Closures

MDN DEFINITION

- A closure gives you access to an outer function's scope from an inner function.

- In JavaScript, closures are created every time a function is created, at function creation time.

- Έστω ότι έχουμε αυτό το παράδειγμα.
- Ξέρουμε ότι ένα function έχει πρόσβαση στις μεταβλητές που ορίζονται σε outer function. Είναι λογικό να βλέπουμε αυτό το αποτέλεσμα

```
function person(name) {

    function hiFriends(){
        console.log(`hi, I am ${name}`);
    }
    function iamHappy(){
        console.log(` ${name} is happy`);
    }
    hiFriends();
    iamHappy();
}
person("aristea")
person("maria")
</script>
```

```
hi, I am aristea
 aristea is happy
hi, I am maria
 maria is happy
>
```

- Αν τώρα θέλω να «τρέχουν» οι inner functions κάποια άλλη στιγμή στον κώδικά μου…
- Στο παρακάτω παράδειγμα φαίνεται η σημασία των closures.
- Τα closures «συγκρατούν - θυμούνται» το outer function scope ακόμα και μετά το creation time του.

```
//person function is actually a factory function that creates objects
function person(name) {

    function hiFriends(){
        console.log(`hi, I am ${name}`);
    }
    function iamHappy(){
        console.log(` ${name} is happy`);
    }
    return{
        hiFriends,
        iamHappy
    }

}
//once person function is assigned to variables it is called immediatelly
const aristea= person("aristea")
const maria = person("maria")
/**
The closure is that inner functions remember and has access to the name parameter
we originally set with person even though we have already
executed the person function in the past

Inner function is holding on the value of name for future use
maintains a reference to its lexical environment,
within which the variable name exists
**/
aristea.hiFriends();
maria.iamHappy();
```

```
hi, I am aristea
maria is happy
```

A **closure** is created **when a function is defined within another function**, allowing the inner function to **access variables** from the **outer** (enclosing) **function's scope**. Closures have the ability to "**remember**" the environment in which they were created, even after the **outer function has finished executing.**

Here's a concise explanation of closures:

Definition: A closure is the combination of a function and the lexical(place where code is writen) environment within which that function was declared.

Key Points:

- Closures allow inner functions to access variables from outer functions even after the outer function **has completed execution**.

- The **inner function "closes over" the variables it uses**, preserving their values.

Factory functions

- A **factory function** in JavaScript is a **function that returns an object**.
- It is called a "factory" because it's designed to **produce instances of objects.**
    - Just like a car factory would do

```
function createCar(brand, model) {
  return {
    brand: brand,
    model: model,
    start: function() {
      console.log("Engine started for " + this.brand + " " + this.model);
    },
    stop: function() {
      console.log("Engine stopped for " + this.brand + " " + this.model);
    },
  };
}

// Create instances using the factory function
const car1 = createCar("Toyota", "Camry");
const car2 = createCar("Honda", "Civic");

// Use the objects
car1.start(); // Output: Engine started for Toyota Camry
car2.start(); // Output: Engine started for Honda Civic

car1.stop();  // Output: Engine stopped for Toyota Camry
car2.stop();  // Output: Engine stopped for Honda Civic
```

Factory functions

- Simple!
- Easy to read
- No class constructor for example!
- No duplicate
- Data privacy

```
function makeAdder(x) {
    // The outer function takes a

    return function(y) {
        // The inner function is
        // It has access to the pa
        return x + y;
    };
}

// Create two functions using make
const add5 = makeAdder(5);
const add10 = makeAdder(10);

// Call the created functions with
console.log(add5(2)); // Output:
console.log(add10(2)); // Output:
```

- makeAdder : takes a parameter x.
- Inside makeAdder, inner function is defined that takes a parameter y.
- inner function is returned from outer function.

- At this point, the inner function has access to the x parameter from the outer function's scope, even though the outer function **has already finished executing.** This behavior is known as a closure.
- When you create new functions using makeAdder and assign them to add5 and add10, they "remember" the value of x that was passed during their creation.

- So:
- add5 effectively becomes a function that adds 5 to its argument.
- add10 becomes a function that adds 10 to its argument.

- When you later call add5(2), it's equivalent to **makeAdder(5)(2),** and it **returns 5 + 2,** which is 7. Similarly, add10(2) is equivalent to **makeAdder(10)(2),** and it returns **10 + 2,** which is 12.

UNIVERSITY OF PIRAEUS

SOURCE

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures