

# Τεχνολογία Διαδικτύου

## 6. Javascript 3

Γρηγόρης Τζιάλλας

Καθηγητής

Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Σχολή Θετικών Επιστημών

Πανεπιστήμιο Θεσσαλίας

# Functions

# Οι συναρτήσεις ως αντικείμενα

- Οι συναρτήσεις στη JavaScript είναι αντικείμενα (first class objects).
  - Μπορούν να αποθηκευτούν σε μεταβλητές
  - Μπορούν να περαστούν ως παράμετροι
  - Έχουν ιδιότητες, όπως τα άλλα αντικείμενα
  - Μπορούν να οριστούν ανώνυμα, χωρίς αναγνωριστικό.
- Οι συναρτήσεις είναι αντικείμενα τύπου Function, και διαθέτουν ιδιότητες και μεθόδους, όπως:
  - name
  - toString()
  - call()
  - Ο τελεστής () απλά καλεί την call()
- Οι συναρτήσεις σχετίζονται με εκτελέσιμο κώδικα και μπορούν να κληθούν (εκτελεσθούν)

*functionName(); ή  
functionName.call();*

# Ορισμός συναρτήσεων

Οι συναρτήσεις μπορούν να ορισθούν:

- Με δήλωση

```
function functionName(parameters) {  
    // code to be executed  
}
```

- Με έκφραση

```
const functionName = function (parameters) {  
    // code to be executed  
}
```

- Με την δημιουργία νέου αντικειμένου Function

```
const functionName = new Function (parameters, code to be executed)
```

# Ανώνυμες συναρτήσεις

- Ανώνυμες συναρτήσεις καλούνται οι συναρτήσεις που δεν έχουν όνομα (πχ. ορίζονται χωρίς να δηλωθεί κάποιο όνομα μετά την λέξη κλειδί function)

```
let x = function (a) { return a*a; }
```

## Arrow functions

- Είναι μία σύντομη μορφή συνάρτησης όπου παραλείπεται η λέξη function και αντικαθίσταται με βέλος. Επίσης μπορεί να παραληφθούν το return και τα άγκιστρα για απλές εκφράσεις. Οι συναρτήσεις αυτές δεν γίνονται hoisted και πρέπει να δηλώνονται πριν την χρήση τους.

```
πχ.    let x = (a) => { return a*a; } ή και  
        let x = (a) => a*a;
```

# Παράμετροι συναρτήσεων

- Οι παράμετροι μιας συνάρτησης δεν καθορίζουν τύπο δεδομένων.
- Δεν γίνεται έλεγχος του τύπου των ορισμάτων κατά την κλήση μιας συνάρτησης.
- Δεν ελέγχεται ο αριθμός των παραμέτρων σε μία κλήση.
- Αν μια συνάρτηση καλείται με μικρότερο αριθμό ορισμάτων από αυτά που έχουν δηλωθεί, οι τιμές που λείπουν έχουν την τιμή `undefined`
- Μπορεί να γίνεται προγραμματιστικά έλεγχος των τιμών των ορισμάτων και να τους δίνεται μια αρχική τιμή αν απαιτείται

```
function myFunction(x, y) {  
  if (y === undefined) {  
    y = 0;  
  }  
}
```

# Παράμετροι συναρτήσεων

- Καθορισμός αρχικής τιμή παραμέτρων
- Παλιός τρόπος

```
function oldWayFunction(a, b) {  
    a = a || 1;  
    b = b || "Hello";  
    console.log(a,b);  
}
```

- Νέος τρόπος

```
function newWayFunction(a=1, b="Hello")  
{  
    console.log(a,b);  
}
```

# Το αντικείμενο arguments

- Οι συναρτήσεις JavaScript έχουν ένα ενσωματωμένο αντικείμενο με όνομα arguments το οποίο περιέχει όλες τις παραμέτρους της κλήσης.
- Μια συνάρτηση μπορεί να δέχεται ένα μεταβλητό πλήθος παραμέτρων, τις οποίες δεν καθορίζει στην δήλωση της, και να έχει πρόσβαση σε αυτές με το αντικείμενο arguments

```
function mySum(){  
    let sum=0;  
    for (let anArg of arguments){  
        sum += anArg;  
    }  
    return sum;  
}  
console.log("Sum = ",mySum(1,2,3,4,5));
```



# Κλήση με τιμή ή αναφορά (Call by value and call by reference)

Οι βασικοί τύποι δεδομένων όταν χρησιμοποιούνται ως παράμετροι μιας συνάρτησης δεν αλλάζουν γιατί η συνάρτηση χρησιμοποιεί την τιμή της μεταβλητής (call by value).

Τα αντικείμενα όμως, αν τροποποιηθούν από την συνάρτηση, αλλάζουν γιατί η συνάρτηση χρησιμοποιεί την αναφορά στο αντικείμενο (call by reference)

```
function incInteger(n) {  
    n = n + 1;  
}  
function incArrayInteger(n) {  
    n[0] = n[0] + 1;  
}  
let i = 2;  
console.log(`before incInteger  ${i}`);  
incInteger(i);  
console.log(`after incInteger  ${i}`);  
let m = [i];  
console.log(`before incArrayInteger  ${m[0]}`);  
incArrayInteger(m);  
console.log(`after incArrayInteger  ${m[0]}`);
```

```
before incInteger  2  
after incInteger  2  
before incArrayInteger  2  
after incArrayInteger  3
```

# Η λέξη κλειδί this

- Έχει διαφορετικές τιμές ανάλογα με το πού χρησιμοποιείται:
  - Σε μια μέθοδο, το this αναφέρεται στο αντικείμενο ιδιοκτήτη .
  - Σε μια συνάρτηση, το this αναφέρεται στο καθολικό αντικείμενο.
  - Χωρίς να ανήκει κάπου, το this αναφέρεται στο καθολικό (global) αντικείμενο.
  - Σε μια συνάρτηση, σε αυστηρή λειτουργία (“use strict”), το this είναι undefined.
  - Σε ένα γεγονός, το this αναφέρεται στο στοιχείο που έλαβε το γεγονός.
  - Στις μεθόδους call() και apply() το this μπορεί να ορισθεί να αναφέρεται σε οποιοδήποτε αντικείμενο.

# Παράδειγμα τιμών this

```
//this alone
let k = this;
console.log("This Alone ", k);

//this in a function
function myThis() {
  console.log("This in a function ", this);
}

//in an method
const nick = {
  name: "Nick",
  sayThis: function () {
    console.log("This in a method ", this);
  }
}
myThis();
nick.sayThis();
function handleClick(aParam) {
  console.log("This passed to event handler", aParam);
}
```

This Alone ▶ Window

This in a function ▶ Window

This in a method ▶ Object

Live reload enabled.

This in an event [object HTMLButtonElement]

This passed to event handler

```
<button onclick="handleClick(this);"> Click to s
</button>
```

```
<div>
  <!-- in an event -->
  <button onclick="console.log('This in an event ' + this);">
    Click to log this
  </button>

  <!-- in an event passed to event handler-->
  <button onclick="handleClick(this);">
    Click to send this
  </button>
```

# Η μέθοδος call

- Με τη μέθοδο call(), ένα αντικείμενο μπορεί να χρησιμοποιήσει μια μέθοδο ενός άλλου αντικειμένου.
- Η μέθοδος call() καλεί μία μέθοδο ή συνάρτηση με πρώτη παράμετρο την τιμή της λέξης κλειδί this και στην συνέχεια τις υπόλοιπες παραμέτρους τις οποίες δέχεται η μέθοδος ή συνάρτηση.

```
const person = {  
  info: function () {  
    return this.name + " " + this.email;  
  }  
}  
const person1 = {  
  name: "John",  
  email: "john@gmail.com"  
}  
console.log(person.info.call(person1));
```

# Η μέθοδος apply

- Παρόμοια με την μέθοδο call() με την διαφορά ότι οι παράμετροι, εκτός από την τιμή του this, περνούν ως πίνακας.
- Είναι χρήσιμη σε περιπτώσεις που θέλουμε να χρησιμοποιήσουμε πίνακα για να καλέσουμε μέθοδο που δέχεται πολλές μεταβλητές.

```
const numbers = [1,2,9,3,23,4,5];  
//passing null for value of this  
let max= Math.max.apply(null,numbers)  
console.log(max);
```

Στο συγκεκριμένο παράδειγμα, η μέθοδος max δέχεται ένα μεταβλητό αριθμό παραμέτρων και όχι πίνακα. Με την μέθοδο apply την καλούμε με πίνακα. Ως τιμή του this δίνουμε το null γιατί η μέθοδος δεν χρησιμοποιεί το this

# Η μέθοδος bind

- Με την χρήση των μεθόδων call και apply η τιμή του this είναι προσωρινή και ισχύει για μία μόνο κλήση χωρίς να επηρεάζει την μέθοδο στην οποία εφαρμόζεται.
- Η μέθοδος bind() συνδέει μόνιμα την τιμή του this σε μία νέα συνάρτηση που επιστρέφει.
- Με την μέθοδο bind μπορούμε να «δανείσουμε» μία μέθοδο ενός αντικειμένου σε κάποιο άλλο η και να καθορίσουμε / διασφαλίσουμε την τιμή της λέξης κλειδί this σε κάποια μέθοδο.

# Παράδειγμα bind

```
function talk() {  
  let anElement = document.getElementById("output");  
  anElement.textContent = `I am ${this.name}`;  
}  
let nick = {name: "Nick" };;  
let petros = { name: "Petros" };  
let mary = {name: "Mary" };  
let talkPetros = talk.bind(petros);  
nick.talk = talk.bind(nick);  
//we cannot change the binding  
//of a method already binded  
let talkMary = nick.talk.bind(mary);
```

```
<h1>Μέθοδος Bind</h1>  
<button onclick="nick.talk();">Nick Talk</button>  
<button onclick="talkPetros();">Petros Talk</button>  
<button onclick="talkMary();">Mary Talk</button>  
<div id="output"></div>
```

Στο συγκεκριμένο παράδειγμα, η μέθοδος bind συνδέει την τιμή του this της συνάρτησης talk με τα αντικείμενα nick και petros.

Το bind δεν μπορεί να εφαρμοστεί σε μία μέθοδο ή συνάρτηση που έχει ήδη συνδεθεί με bind (πχ nick.talk). Η τιμή του this παραμένει ή ίδια.

# Χρήση συναρτήσεων ως callbacks

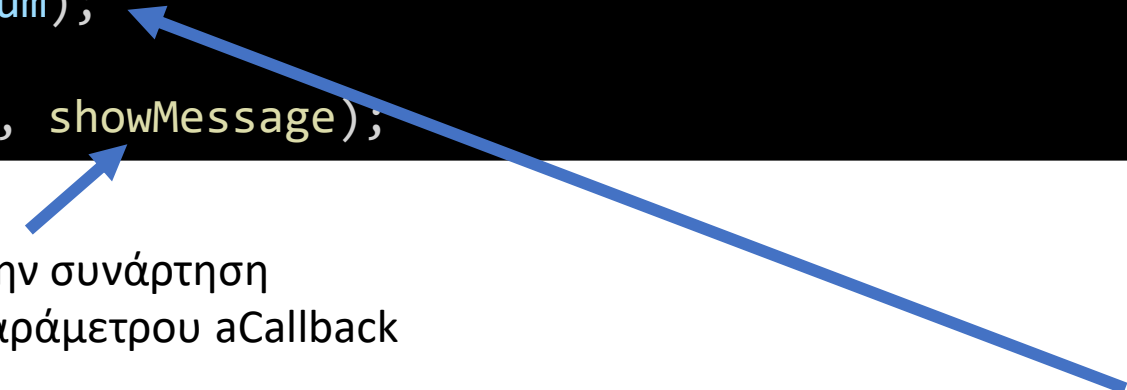
- Οι συναρτήσεις μπορούν να ορισθούν ως παράμετροι άλλων συναρτήσεων (callbacks).
- Οι παράμετροι της callback τροφοδοτούνται από την εξωτερική συνάρτηση
- Στο παρακάτω παράδειγμα η συνάρτηση onClick ορίζεται ως παράμετρος της μεθόδου addEventListener για να κληθεί όταν συμβεί κάποιο γεγονός

```
function onClick() {  
    console.log("Button Clicked");  
}  
const buttonClick = document.querySelector("#ButtonClick");  
buttonClick.addEventListener('click', onClick);
```



# Παράδειγμα callback

```
function showMessage(aMessage) {  
    document.getElementById("output").innerHTML = aMessage;  
}  
  
function addNumbers(num1, num2, aCallback) {  
    let sum = num1 + num2;  
    aCallback(sum);  
}  
addNumbers(5, 5, showMessage);
```

A blue arrow originates from the 'showMessage' argument in the 'addNumbers(5, 5, showMessage);' line and points to the 'aCallback' parameter in the 'function addNumbers' definition. Another blue arrow originates from the 'aCallback(sum);' line and points to the same 'aCallback' parameter in the function definition.

Η addNumbers καλείται με την συνάρτηση showMessage ως τιμή της παράμετρου aCallback

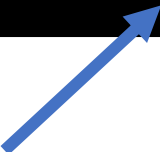
```
<body>  
  <h1>Callback</h1>  
  <div id="output"></div>  
</body>
```

Στην συγκεκριμένη κλήση καλείται η showMessage (τιμή της παραμέτρου aCallback) για να εμφανίσει το άθροισμα.

# Χρήση των callbacks

- Τα callbacks χρησιμοποιούνται κυρίως στον ασύγχρονο προγραμματισμό όταν μια συνάρτηση περιμένει το αποτέλεσμα κάποιας άλλης

```
function showMessage(aMessage) {  
    document.getElementById("output").innerHTML = aMessage;  
}  
function displayHello(){  
    showMessage("Hello with 3 seconds delay");  
}  
setTimeout(displayHello, 3000);
```



Η συνάρτηση setTimeout καλεί την συνάρτηση που ορίζεται από την πρώτη παράμετρο της μετά από καθυστέρηση (πχ 3000 ms) που ορίζεται από την δεύτερη παράμετρο της

```
<body>  
  <h1>Callback Timer</h1>  
  <div id="output">  
    Περιμένετε ..  
  </div>  
</body>
```

# Παράδειγμα callbacks και τιμών this

Η λέξη κλειδί this μετά από callback δεν αναφέρεται στο αντικείμενο αλλά στο αντικείμενο window

Όταν η συνάρτηση είναι arrow function, η λέξη κλειδί δεν επηρεάζεται και συνεχίζει να είναι το αντικείμενο petros στην συγκεκριμένη περίπτωση

Εναλλακτικά για να καθορίσουμε την μεταβλητή this μπορούμε να χρησιμοποιήσουμε κάποια άλλη μεταβλητή στην οποία θα δώσουμε την τιμή του this κατά την δημιουργία του αντικειμένου.

```
function showMessage(aMessage) {
  let anElement = document.getElementById("output");
  anElement.innerHTML += `<div> ${aMessage} </div>`;
}

const nick = {
  name: "Nick",
  talk() {
    setTimeout(function () {
      console.log(this);
      showMessage(this.name);
    }, 1000)
  }
}

const petros = {
  name: "Petros",
  talk() {
    setTimeout(() => {
      console.log(this);
      showMessage(this.name);
    }, 2000)
  }
}

const mary = {
  name: "Mary",
  talk() {
    let myself = this;
    setTimeout(() => {
      console.log(myself);
      showMessage(myself.name);
    }, 3000)
  }
}

nick.talk();
petros.talk();
mary.talk();
```

# CLOSURES

# Closures - Κλεισίματα

- Τα κλεισίματα συνήθως εμφανίζονται σε γλώσσες στις οποίες οι συναρτήσεις είναι τιμές / αντικείμενα πρώτης τάξης—σε αυτές τις γλώσσες μια συνάρτηση μπορεί να περαστεί ως παράμετρος, να επιστραφεί από κλήση άλλης συνάρτησης, να δεσμευτεί σε κάποιο όνομα μεταβλητής, κ.λπ., ακριβώς όπως οι απλούστεροι τύποι (π.χ. συμβολοσειρές, ακέραιοι).
- Η ιδέα των κλεισιμάτων αναπτύχθηκε κατά τη δεκαετία του 1960 και υλοποιήθηκε αρχικά στη γλώσσα προγραμματισμού Scheme. Από τότε έχουν σχεδιαστεί πολλές γλώσσες που υποστηρίζουν κλεισίματα.
- Σε κάποιες γλώσσες, ένα κλείσιμο μπορεί να προκύψει όταν μια συνάρτηση ορίζεται μέσα σε μια άλλη συνάρτηση και η εσωτερική αναφέρεται σε μεταβλητές της εξωτερικής.
- Στο χρόνο εκτέλεσης, όταν εκτελείται η εξωτερική συνάρτηση, δημιουργείται ένα κλείσιμο, που αποτελείται από την εσωτερική συνάρτηση και τις αναφορές σε μεταβλητές της εξωτερικής οι οποίες χρειάζονται (αυτές οι τιμές ονομάζονται *upvalues* του κλεισίματος).

# Χρήση Closures

- Τα κλεισίματα συνήθως υλοποιούνται με μια ειδική δομή δεδομένων που περιέχει ένα δείκτη στον κώδικα της συνάρτησης και μια αναπαράσταση του λεκτικού περιβάλλοντος της συνάρτησης (π.χ. το σύνολο των διαθέσιμων μεταβλητών και τις τιμές τους) τη στιγμή που δημιουργήθηκε το κλείσιμο.
- Τα κλεισίματα έχουν πολλές χρήσεις:
  - Επιτρέπουν την προσαρμογή της συμπεριφοράς συναρτήσεων. Για παράδειγμα, μια συνάρτηση που ταξινομεί τιμές μπορεί να δεχτεί μια παράμετρο κλεισίματος που συγκρίνει τις προς ταξινόμηση τιμές σύμφωνα με ένα κριτήριο που ορίζει ο χρήστης.
  - Τα κλεισίματα μπορούν να χρησιμοποιηθούν για τον ορισμό δομών ελέγχου. Για παράδειγμα, όλες οι τυπικές δομές ελέγχου της Smalltalk, συμπεριλαμβανομένων των διακλαδώσεων (if/then/else) και των βρόχων (while και for), ορίζονται χρησιμοποιώντας αντικείμενα των οποίων οι μέθοδοι δέχονται closures. Οι χρήστες μπορούν επίσης εύκολα να ορίσουν τις δικές τους δομές ελέγχου.
  - Μπορούν να παραχθούν πολλαπλές συναρτήσεις που κλείνουν πάνω από το ίδιο περιβάλλον, επιτρέποντάς τους να επικοινωνούν ιδιωτικά μεταβάλλοντας αυτό το περιβάλλον (σε γλώσσες που επιτρέπουν την ανάθεση).

# Mozilla definition of closure

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, 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.

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

# Δημιουργία κλεισίματος

- Μία συνάρτηση ορίζει μία εσωτερική (εμφωλευμένη) συνάρτηση την οποία επιστρέφει.
- Η εσωτερική συνάρτηση έχει πρόσβαση όχι μόνο στις δικές της τοπικές μεταβλητές αλλά και στις μεταβλητές της γονικής συνάρτησης που την περιέχει και τις καθολικές μεταβλητές.
- Η κλήση της γονικής συνάρτησης επιστρέφει την εσωτερική συνάρτηση, η οποία διατηρεί τις τιμές των μεταβλητών στις οποίες είχε πρόσβαση κατά την κλήση της γονικής συνάρτησης.
- Το κλείσιμο είναι η εσωτερική συνάρτηση που έχει πρόσβαση στις γονικές μεταβλητές, ακόμη και μετά τη λήξη της γονικής συνάρτησης.



# Παράδειγμα closure (1)

```
const add = (function () {  
  let counter = 0;  
  return function () {  
    counter += 1;  
    console.log(counter);  
    return counter;  
  };  
})();  
add();  
add();  
add();
```

## Παράδειγμα closure adder (2)

```
function makeAdder(x) {  
    return function (y) {  
        return x + y;  
    };  
}  
const add5 = makeAdder(5);  
const add10 = makeAdder(10);  
console.log(add5(2)); // 7  
console.log(add10(2)); // 12
```


In this example, we have defined a function `makeAdder(x)`, that takes a single argument `x`, and returns a new function. The function it returns takes a single argument `y`, and returns the sum of `x` and `y`.

In essence, `makeAdder` is a function factory. It creates functions that can add a specific value to their argument. In the above example, the function factory creates two new functions—one that adds five to its argument, and one that adds 10.

# Παράδειγμα πολλαπλών κλεισιμάτων

Η γονική συνάρτηση, κάθε φορά που καλείται, επιστρέφει ένα αντικείμενο με 3 μεθόδους οι οποίες λειτουργούν ως πολλαπλά κλεισίματα.

Η μεταβλητή `privateCounter`, διαφορετική για κάθε κλήση, είναι ιδιωτική μεταβλητή και προσβάσιμη μόνο από τις μεθόδους `increment`, `decrement` και `value`.



```
const makeCounter = function () {  
  let privateCounter = 0;  
  function changeBy(val) {  
    privateCounter += val;  
  }  
  return {  
    increment() {  
      changeBy(1);  
    },  
  
    decrement() {  
      changeBy(-1);  
    },  
  
    value() {  
      return privateCounter;  
    },  
  };  
};  
  
const counter1 = makeCounter();  
const counter2 = makeCounter();  
console.log(counter1.value()); // 0.  
counter1.increment();  
counter1.increment();  
console.log(counter1.value()); // 2.  
counter1.decrement();  
console.log(counter1.value()); // 1.  
console.log(counter2.value()); // 0.
```

# Currying

- Currying: Η ανάλυση μια συνάρτησης με πολλές παραμέτρους σε ακολουθίες συναρτήσεων μίας μόνο μεταβλητής.
- Προς τιμήν του Haskell Brooks Curry που μελέτησε μαθηματικά το θέμα: «Οι συναρτήσεις πολλών μεταβλητών μπορούν να αναλυθούν σε ακολουθίες συναρτήσεων μίας μόνο μεταβλητής»

```
const addNumbers = function (x, y, z) {  
  return x + y + z;  
}  
console.log(addNumbers(1,2,3));
```



```
const addNumbersCurry = function (x) {  
  return function (y) {  
    return function (z) {  
      return x + y + z;  
    }  
  }  
}  
console.log(addNumbersCurry(1)(2)(3));
```

# Παράδειγμα Currying

```
const flavors =  
  ['vanilla', 'chocolate', 'strawberry', 'green tea'];  
  
function createFlavorTest(flavor) {  
  function isFlavor(element) {  
    return element === flavor;  
  }  
  return isFlavor;  
}  
  
const isStrawberry = createFlavorTest('strawberry');  
const indexOfFlavor = flavors.findIndex(isStrawberry);  
console.log("Index of strawberry = ",indexOfFlavor);
```

<https://web.stanford.edu/class/archive/cs/cs193x/cs193x.1176/lectures/16/lecture16.pdf>

# Functional Programming

# Συναρτησιακός προγραμματισμός – Functional Programming

- Στην επιστήμη υπολογιστών, συναρτησιακός προγραμματισμός είναι ένα προγραμματιστικό παράδειγμα που αντιμετωπίζει τον υπολογισμό ως την αποτίμηση μαθηματικών συναρτήσεων
- Ο συναρτησιακός προγραμματισμός έχει τις ρίζες του στο λογισμό λάμδα, ένα τυπικό σύστημα που αναπτύχθηκε τη δεκαετία 1930 για τη διερεύνηση του ορισμού συναρτήσεων, της εφαρμογής συναρτήσεων και της αναδρομής. Πολλές συναρτησιακές γλώσσες προγραμματισμού μπορούν να θεωρηθούν επεκτάσεις του λογισμού λάμδα.

[https://el.wikipedia.org/wiki/Συναρτησιακός\\_προγραμματισμός](https://el.wikipedia.org/wiki/Συναρτησιακός_προγραμματισμός)

- Πρακτικά, η διαφορά μεταξύ μιας μαθηματικής συνάρτησης και της έννοιας της "συνάρτησης" που χρησιμοποιείται στον προστακτικό προγραμματισμό είναι ότι οι προστακτικές συναρτήσεις μπορούν να έχουν παρενέργειες, αλλάζοντας την τιμή των ήδη αποτιμημένων υπολογισμών.

# Λογισμός λάμδα - Alonzo Church, 1930

expression: <name> | <abstraction> | <application>

|             |                       |
|-------------|-----------------------|
| abstraction | συναρτησιακή αφαίρεση |
|-------------|-----------------------|

|             |                       |
|-------------|-----------------------|
| application | συναρτησιακή εφαρμογή |
|-------------|-----------------------|

Name --> έκφραση μεταβλητή για την αναπαράσταση μίας τιμής

Abstraction --> έκφραση της μορφής (λ name.expression)

Application --> έκφραση της μορφής (expression expression)



# Εκφράσεις λάμδα

- Στο λογισμό λάμδα, κάθε έκφραση είναι μια μοναδιαία συνάρτηση, δηλαδή συνάρτηση με μόνο ένα όρισμα.
- Όταν μία έκφραση εφαρμόζεται σε μία άλλη (λέμε ότι "καλείται" η συνάρτηση με παράμετρο την άλλη έκφραση), επιστρέφει μία μοναδική τιμή, που λέγεται το αποτέλεσμα της.

- Μια συνάρτηση ορίζεται χωρίς να της δοθεί όνομα από μια έκφραση λάμδα

$f(x) = x + 2$  εκφράζεται στο λογισμό λάμδα ως  $\lambda x. x + 2$

- Η εφαρμογή συνάρτησης ακολουθεί συσχέτιση προς τα αριστερά:  $f\ x\ y = (f\ x)\ y$ .  
Έστω η συνάρτηση που παίρνει μία συνάρτηση ως παράμετρο και την εφαρμόζει στον αριθμό 3 ως εξής:

$\lambda f. f\ 3$ .

Η τελευταία αυτή συνάρτηση μπορεί να εφαρμοστεί στην προηγούμενη "πρόσθεσε δύο" ως εξής:

$(\lambda f. f\ 3)\ (\lambda x. x + 2)$ .

Οι τρεις εκφράσεις:

$(\lambda f. f\ 3)\ (\lambda x. x + 2)$

$(\lambda x. x + 2)\ 3$

$3 + 2$

# Παράδειγμα εκφράσεων και εφαρμογών

def adder =  $\lambda x. \lambda y. (x + y)$

adder 2 =  $\lambda x. \lambda y. (x + y)$  2 =  $\lambda y. (2 + y)$

adder 2 3 =  $\lambda x. \lambda y. (x + y)$  2 3 =  $\lambda y. (2 + y)$  3 =  $2 + 3 = 5$

def applicer =  $\lambda s. \lambda a. \lambda b. (s a b)$

applicer adder =  $\lambda s. \lambda a. \lambda b. (s a b)$  =  $\lambda a. \lambda b. (\lambda x. \lambda y. (x + y) a b)$

applicer adder 3 4 =  $\lambda a. \lambda b. (\lambda x. \lambda y. (x + y) a b)$  3 4 =  
=  $\lambda x. \lambda y x + y$  3 4 = 7

# Παράδειγμα εκφράσεων λάμδα σε Javascript (1)

def adder = λx.(x + 2)

Για είσοδο x επιστρέφει x + 2

```
// def adder = λx.(x + 2)
// adds 2 to input parameter x
let adder = function (x) {
  return x + 2;
};
```

Η adder με συνάρτηση  
βέλους της Javascript

```
let adder_Arrow = x => x + 2;
```

def applicer = λf.(f 3)

Καλεί την συνάρτηση f που δέχεται  
ως είσοδο με παράμετρο εισόδου 3

```
// def applicer = λf.(f 3)
// calls function f with argument 3
let applicer = function (f) {
  return f(3);
};
```

Η applicer με συνάρτηση  
βέλους της Javascript

```
let applicer_Arrow = f => f(3);
```

Κλήση της adder από την applicer

```
let result = applicer(adder); // 5
```

# Παράδειγμα εκφράσεων λάμδα σε Javascript (2)

def adder =  $\lambda x. \lambda y. (x + y)$

Ορίζει έκφραση που επιστρέφει το άθροισμα των μεταβλητών x και y

```
let adder = function (x) {  
    return function (y) {  
        return x + y;  
    };  
};
```

```
const adder_Arrow = x => y => x + y;
```

def applicer =  
     $\lambda f. \lambda a. \lambda b. (f\ a\ b)$

Καλεί την συνάρτηση f που δέχεται με παράμετρο a και στην συνέχεια με b

```
let applicer = function (f) {  
    return function (x) {  
        return function (y) {  
            return f(x)(y);  
        }  
    }  
};
```

```
const applicer1_Arrow = f => x => y => f(x)(y);
```

# Προγραμματιστικό μοντέλο

1. Αγνές Συναρτήσεις ( pure functions )

$$f(x) = x + 1, f(3) = 4$$

2. Σύνθεση συναρτήσεων

3. Αναφορική Ακεραιότητα (Referential Transparency)

Η συνάρτηση μπορεί να αντικατασταθεί με τις τιμές που επιστρέφει

4. Χωρίς κοινή κατάσταση (shared state)

5. Χωρίς αλλαγές κατάστασης (mutating state)

6. Χωρίς παρενέργειες (no side effects)

$$x = 1;$$

$$f(y) = y + x, f(3) = 4, x = 2; f(3) = 5$$

# Αγνές συναρτήσεις (pure function)

- Με τις ίδιες εισόδους παράγουν την ίδια έξοδο
- Χωρίς παρενέργειες (side effects) όπως:
  - Αλλαγή καθολικών ή άλλων μεταβλητών
  - Εκτύπωση στην κονσόλα ή την οθόνη
  - Διάβασμα στοιχείων ή αποστολή στοιχείων σε περιφερειακές συσκευές
  - Αποστολή στοιχείων στο δίκτυο
  - Δημιουργία γεγονότων που προκαλούν κλήσεις άλλων μεθόδων
  - Κλήση άλλων συναρτήσεων που προκαλούν παράπλευρα αποτελέσματα
- Ο συναρτησιακός προγραμματισμός απομονώνει και διαχωρίζεται διαφορετικά τις συναρτήσεις που δεν είναι αγνές και προκαλούν παράπλευρα αποτελέσματα.

# Συναρτήσεις υψηλής τάξης

- Η JavaScript υποστηρίζει συναρτήσεις πρώτης τάξης (first-class functions ή functions as first-class objects), οι οποίες μας επιτρέπουν να χειρισθούμε τις συναρτήσεις ως δεδομένα, να τις συσχετίσουμε με μεταβλητές, να τις χρησιμοποιήσουμε ως παραμέτρους άλλων συναρτήσεων, να τις επιστρέψουμε από μία 'συνάρτηση κ.λπ.
- Συνάρτηση υψηλής τάξης (higher order function) καλείται η συνάρτηση η οποία δέχεται ως παράμετρο κάποια άλλη συνάρτηση ή επιστρέφει κάποια άλλη συνάρτηση ή και τα δύο.
- Με την χρήση συναρτήσεων υψηλής τάξης, ο συναρτησιακός προγραμματισμός δίνει την δυνατότητα ορισμού συναρτήσεων γενικής χρήσης για την επεξεργασία δεδομένων.
  - Ο αντικειμενοστραφής προγραμματισμός ομαδοποιεί τα δεδομένα με μεθόδους. Οι μέθοδοι των αντικειμένων μπορούν να χρησιμοποιηθούν μόνο στα αντικείμενα για τα οποία σχεδιάστηκαν.
  - Στον συναρτησιακό προγραμματισμό η ίδια συνάρτηση μπορεί να χρησιμοποιηθεί σε οποιοδήποτε αντικείμενο ή τύπο δεδομένων γιατί μπορεί να δεχθεί ως παράμετρο μια άλλη συνάρτηση η οποία θα χειρίζεται τους διαφορετικούς τύπους αντικειμένων και δεδομένων.

# Πλεονεκτήματα χρήσης συναρτησιακού προγραμματισμού

- Προβλεψιμότητα - Predictability
  - Οι καθαρές συναρτήσεις, για την ίδια είσοδο επιστρέφουν το ίδιο αποτέλεσμα
- Εύκολες στην δοκιμή - Testability
  - Απλά σενάρια δοκιμών διότι δεν χρειάζονται δοκιμές για πολλές περιπτώσεις χρήσης
- Κατανοητές - Easy To Reason About
  - Δηλώνουν πρόθεση και σκοπό και δεν χρειάζονται τεκμηρίωση
- Αποφεύγεται η επανάληψη – DRY: Don't Repeat Yourself
  - Οι συναρτήσεις μπορούν να εύκολα να χρησιμοποιηθούν ως δομικά στοιχεία (lego blocks) και να επαναχρησιμοποιηθούν
- Τροποποιήσιμα - Easy To Refactor
  - Μικρά και ευέλικτα δομικά στοιχεία τα οποία μπορούν εύκολα να τροποποιηθούν, μετακινηθούν και διαγραφούν
- Παραλληλία
  - Επειδή το αποτέλεσμα των συναρτήσεων εξαρτάται μόνον από την είσοδο τους και δεν έχουν παρενέργειες (side effects) είναι εύκολη η παράλληλη εκτέλεση τους



# Γιατί πρέπει να τον γνωρίζετε

- Υπάρχουν πολλές περιπτώσεις στις οποίες η χρήση του κάνει απλούστερη την επίλυση προβλημάτων και μειώνει την πολυπλοκότητα
- Κάνει πιο κατανοητό τον κώδικα στην επίλυση σύνθετων προβλημάτων
- Χρησιμοποιείται ήδη από πολλές εφαρμογές Javascript οπότε χρειάζεται να τον γνωρίζετε για να κατανοήσετε πως λειτουργούν.
- Ήδη αρκετές άλλες γλώσσες όπως η C++ και η Java τα τελευταία χρόνια πρόσθεσαν βασικά στοιχεία συναρτησιακού προγραμματισμού

# Σύγκριση δηλωτικού με προστακτικό προγραμματισμό

Δηλωτικός (Declarative)

```
const w = [1, 2, 3, 4, 5, 6];  
const e = w.map(x => x + 1);  
console.log(e);
```

Προστακτικός (Imperative)

```
const w = [1, 2, 3, 4, 5, 6];  
const e = [];  
{  
  for (let item of w) {  
    e.push(item + 1);  
  }  
}  
console.log(e);
```

# Μέθοδοι πινάκων με συναρτησιακό προγραμματισμό

Οι πίνακες / λίστες Javascript διαθέτουν έτοιμες μεθόδους που κάνουν χρήση του συναρτησιακού προγραμματισμού όπως.


| Μέθοδος | Περιγραφή  |
|---------|--|
| forEach | Εκτελεί την συνάρτηση που δίνεται ως παράμετρος για το κάθε στοιχείο του πίνακα <a href="#">mdn</a>  |
| map     | Επιστρέφει ένα νέο πίνακα του οποίου τα στοιχεία υπολογίζονται με μία συνάρτηση η οποία δίνεται ως παράμετρος εισόδου <a href="#">mdn</a>                |
| filter  | Δημιουργεί ένα νέο πίνακα με τα στοιχεία τα οποία ικανοποιούν μία συνθήκη η υπολογίζεται από μία συνάρτηση που δίνεται ως παράμετρος <a href="#">mdn</a> |
| every   | Ελέγχει εάν όλα τα στοιχεία ενός πίνακα ικανοποιούν μια συνθήκη που δίνεται ως παράμετρος εισόδου <a href="#">mdn</a>                                    |
| some    | Ελέγχει εάν κάποιο στοιχείο ενός πίνακα ικανοποιεί μια συνθήκη που δίνεται ως παράμετρος εισόδου <a href="#">mdn</a>                                     |
| reduce  | Η μέθοδος reduce εκτελεί για κάθε στοιχείο μια συνάρτηση με την οποία υπολογίζει μια συγκεντρωτική τιμή (πχ άθροισμα) για τον πίνακα <a href="#">mdn</a> |

# Η μέθοδος forEach

Εκτελεί την συνάρτηση που δίνεται ως παράμετρος για το κάθε στοιχείο του πίνακα.

```
let anArray = [1,2,3,4,5]
//Example using the new method
anArray.forEach(console.log);
```

Υλοποίηση με συνάρτηση  
προστακτικού προγραμματισμού



```
//Implementation of forEach
//with a function
function forEach(arr, aFunction){
    for (let anItem of arr){
        aFunction(anItem);
    }
}

forEach (anArray, console.log);
```

[forEach method MDN](#)

# Υλοποίηση μεθόδου πίνακα forEachItem

- Το παρακάτω παράδειγμα υλοποιεί μια νέα μέθοδο η οποία για κάθε στοιχείο ενός πίνακα εκτελεί την συνάρτηση η οποία δίνεται ως είσοδος στην μέθοδο
- Η μέθοδος δηλώνεται στο prototype του Array οπότε ισχύει για όλα τα αντικείμενα τύπου Array


```
//Definition of a new for each method
//for all Array objects
Array.prototype.forEachItem = function(func) {
    for (let item of this){
        func(item);
    }
}
//Example using the new method
let m=[1,2,3];
m.forEachItem(console.log);
```

# Η μέθοδος filter

Επιστρέφει ένα νέο πίνακα με τα στοιχεία τα οποία ικανοποιούν μία συνθήκη η οποία υπολογίζεται από συνάρτηση που δίνεται ως παράμετρος εισόδου

```
let arr = [1, 2, 3, 4, 5];  
let evenArr = arr.filter(num => num % 2 === 0);  
console.log(`Τα άρτια στοιχεία είναι ${evenArr}`);
```

Υλοποίηση με συνάρτηση



```
// filter takes an array and function as argument  
function filter(arr, filterFunc) {  
  const filterArr = [];  
  // empty array  
  // loop through array  
  for (let i = 0; i < arr.length; i++) {  
    const result = filterFunc(arr[i], i, arr);  
    // push the current element if result is true  
    if (result) filterArr.push(arr[i]);  
  }  
  return filterArr;  
}  
let oddArr2 = filter(arr, num => num % 2 === 0);  
console.log(`Τα άρτια στοιχεία είναι ${oddArr2}`);
```


[Filter method MDN](#)

# Η μέθοδος map

Επιστρέφει ένα νέο πίνακα με τα στοιχεία τα οποία υπολογίζονται από μία συνάρτηση που δίνεται ως παράμετρος εισόδου

```
let arr = [1, 2, 3, 4, 5];  
let newArr = arr.map(num => num*num);  
console.log(newArr);
```

Υλοποίηση με συνάρτηση



```
function map(arr, mapFunc) {  
  const mappedArr = [];  
  for (let anItem of arr) {  
    mappedArr.push(mapFunc(anItem));  
  }  
  return mappedArr;  
}  
  
console.log(map(arr, (num) => num*num));
```

# Η μέθοδος reduce

Η μέθοδος reduce εκτελεί για κάθε στοιχείο μια συνάρτηση με την οποία υπολογίζει μια συγκεντρωτική τιμή (πχ άθροισμα) για τον πίνακα.

```
let m = [1, 2, 3, 4];  
const sum = m.reduce((accumulator, currentValue) =>  
    accumulator + currentValue);  
console.log(`Το άθροισμα είναι ${sum}`);
```

Υλοποίηση με συνάρτηση



```
// reducer takes an array, reducer() and  
// initialValue as argument  
function reduce(arr, reducer, initialValue) {  
    let accumulator =  
        initialValue === undefined ? 0 : initialValue;  
    for (let i = 0; i < arr.length; i++) {  
        accumulator = reducer(accumulator, arr[i], i, arr);  
    }  
    return accumulator;  
}  
  
const sum1 = reduce(m,  
    (accumulator, currentValue) => accumulator + currentValue);  
console.log(`Το άθροισμα είναι ${sum1}`);
```

[reduce method MDN](#)



# Παράδειγμα reduce

- Επιστροφή πίνακα χωρίς διπλότυπα

```
const myArray = ["a", "b", "a", "b", "c"];
const myArrayWithNoDuplicates = myArray.reduce(
  (accumulator, currentValue) => {
    if (!accumulator.includes(currentValue)) {
      return [...accumulator, currentValue];
    }
    return accumulator;
  },
  [],
);
console.log(myArrayWithNoDuplicates);
```

```
(3) ['a', 'b', 'c']
```

# Αναφορές

- <https://developer.mozilla.org/en-US/docs/Learn/JavaScript>
- <https://www.w3schools.com/js/default.asp>
- [https://www.w3schools.com/js/js\\_examples.asp](https://www.w3schools.com/js/js_examples.asp)
- <https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/#basic-javascript>

# Αναφορές συναρτησιακού προγραμματισμού

- <https://www.freecodecamp.org/news/how-to-write-your-own-map-filter-and-reduce-functions-in-javascript-ab1e35679d26/>
- <https://javascript.plainenglish.io/how-implement-the-filter-method-in-javascript-from-scratch-da4ae2ca4321>
- <https://dspace.lib.uom.gr/bitstream/2159/24541/2/SafaridisFelixMsc2019present.pdf>
- [https://el.wikipedia.org/wiki/%CE%A3%CF%85%CE%BD%CE%B1%CF%81%CF%84%CE%B7%CF%83%CE%B9%CE%B1%CE%BA%CF%8C%CF%82\\_%CF%80%CF%81%CE%BF%CE%B3%CF%81%CE%B1%CE%BC%CE%BC%CE%B1%CF%84%CE%B9%CF%83%CE%BC%CF%8C%CF%82](https://el.wikipedia.org/wiki/%CE%A3%CF%85%CE%BD%CE%B1%CF%81%CF%84%CE%B7%CF%83%CE%B9%CE%B1%CE%BA%CF%8C%CF%82_%CF%80%CF%81%CE%BF%CE%B3%CF%81%CE%B1%CE%BC%CE%BC%CE%B1%CF%84%CE%B9%CF%83%CE%BC%CF%8C%CF%82)
- <https://github.com/rjhilgefort/functional-javascript-presentation>
- <https://mostly-adequate.gitbook.io/mostly-adequate-guide/>
- <https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0>