JavaScript functions

Introduction

- JavaScript functions -> **blocks of code** that performs predefined tasks
  - code is reusable
  - can be called from anywhere in your program
  - small task is divided into a function
  - Functions increase readability

- Traditional Function syntax:

```
function name(parameters) {
        statements
    }
```

Invoking a function

- To invoke a function somewhere later in the script:
  - functionname();

Examples

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

  alert( message );
}

showMessage(); // Hello, I'm JavaScript!
```

```
<!DOCTYPE html>
<html>
    <head>
        <script>
            function sayHello() {
            alert("Hello there!");
            }
        </script>

    </head>

    <body>
        <p>Click the button and call the function</p>
        <button onclick="sayHello()">Say Hello</button>

    </body>
</html>
```

Functions & Variables

- A variable declared inside a function -> only visible there!
- An outer variable -> can be accessed by a function!
- An outer variable -> can be modified by a function
- The outer variable is only used if there’s no local one.
- If a same-named variable is **declared** inside the function -> it shadows the outer one, which is ignored.

```
var userName = 'Doe';

function showMessage() {
  var userName = "Aristea";
  var message = 'Hello, ' + userName;
  alert(message);

}
 showMessage(); //Hello, Aristea

 alert( userName ); //Doe
```

What do we expect to see here?

```
var userName = 'Doe';

  function showMessage() {
    userName = "Aristea";
    //var userName = "Aristea";
    var message = 'Hello, ' + userName;
    alert(message);

  }
   showMessage(); //Hello, Aristea

   alert( userName );
```

```
var userName = 'Doe';

function showMessage() {
  var userName = "Aristea";
  var message = 'Hello, ' + userName;
  alert(message);

}
 showMessage(); //Hello, Aristea

 alert( userName ); //Doe
```

We changed the outer variable

```
var userName = 'Doe';

 function showMessage() {
   userName = "Aristea";
   //var userName = "Aristea";
   var message = 'Hello, ' + userName;
   alert(message);

 }
  showMessage(); //Hello, Aristea

  alert( userName );
```

Global variables & Functions

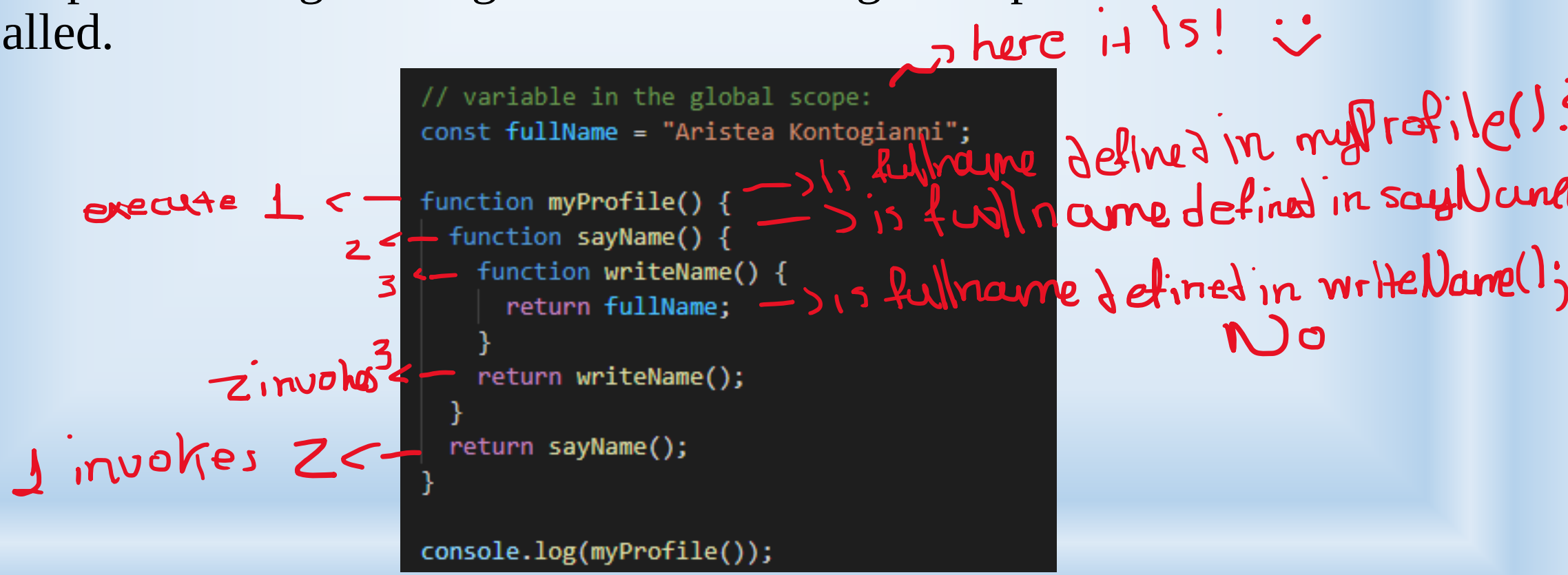
- Variables declared outside of **any** function-> global variables
- Global variables -> visible from any function (unless shadowed by local variables)
- Note: It’s good practice to minimize the use of global variables. Modern code has **few or no** global variables.

Nested Functions

- JavaScript support nested functions
- When functions get particularly complex, it can be helpful to break them into smaller functions
  - clarify exactly what each section of the code is doing.
  - break up long sections into smaller more readable ones
- Creation of a nested function is simple: **declare the function as it is normally done, only within the scope of another function.**

Scope Chain

- JavaScript's scope chain -> determines the hierarchy of places the computer must go through to locate the origin of specific variable called.



In the snippet above

- When myProfile() function gets invoked-> the computer will first invoke the sayName() function
- Secondly, writeName() function is invoked
- At this point, computer will call fullName
- But it will not go directly to the global scope to call fullName. (is fullName defined locally within the writeName() function? etc)

Scope Chain

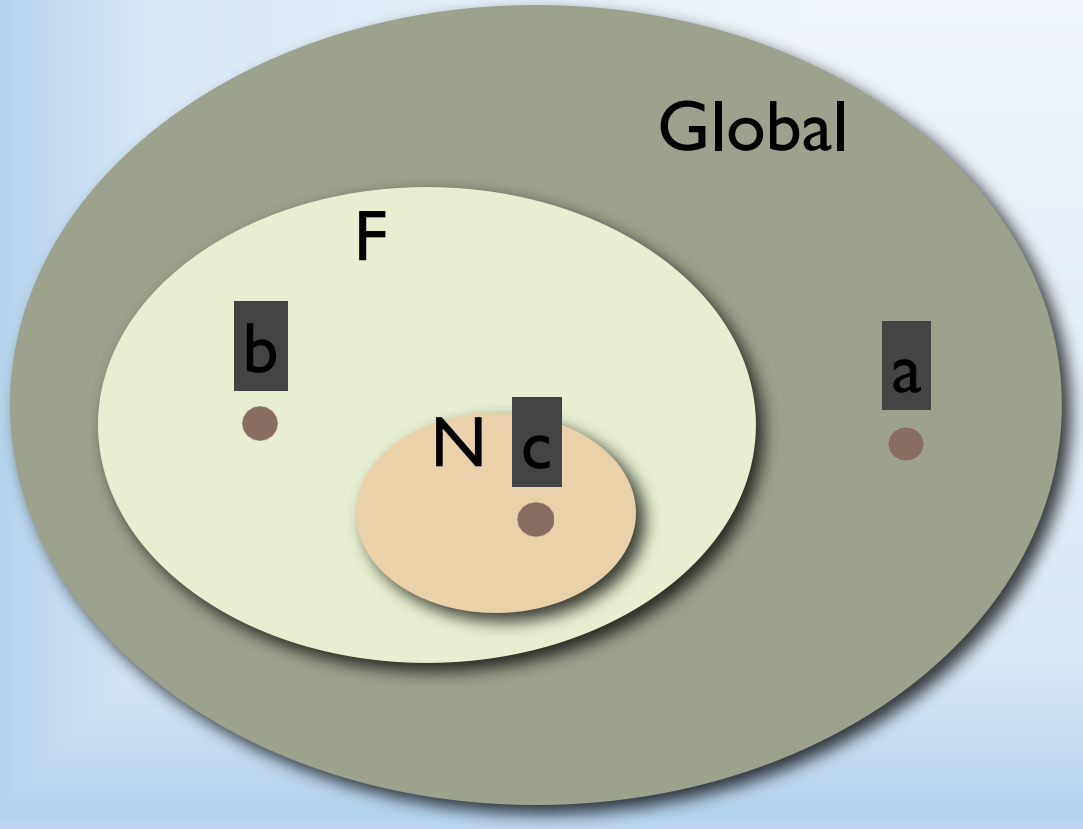
- What do we expect to see in the console for code below?

```
// variable in the global scope:
const fullName = "Aristea Kontogianni";

function myProfile() {
  function sayName() {
    const fullName = "Joe doe";
    function writeName() {
      return fullName;
    }
    return writeName();
  }
  return sayName();
}

console.log(myProfile());
```

Functions: Scope




```
var a;
//..
function F() {
   var b;
   //..
   function N() {
      var c;
      //..
   }
}
```

Lexical scope

- In JavaScript lexical scope is the definition area of an expression.
- Item's lexical scope-> is the place it got created (its definition space)
- The place an item got invoked (or called) is not necessarily the item's lexical scope (Αλλού το δηλώνουμε αλλού το καλούμε)
- Functions create their environment (**scope**) when they are defined, not when they are executed.

What do we expect to see here?

```
function func1(){
    var v=1;
    return func2();
}

function func2(){
    return v;
}

function func3(){
    var v=2;
    return(
        function(){
            return v;
        }
    )();

}
    console.log("one : "+func3());
    console.log("two : "+ func1());
</script>
```

//self invoking function

```
out(func4());
```

function arguments

- Function arguments : parameters passed to functions
- They are listed inside the parentheses () in function definition
- They act as **local variables**

```
<script>
function myFunction(p1, p2) {

  alert(p1+p2);
}
myFunction(2,3);
```

Return

- Functions often compute a return value ->return statement
- What will alert show here?

```
function myFunction(p1, p2) {

  return(p1+p2);
  alert(p1+p2);
}

document.getElementById("demo").innerHTML = myFunction
    (4, 3);
```

Return

- Functions often compute a return value ->return statement
- return statement: stops a function from executing

```
function myFunction(p1, p2) {

  return(p1+p2);
  alert(p1+p2);
}

document.getElementById("demo").innerHTML = myFunction
    (4, 3);
```

Anonymous function

Anonymous functions:

- are defined without using a name
- must be assigned to a variable.
- the var is then used as function

```
<p id="demo"></p>

<script>
var x = function (a, b) {return a * b};
document.getElementById("demo").innerHTML = x(4, 3);

//The function above is actually anonymous function (
a function without a name).

//Functions stored in variables do not need function
names as they are always called using the variable
name!
```

Note that normal functions are run before any other code, meaning they do not have to be declared before the usage of them.

Anonymous functions are created at run time.

Interesting source: https://dev.to/chris_bertrand/coding-concepts---anonymous-methods-a9o

Self-Invoking Functions

- **Function expressions** can be made "self-invoking".

- This means that they can be invoked automatically, without being called!

Syntax

```
(function () {
  document.getElementById("demo").innerHTML = "Hello! I run by myself";
})();
```

More function properties:

- JavaScript functions can be used as **values and in expressions**!

```
function myFunction(a, b) {
  return a * b;
}
var x = myFunction(4, 3);
var x = myFunction(4, 3) * 2;
```

Arrow functions

arrow functions

- alternative to traditional functions
- **can't be** used in all cases

Arrow functions Syntax

- One param

  param => expression

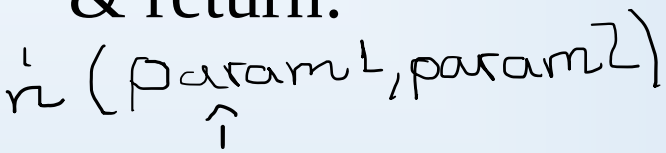
  x => x+10

- Multiple params require ()

  (param1, paramN) => expression

  (x, y, z) => x + y+ z + 10;

- Multiline statements need {} & return:




```
param => {
  let x = 2;
  let y= 3:
  return x*y + param;
}
```

Arrow functions Syntax

- no params

```
var x = 2;
var y= 3:
() => x + y + 100;
```

- Default parameters :

- (x=2, y=3, c) => x + y + c*100

Arrow functions

Difference in syntax example:

```
// Traditional Function
function nameme (y){
  return y + 100;
}

// Arrow Function
let nameme = y => y + 100;
```

```
<script>
var result1,result2;

  // Traditional Anonymous Function
  result1 = function (a){
    return a + 100;
  }

  // Arrow Function
  result2 = a => a + 100;

  console.log(result1(10)+" "+result2(20)); // 110 120

</script>

</body>
```

We return only one thing, no other commands, so syntax can be that simple

Conditions

if statement

- if statement -> the most common type of condition
- if statement runs only if the condition enclosed in parentheses () is ***truthy***
- extend if statement with an **else** statement -> adds another block to run when the if conditional doesn’t pass
- **else if** statement, which adds another condition with its own block

truthy value

- In JavaScript, a *truthy* value is a value that is considered **true** when encountered in a **Boolean context**

- Truthy= not falsy

JS falsy values

Some **falsy values**:

- false
- 0 and -0
- "“: Empty string
- Null: absence of any value
- undefined
- NaN: Not a Number

What will outcome be?

```
var outcome ;
if (false) {
        outcome = "if block";
} else if (true) {
        outcome = "first else if block";
} else if (true) {
        outcome = "second else if block";
} else {
        outcome = "else block";
}
console.log(outcome);
```

Logical Operators

- Logical operators are widely used in conditions

- && and
- || or
- ! not

&&

- **Both statements must be true in order to return true**
- true && true // t && t returns true
- true && false // t && f returns false
- false && (3 == 4) // f && f returns false
- 'Cat' && 'Dog' // t && t returns "Dog“

  Returns expr1 if it can be converted to false; otherwise, returns expr2. Since a non-empty string can't be converted to false, it will return dog

&&

- false && 'Cat' // f && t returns false
- 'Cat' && false // t && f returns false

||

- The logical OR (||) operator is true **if and only if one or more of its operands is true**
- true || true // t || t returns true
- false || true // f || t returns true
- true || false // t || f returns true
- false || (1 == 4) // f || f returns false
- 'Cat' || 'Dog' // t || t returns "Cat"
- false || 'Cat' // f || t returns "Cat"
- 'Cat' || false // t || f returns "Cat"

- Note: The OR (||) operation returns the **first truthy value** or the **last** value if **no** truthy value is found

!

- !true // !t returns false
- !false // !f returns true
- !’’ // !f returns true Empty string is **falsy** value
- !'Cat' // !t returns false

Operator precedence

- The sequence that logical operators are executed is the following:

- ! && ||

What console log will display below

```
const a = true , b = false , c = false;
var d =  a && b || a && c;
console.log(d);


var z =  !b && a || a && c;
console.log(z);
```

What console log will display below

```
//logical operator presedence
const a = true , b = false , c = false;
var d =  a && b || a && c;
//a && b => false
//a && c => false
//false
console.log(d);


var z =  !b && a || a && c;

//!b && a true
//a && c false
//true

console.log(z);
```

What console log will display below

```
const a = true , b = false , c = false;
```

```
d =  a && ! b || a ;
```