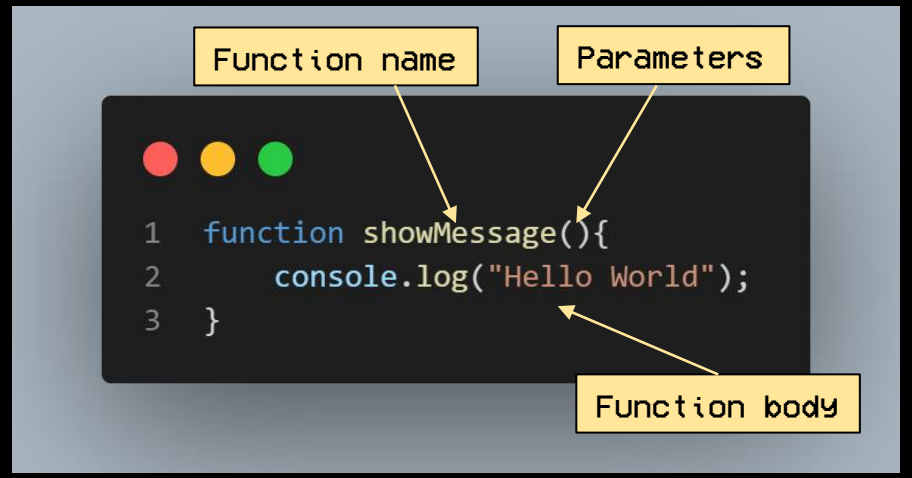
Functions

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

Function declaration

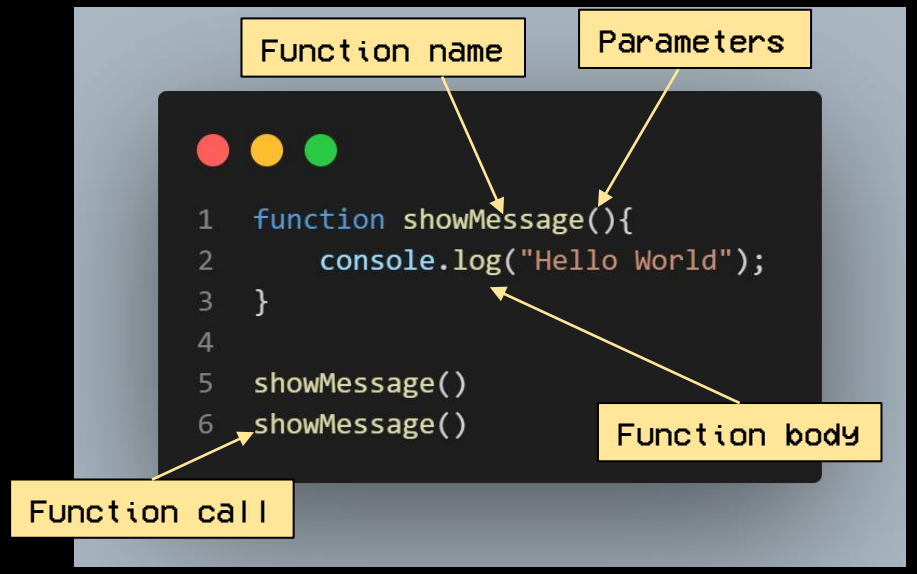
- To create a function, you can use a function declaration



Function declaration

Invoking a Function:

- To **invoke a function**, use the function name with parentheses and parameters (if any)
- This example clearly demonstrates 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 naming

- Functions are action. 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.

Function naming

Functions that begin with…

- “get…” – returns a value
- “calc…” – calculates something
- “create…” – creates something
- “check…” – checks something and returns a Boolean, etc

```
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
```

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

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() {
    userName = "Bob" // modify the global variable userName
    console.log(`Hello, ${userName}`); // Output : 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 change the global variable)
```

Access to local variable

Unchanged global variable

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 receive by) the function

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

showMessage("Ann", "Hello!") // Ouput: Ann: Hello! (*)
showMessage("Ann", "What's up?") // Ouput: 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 functions uses these local variables

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)); // Output 1 + 2 = 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

Parameters

Rest parameters

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

Rest parameters for all the arguments

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

console.log(sum(1)); // Output 1
console.log(sum(1, 2)); // Output 3
console.log(sum(1, 2, 3)); // Output 6
```

Parameters

Default values

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

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

showMessage("John") // Output: John: undefined
```

Parameters

Default values

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

```
function showMessage (from, text = "No text given") {
    console.log(`${from}: ${text}`);
}

showMessage("John") // Output: John: No text given
```

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)

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("Acess granted")
} else {
    alert("Acess denied")
}
```

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