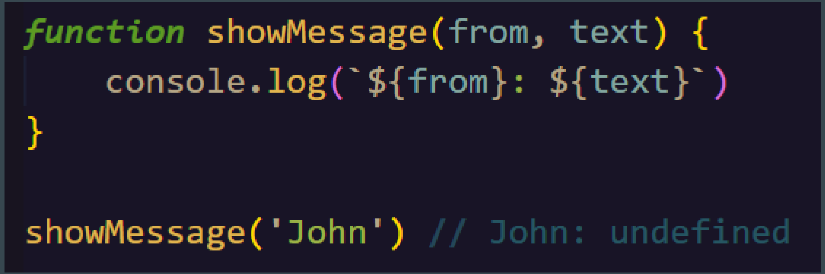


```javascript
function showName(firstName, lastName, ...titles) {
    console.log(`${firstName} ${lastName}`) // Julius Caesar
    // rest of the arguments go to the array titles
    // i.e. titles = ['Consul', 'Imperator']
    console.log(titles[0]) // Consul
    console.log(titles[1]) // Imperator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

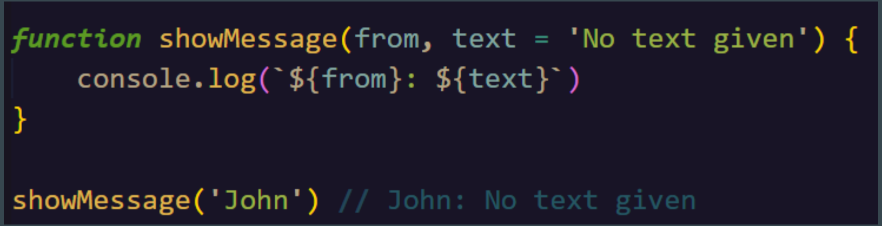


```
function showMessage(from, text) {
    console.log(`${from}: ${text}`)
}

showMessage('John') // John: undefined
```

Functions

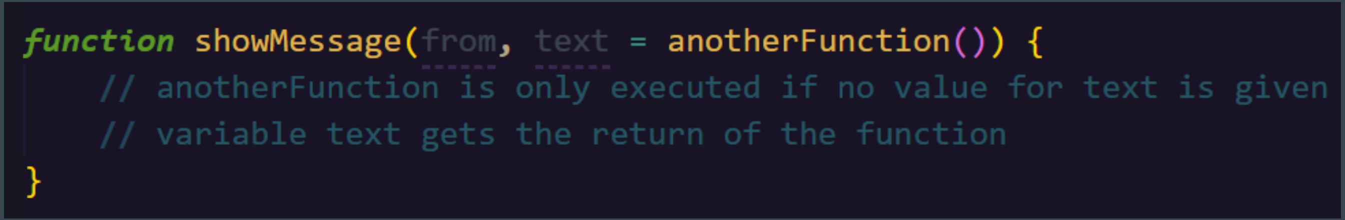
4. Parameters

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



```javascript
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



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

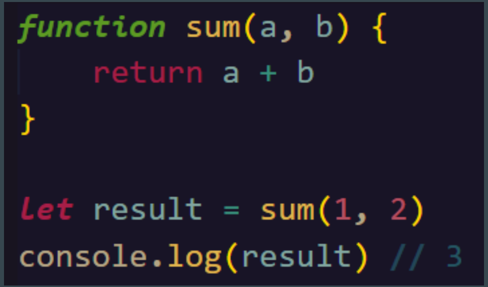
Functions

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:



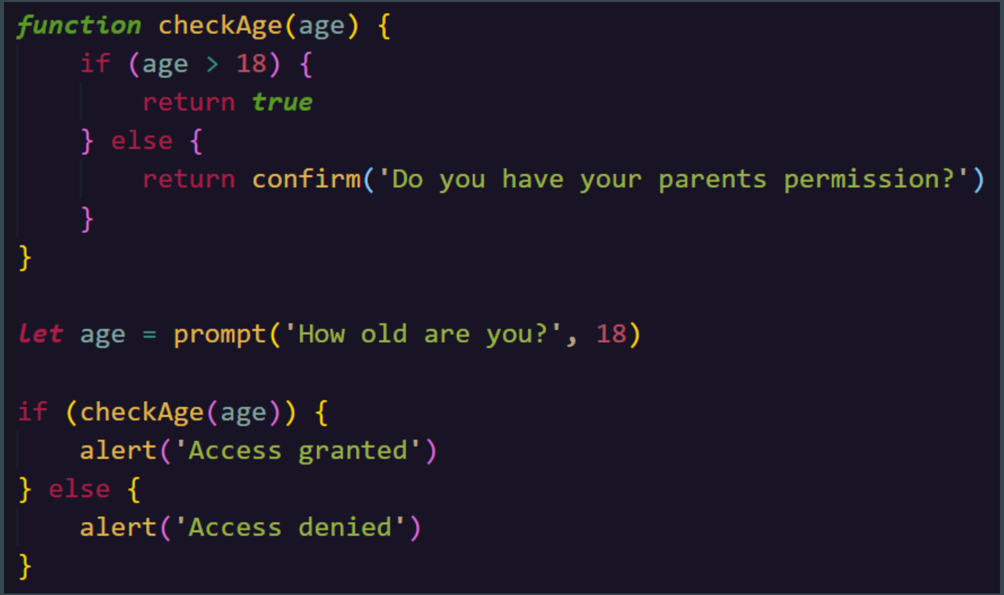
```
function sum(a, b) {
    return a + b
}

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:

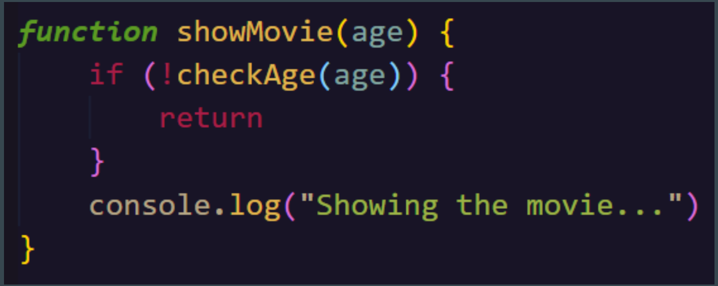


```
function checkAge(age) {
    if (age > 18) {
        return true
    } else {
        return confirm('Do you have your parents permission?')
    }
}

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...")
}
```

- A function with an empty return or without it, returns undefined

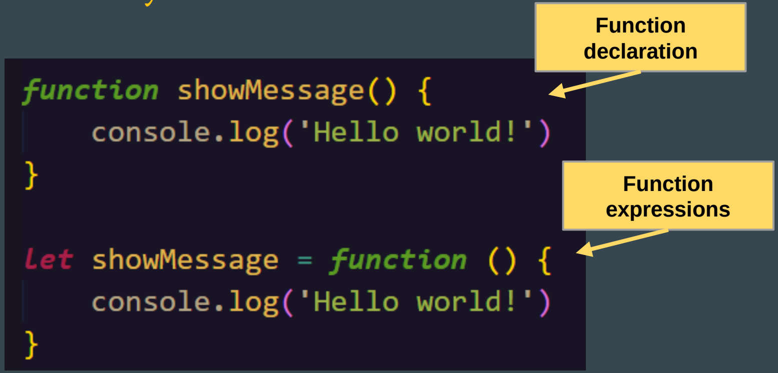
Functions

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



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

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

  - 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

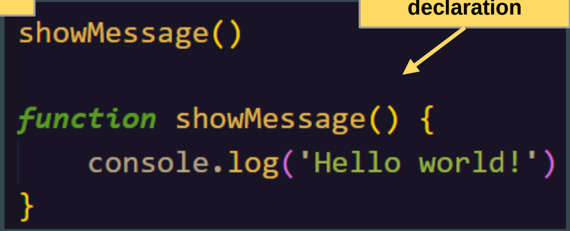
Functions

6. Function expressions

- Differences between function declarations and expressions

It works!

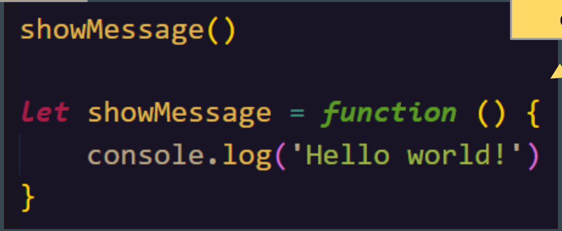
Function declaration



```
showMessage()

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

It does not work!

Function expression



```
showMessage()

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

Functions

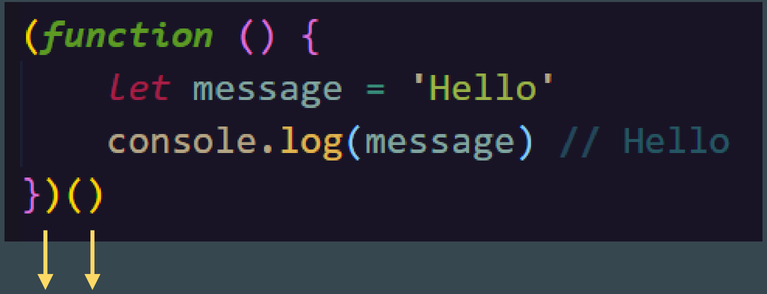
6. Function expressions

- 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 *nameFunction* (...) {...} in the code than let f = *nameFunction* (...) {...}

Functions

6. Function expressions

- Consider function declaration syntax
  - 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

- Arrow function expressions are alternatives to traditional functions that were first introduced in ES6.

- Aside from a relatively concise syntax, arrow functions have a few semantic differences along with some limitations.

```
const functionA = (parameter1, parameter2, ..., parameterN) => {
  // Function body here
}
```

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
```

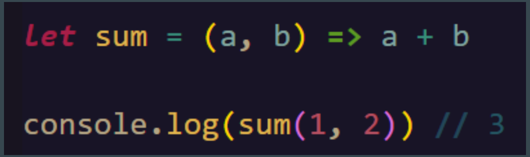
- Explaining the example:
  - Creates a func function that has arguments arg1..argN
  - Evaluates the expression on the right side
  - Returns the result

Functions

7. Arrow functions

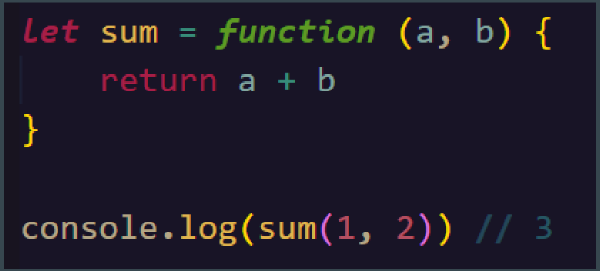
- Example of an arrow function and a similar function expression:

Arrow function



```
let sum = (a, b) => a + b

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

Function expression



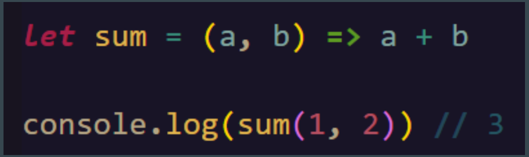
```
let sum = function (a, b) {
    return a + b
}

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

Functions

7. Arrow functions

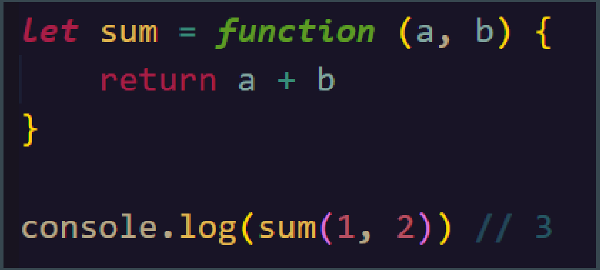
- Removals:
  - *function* word
  - curly braces
  - return word
- Additions:
  - arrow (=>)

Arrow function



```
let sum = (a, b) => a + b

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

Function expression

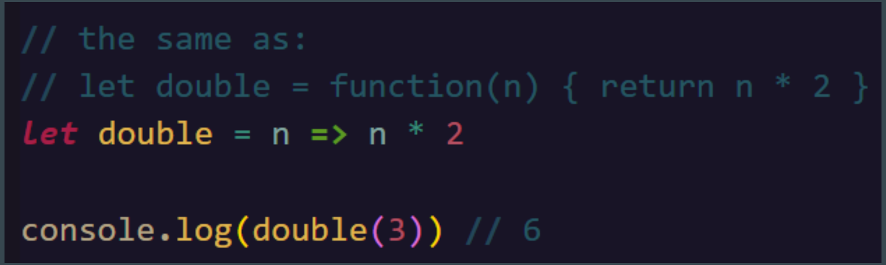


```
let sum = function (a, b) {
    return a + b
}

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:

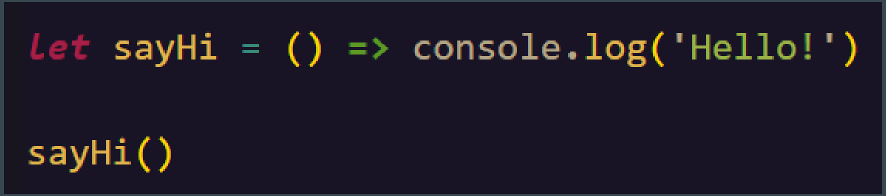


```
// the same as:
// let double = function(n) { return n * 2 }
let double = n => n * 2

console.log(double(3)) // 6
```

Functions

7. Arrow functions

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

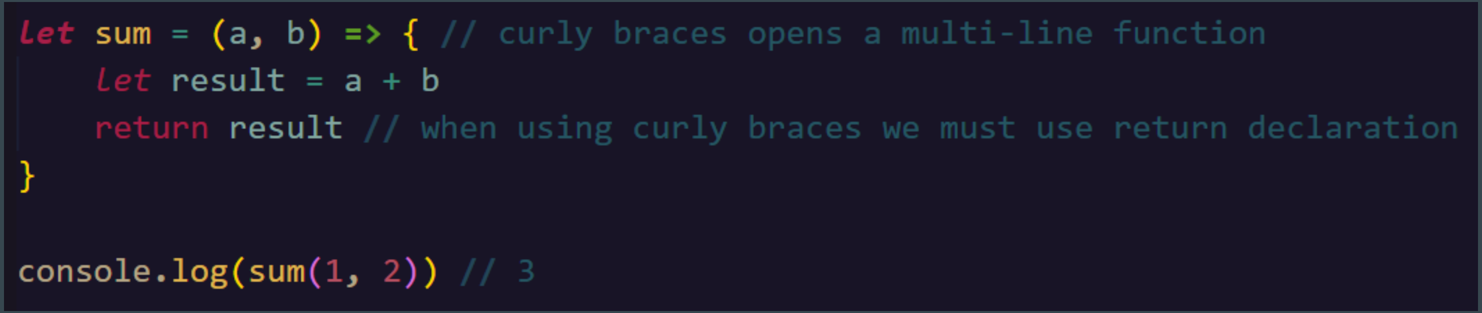


```
let sayHi = () => console.log('Hello!')

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

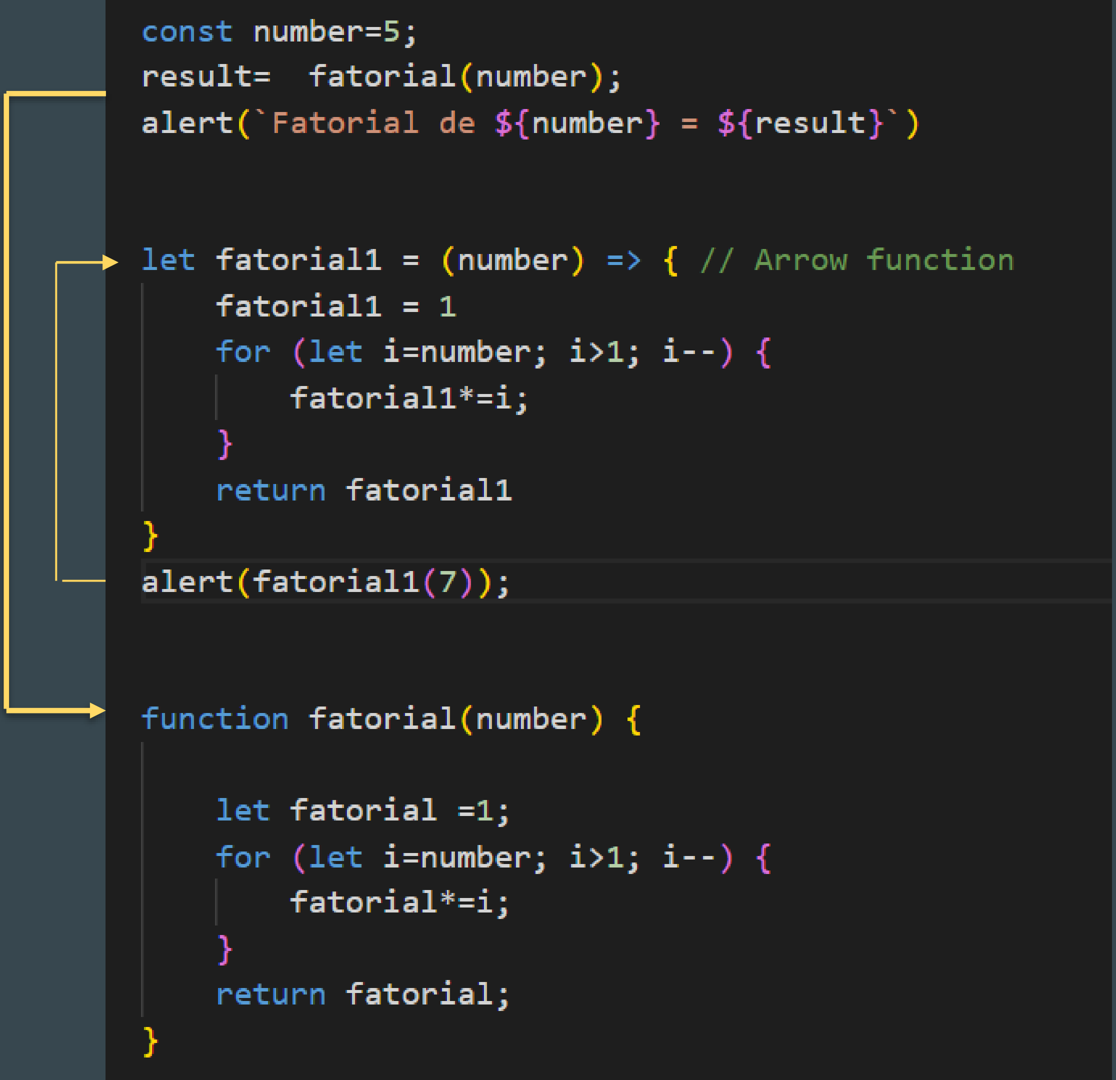
7. Arrow functions

- 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

Functions

7. Examples



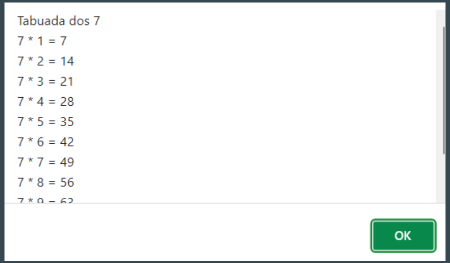
```
const number=5;
result=  fatorial(number);
alert(`Fatorial de ${number} = ${result}`)


let fatorial1 = (number) => { // Arrow function
    fatorial1 = 1
    for (let i=number; i>1; i--) {
        fatorial1*=i;
    }
    return fatorial1
}
alert(fatorial1(7));


function fatorial(number) {

    let fatorial =1;
    for (let i=number; i>1; i--) {
        fatorial*=i;
    }
    return fatorial;
}
```

Functions

7. Sinopse Exercices

- Creates a function **showTabuada()** that takes a number as an argument and prints the table of that number in an alert box

```
showTabuada(7);
```



```
Tabuada dos 7
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
```

- Create a function **numberCities()** that receives a set of cities (number of cities may vary), as input arguments, and prints in an alert box the number of cities visited

```
numberCities('Braga', 'Madrid', 'Aveiro', 'Funchal')
```

Functions

7. Sinopse Exercices

- Change the function to show, now, the names of the cities you received as input

- Create an abbreviated function expression (arrow function) that given a number, it checks if it is a palindrome number.

  If so, it must return true. Otherwise, it must return false. Choose a good name for the function.