ECE_ΓΚ802 ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΔΙΑΔΙΚΤΥΟΥ
8ο εξάμηνο

ενότητα 4

JavaScript – μέρος 3

Ασύγχρονη JavaScript – διαχείριση συμβάντων

Η ενότητα αυτή περιγράφει μηχανισμούς ασύγχρονης εκτέλεσης κώδικα στην JavaScript, που είναι απαίτηση που συναντάμε συχνά στον φυλλομετρητή (διαχείριση συμβάντων και ασύγχρονη πρόσβαση σε δεδομένα) αλλά και στο περιβάλλον του node.js

Ασύγχρονη εκτέλεση κώδικα Javascript

- Στην **ακολουθιακή (σύγχρονη)** εκτέλεση ενός προγράμματος, η μία εντολή εκτελείται μετά την άλλη.
- Στην **ασύγχρονη εκτέλεση,** ένα τμήμα του προγράμματος, που πρέπει να περιμένει, τοποθετείται σε μια άλλη ουρά εκτέλεσης, χωρίς να μπλοκάρει τη ροή εκτέλεσης.
- Η JS έχει δύο μηχανισμούς ασύγχρονης εκτέλεσης:
  - **Την κλήση συνάρτησης επιστροφής** (callback function),
  - **Τον μηχανισμό Promise** (έχει επιπλέον υλοποιηθεί ως **async** / **await**)

Περιπτώσεις ασύγχρονης εκτέλεσης κώδικα

- Κλήση των μεθόδων του αντικειμένου window: setTimeout(f, t), setInterval(f, t)
- Μέσω χειριστών συμβάντων,π.χ. button.addEventListener(event, handler)
- Χρήση της **fetch** για ανάκτηση δεδομένων από server (Promise).
- Κλήση του **requestAnimationFrame** για επαναληπτική εκτέλεση κώδικα σε κίνηση γραφικών

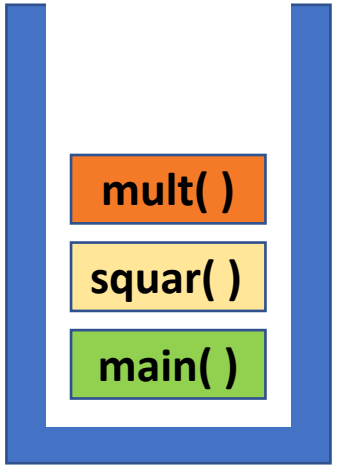
Η JavaScript είναι μονο-νηματική (single-threaded) γλώσσα

Παράδειγμα εκτέλεσης κώδικα χωρίς ασύγχρονη λειτουργία

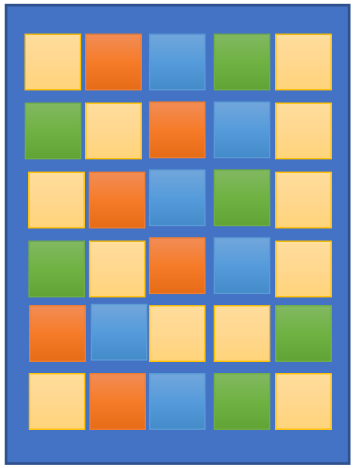
```
function mult(a,b) {
      return ab};
function squar(n) {
      return mult(n,n)};
console.log(squar(5));
```

```
> 25
```

Call Stack
(στοίβα)



Heap
(σωρός)



visualize: http://pythontutor.com/javascript.html#mode=display

Ασύγχρονη λειτουργία: βρόγχος ελέγχου συμβάντων

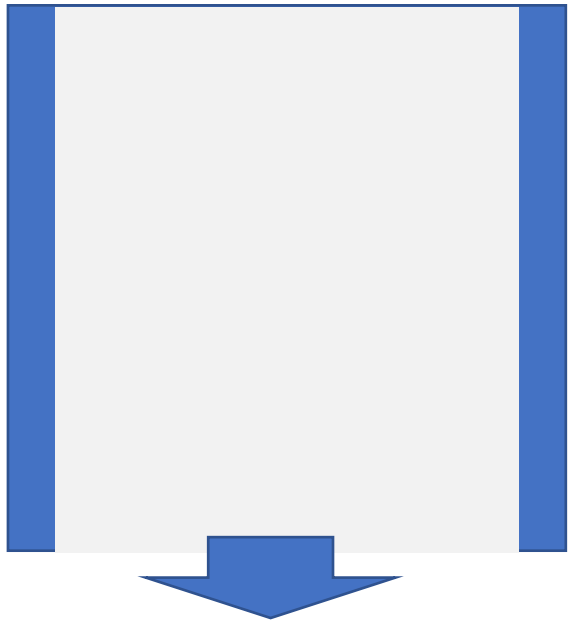
Event loop: Μια διεργασία που τρέχει συνεχώς και εξετάζει ποια είναι η επόμενη εργασία για εκτέλεση. Εξετάζει τη στοίβα αλλά και ουρές.

Event table

event	callback
click <button>	handler()
μετά από 500ms	call()

Callback Queue

(ουρά κλήσεων συναρτήσεων επιστροφής)



παράδειγμα ασύγχρονης εκτέλεσης κώδικα

```
const f2 = () => console.log("f2");
const f3 = () => console.log("f3");
const f1 = () => {
   console.log("f1");
   setTimeout(f2, 0);
   f3();};
f1();
```

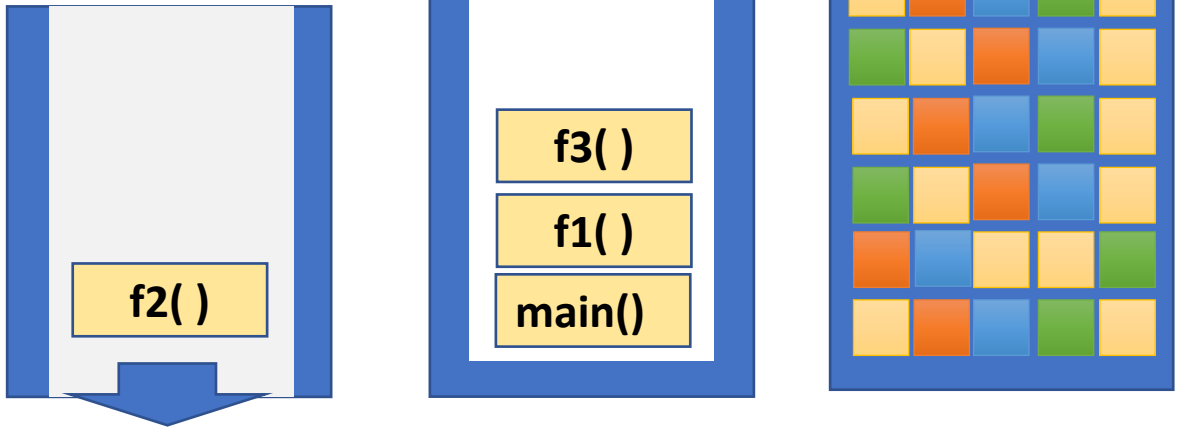
Όταν καλείται η setTimeout (), η JavaScript ξεκινάει ένα χρονόμετρο. Μόλις λήξει ο χρόνος, σε αυτήν την περίπτωση αμέσως, η συνάρτηση επιστροφής τοποθετείται στην ουρά μηνυμάτων **callback queue**.
Ο βρόχος συμβάνων δίνει προτεραιότητα στη **call stack**, όταν δεν υπάρχει τίποτα στη call stack, μετακινείται για να παραλάβει στοιχεία από την ουρά συμβάντων **callback queue**.

Παράδειγμα – κλήση συνάρτησης επιστροφής

Callback Queue
(ουρά κλήσεων συναρτήσεων επιστροφής)

Call Stack
(στοίβα)

Heap
(σωρός)




```
function f2() {console.log("f2") };
function f3() {console.log("f3") };
function f1() {
    console.log("f1");
    setTimeout(f2, 0);
    f3(); };
f1();
```

```
f1
f3
f2
```

Ο Βρόχος ελέγχου συμβάντων (event loop) επιλέγει εργασίες από την ουρά κλήσεων συναρτήσεων επιστροφής, μόνο όταν η στοίβα είναι κενή (δεν υπάρχουν εργασίες για εκτέλεση).

Κλήση συναρτήσεων επιστροφής callback

Ορισμός συνάρτησης επιστροφής ενός χειριστή συμβάντων

event-handler για συμβάν click

```
document.querySelector('button').addEventListener('click', f));
function f() { console.log('button clicked')}
```

ορισμός συνάρτησης

```
document.querySelector('button').addEventListener('click',
        () => { console.log('button clicked')});
```

ανώνυμη συνάρτηση =>

```
document. querySelector('button').addEventListener('click',
        function() { console.log('button clicked') });
```

ανώνυμη συνάρτηση function

setTimeout(συνάρτηση επιστροφής, delay);

εκτέλεσης συνάρτησης με καθυστέρηση

```
console.log('before');

setTimeout( () => {
     console.log('με καθυστέρηση 2sec');
     }, 2000)

console.log('after');
```

Προσοχή: όπως είδαμε η ασύγχρονη συνάρτηση θα μπει στο event table, όταν περάσουν τα 2000ms θα μεταβεί στην ουρά και θα εκτελεστεί όταν αδειάσει η στοίβα, άρα η καθυστέρησή εκτέλεσής της θα είναι > 2000ms

before

after

to be delayed by 2 sec

Άσκηση

```
console.log('αρχή');
setTimeout( () => {console.log('timeout1')}, 2000)
setTimeout( () => {console.log('timeout2')}, 1000)
console.log('τέλος');
```

Ποιο το αποτέλεσμα;

A
αρχή
timeout1
timeout2
τέλος

B
αρχή
τέλος
timeout1
timeout2

Γ
αρχή
τέλος
timeout2
timeout1

Δ
αρχή
timeout2
timeout1
τέλος

```
id = setIterval(func, interval);
```

εκτέλεση μιας συνάρτησης επαναληπτικά με κάποιο interval

```
clearInterval(id);
```

Κατάργηση εκτέλεσης

```
const i = setInterval(f, 500); // η συνάρτηση f καλείται κάθε 0.5"
let counter = 0; // μετρητής επαναλήψεων
function f(){
     if(Math.random() < 0.3){
          clearInterval(i); // τερματισμός επανάληψης
          console.log('τέλος επανάληψης...')
     } else console.log("επανάληψη", ++counter);
}
```

Άσκηση

```
let counter = 0;
const id = setInterval(() => {
     console.log(++counter, "hi ");
}, 50);

setTimeout(clearInterval, 200, id);
```

Ποιο το αποτέλεσμα;

Α

Β

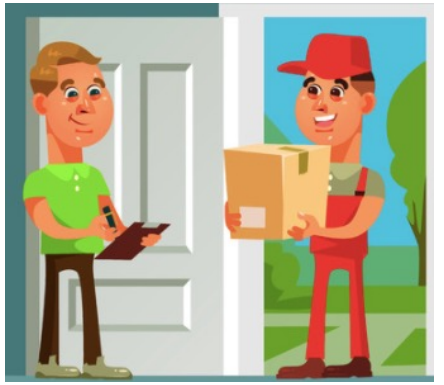
Γ

Δ

Pelatis - Courrier

```
class Pelatis {
//Ο πελάτης που περιμένει και μετράει τα δευτερόλεπτα
    constructor(delState, t) {
    this.delivered = delState;
    this.timer = t;
    this.setting = setInterval(()=>this.checkDelivery(), 1000);
    }
    checkDelivery = function () {
        if (this.delivered == 'έφτασε') {
                clearInterval(this.setting);
                display.innerHTML += 'ΠΕΛΑΤΗΣ: Επιτέλους... έκαναν ...' +
                        this.timer + 'sec.<br>';
                return;
        }
        display.innerHTML += 'ΠΕΛΑΤΗΣ: περιμένω...' + ++this.timer + "<br>";
     }
}
```

Pelatis - Courrier



```
class Courrier{
// Ο κούριερ που παραδίδει μετά από delay msec, στον pelatis
   constructor(pelatis, delay){
      this.pelatis = pelatis;
      this.delay = delay;
      this.setting = setTimeout(() => {
          this.pelatis.delivered = 'έφτασε';
         display.innerHTML += 'ΚΟΥΡΙΕΡ: εγώ το παρέδωσα...<br>';},
       this.delay);
       }
}


p = new Pelatis('όχι', 0);
new Courrier(p, 5000);
```

```
ΠΕΛΑΤΗΣ: περιμένω...1
ΠΕΛΑΤΗΣ: περιμένω...2
ΠΕΛΑΤΗΣ: περιμένω...3
ΠΕΛΑΤΗΣ: περιμένω...4
ΠΕΛΑΤΗΣ: περιμένω...5
ΚΟΥΡΙΕΡ: εγώ το παρέδωσα...
ΠΕΛΑΤΗΣ: Επιτέλους... έκαναν ...5sec.
```

Ο μηχανισμός Promise

Ο μηχανισμός Promise (ES6)

Μια υπόσχεση **Promise** είναι ένα αντικείμενο που εκπροσωπεί το αποτέλεσμα μιας ασύγχρονης εργασίας.

```
let p = new Promise(executor(resolve, reject));
```

executor είναι μια συνάρτηση που εκτελεί την ασύγχρονη εργασία και **resolve** και **reject** είναι δύο συναρτήσεις επιστροφής (callback) οι οποίες θα κληθούν αν η υπόσχεση ικανοποιηθεί ή όχι, αντίστοιχα.

Ο μηχανισμός Promise εισήχθη στην JS στην έκδοση ES6 για να λύσει το πρόβλημα διαδοχικών callbacks και για καλύτερη διαχείριση σφαλμάτων σε ασύγχρονη εκτέλεση κώδικα.

Οι καταναλωτές της Promise

Μπορούν να οριστούν ένας ή περισσότεροι καταναλωτές μιας υπόσχεσης, αυτοί υλοποιούνται με τις μεθόδους then(), catch(), finally() του αντικειμένου Promise.

```
p = new Promise(f)
p.then(call-back-success, call-back-failure);
```

Promises χρησιμοποιούνται από την JS και στον εξυπηρετητή και στον φυλλομετρητή. Παράδειγμα το **fetch** API και το API **Service Workers**, ενώ ο μηχανισμός async/await βασίζεται στις promises

παράδειγμα

```
const p = new Promise(promisedFunc);

function promisedFunc (resolve,reject){//συναρτήσεις που ενεργοποιούνται σε περίπτωση επιτυχούς ή όχι ολοκλήρωσης
    if (Math.random()>0.5){ //προσομοίωση τυχαιότητας λειτουργίας
        resolve('ok')}
    else {
        reject('error')}
}
// χρήση της Promise
function callBackSuccess(result){ //συνάρτηση resolve
        console.log(result)
}
function callBackFailure(err){ //συνάρτηση reject
        console.log(err)
}

p.then(callBackSuccess, callBackFailure) //κλήση Promise
```

παράδειγμα –σύνταξη με ανώνυμες συναρτήσεις

```
let done = true
const p = new Promise( (resolve, reject) => {
        if (done) {
            resolve('ok');
        } else {
            reject('error');
        }
    })
```

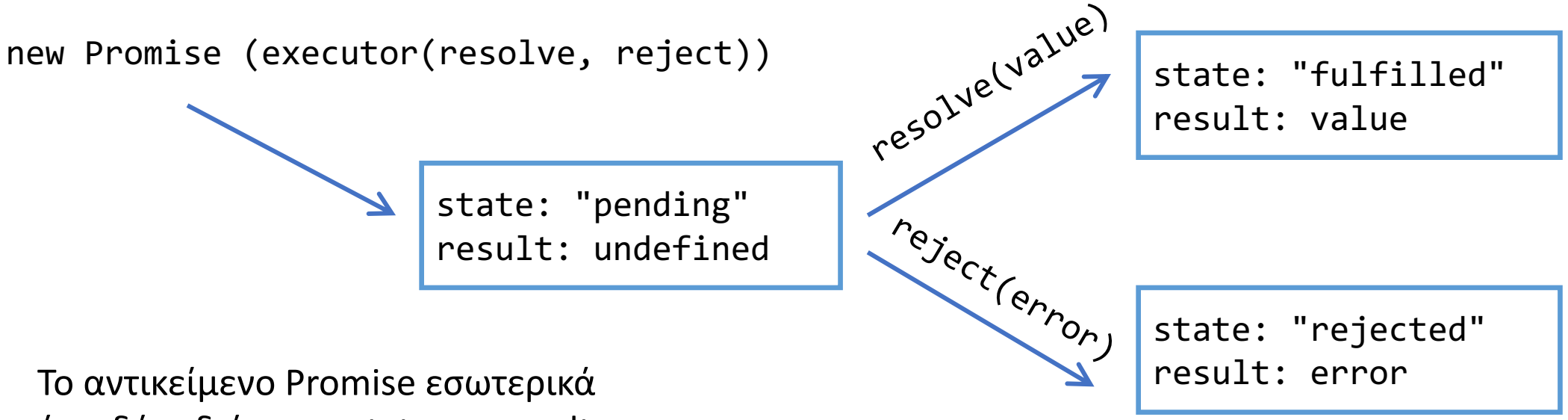
```
p.then( (result)=> {
                console.log(result); }, // "ok!"
            (err)=> {
                console.log(err);});  // error
```

Η new Promise(executor(resolve, reject)) επιστρέφει ένα αντικείμενο Promise που έχει εσωτερικά τις ιδιότητες **state** και **result**

Η συνάρτηση executor που περνάμε στον δημιουργό της Promise καλείται αυτόματα από τον δημιουργό του αντικειμένου Promise. Ο executor **εκτελεί την ασύγχρονη εργασία** και όταν τελειώσει, αν είναι επιτυχής, καλεί τη συνάρτηση επιστροφής resolve που έχει ορίσει η JavaScript, και στην οποία περνάμε τα αποτελέσματα, ενώ η συνάρτηση επιστροφής reject καλείται σε περίπτωση αποτυχίας εκτέλεσης της εργασίας.

Όταν αλλάξει κατάσταση του αντικείμενου Promise η resolve() αν ξανακαλεστεί δεν αλλάζει πλέον την κατάσταση της Promise

https://javascript.info/promise-basics



Το αντικείμενο Promise εσωτερικά
έχει δύο ιδιότητες, state και result
Το state παίρνει τις τιμές:

- "pending"
- "fullfilled"
- "rejected"

καταναλωτές μιας υπόσχεσης Promise

Ο καταναλωτής μια υπόσχεσης Promise είναι η μέθοδος p.**then()** στην οποία περνάμε συναρτήσεις χειρισμού των αποτελεσμάτων

```
promise.then(
     function(result) {
          / χειρισμός επιτυχούς αποτελέσματος /
     },
     function(error) {
          / χειρισμός σφάλματος /
     } );
```

αν ενδιαφερόμαστε μόνο για την επιτυχή ολοκλήρωση μόνο, περνάμε μόνο την πρώτη συνάρτηση στο then()

καταναλωτές μιας υπόσχεσης Promise

Ο καταναλωτής σφάλματος μια υπόσχεσης Promise είναι η μέθοδος p.**catch(handler)** ή εναλλακτικά **.then(null, handler)**

```
.then(null, errorHandlingFunction).

.catch(errorHandlingFunction)

.finally(callback) //εκτελείται πάντα
```

Άσκηση: ποιο το αποτέλεσμα;

Όταν στο σώμα της Promise η resolve() καλείται την πρώτη φορά αλλάζει η κατάσταση του αντικειμένου, η οποία δεν μπορεί να ξανα-αλλάξει με επόμενες κλήσεις της resolve()

```
let promise = new Promise((resolve, reject) =>  {
   setTimeout(() => resolve(10), 2000);
   setTimeout(() => resolve(20), 1000);
   });

promise.then((result) => {
   console.log(result);
   });
```

Άσκηση

Υλοποίηση της setTimeout(f, delay) με Promise

Να ορίσετε συνάρτηση delay(ms) που επιστρέφει ένα αντικείμενο Promise, στο οποίο μπορούμε να ορίσουμε μια συνάρτηση ϕ καταναλωτή της Promise ώστε:

```
delay(ms).then(ϕ); ←→ setTimeout(ϕ, ms);
```

Άσκηση - λύση

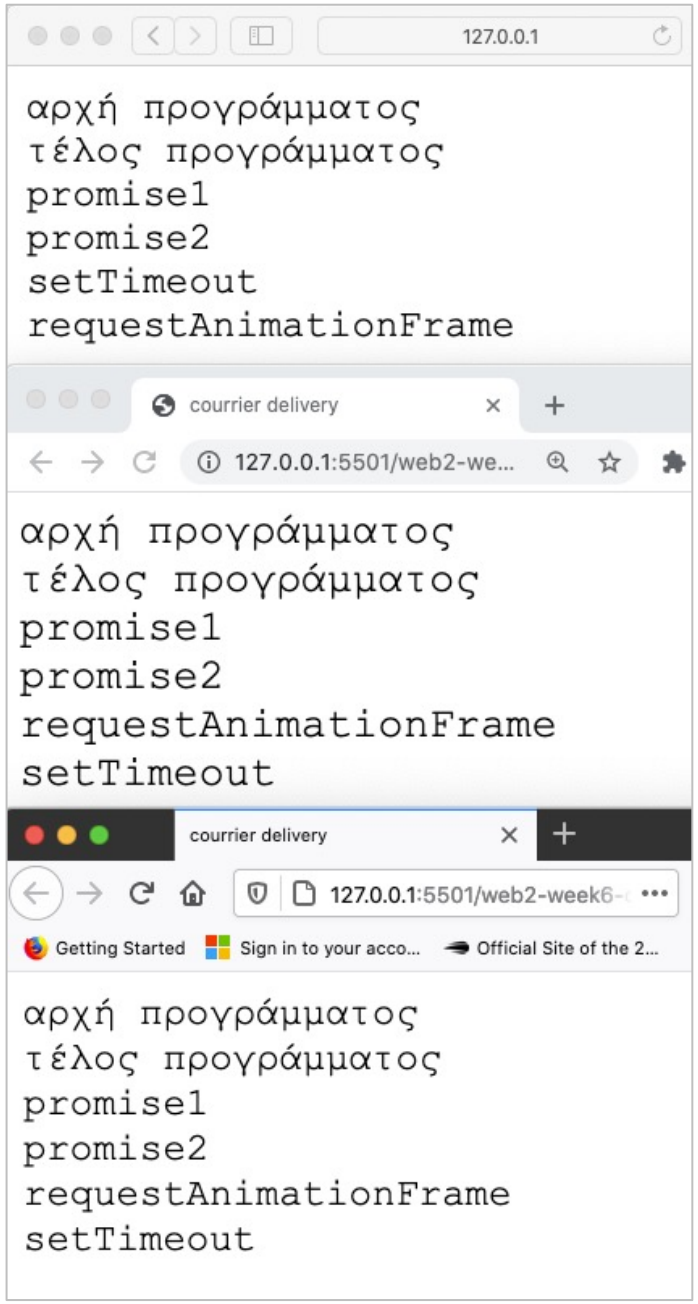
```
let delay = (ms) => new Promise((r) =>
setTimeout(r, ms));

delay(3000).then(() => console.log("runs
after 3 sec"));
delay(2000).then(() => console.log("runs
after 2 sec"));
```

Promises στον βρόχο ελέγχου συμβάντων

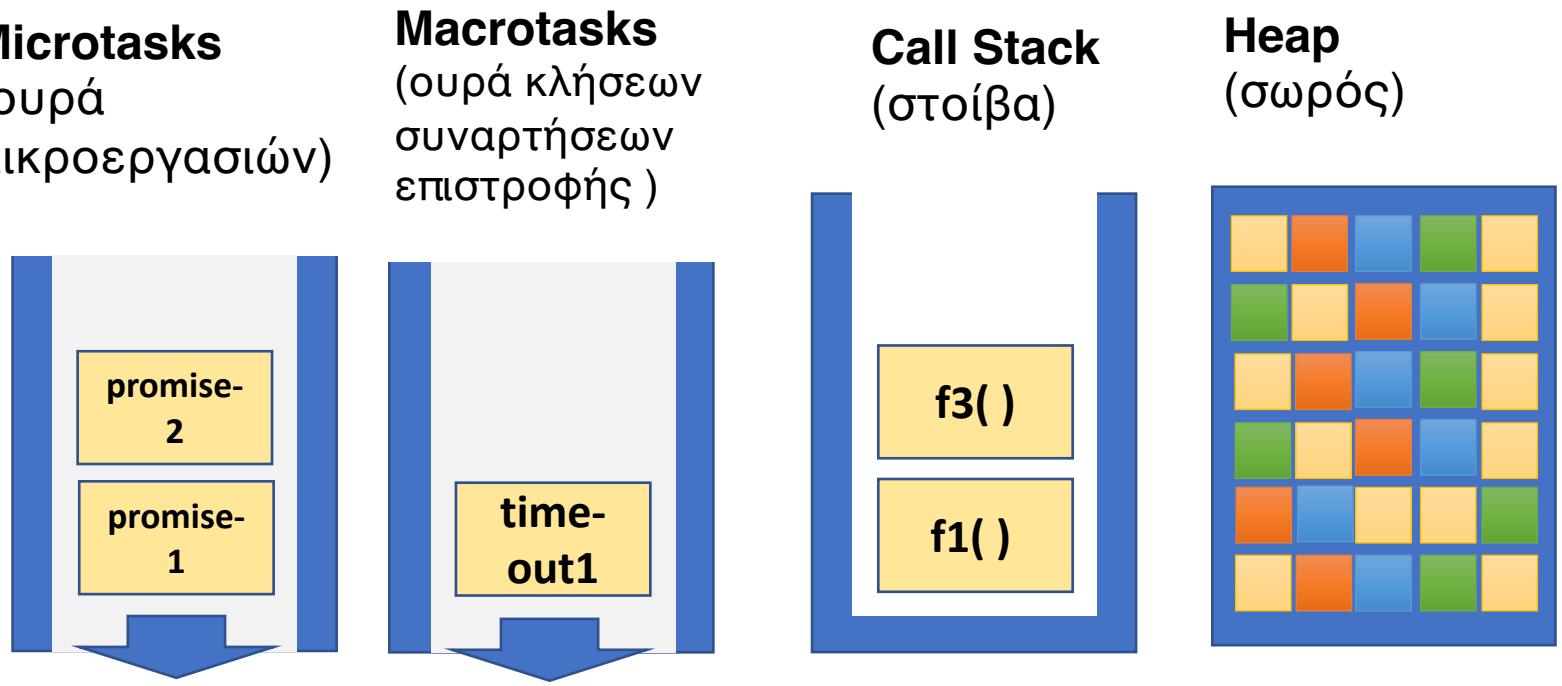
micro-macro task queue
σειρά εκτέλεσης εργασιών

```
let display = document.querySelector('div');
//αρχή προγράμματος
display.innerHTML += 'αρχή προγράμματος<br>';
// setTimout(, 0)
setTimeout(() => display.innerHTML += 'setTimeout<br>', 0);
// requestAnimationFrmae
requestAnimationFrame(() =>
      display.innerHTML +='requestAnimationFrame<br>');
// Promise object with two .then objects
Promise.resolve()
      .then(() => display.innerHTML +='promise1<br>')
      .then(() => display.innerHTML +='promise2<br>');
// end of script
display.innerHTML +='τέλος προγράμματος<br>';
```



πρόγραμμα με Promises – ουρά μικροεργασιών

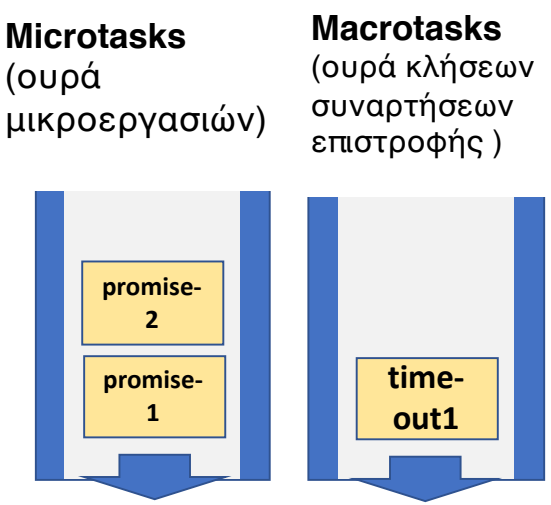
Η ουρά μικροεργασιών (microtask queue) εξαντλείται σε κάθε κύκλο, εκεί μπαίνουν promises

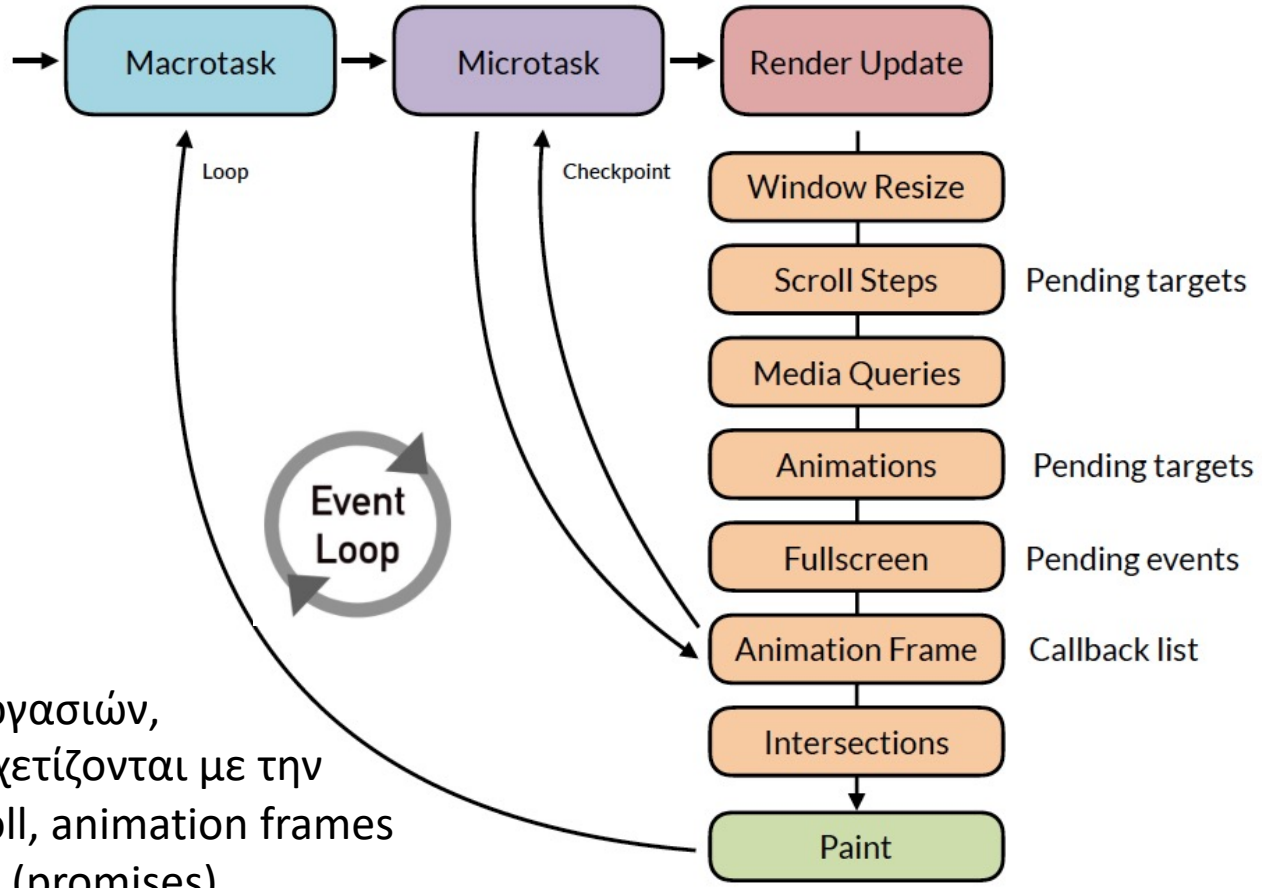



```
console.log('αρχή προγράμματος');
setTimeout(() => console.log('πρώτο timeout'), 0);
requestAnimationFrame(() => console.log('πρώτο animationFrame'));
Promise.resolve()
       .then(() => console.log('promise1'))
       .then(() => console.log('promise2'));
console.log('τέλος προγράμματος');
```

```
αρχή προγράμματος
τέλος προγράμματος
promise1
promise2
πρώτο animationFrame
πρώτο timeout
```

βρόχος ασύγχρονης εκτέλεσης εργασιών





σε κάθε tick του βρόχου εκτέλεσης εργασιών, εκτελούνται διάφορα βήματα, που σχετίζονται με την απεικόνιση πληροφορίας (resize, scroll, animation frames κλπ), ενώ εξαντλούνται τα microtasks (promises) [διαφορετική υλοποίηση του βρόχου σε διάφορες περιπτώσεις]

https://www2.slideshare.net/derekstavis/javascript-event-loop-104368853

Ουρά Promise (microtasks)

```
//εκπλήρωση υπόσχεσης

const promise = Promise.resolve();

//κατανάλωση υπόσχεσης

promise.then(() =>
     {console.log("η υπόσχεση οκ")});

console.log("τέλος κώδικα");
```

αποτέλεσμα:

```
"τέλος κώδικα"
"η υπόσχεση οκ"
```

Αλυσίδες καταναλωτών υποσχέσεων

αλυσίδες καταναλωτών υποσχέσεων

```
Promise.then(φ)
```

- η μέθοδος then() όταν εφαρμόζεται σε αντικείμενο Promise επιστρέφει Promise
- συνεπώς, μπορούμε να δημιουργήσουμε αλυσίδα από καταναλωτές

παράδειγμα αλυσίδας καταναλωτών

```
αρχή προγράμματος
τέλος προγράμματος
καταναλωτής-1: 1
καταναλωτής-2: 2
καταναλωτής-3: 4
```

```
let display = document.querySelector('div');
//αρχή προγράμματος
display.innerHTML += 'αρχή προγράμματος<br>';
new Promise(function(resolve, reject) {
      setTimeout(() => resolve(1), 1000); // η υπόσχεση υλοποιείται σε 1"
}).then(function(result) { // καλείται ο 1ος καταναλωτής then()
      display.innerHTML += "καταναλωτής-1: "+result+"<br>";
      return result  2; // επιστρέφει την τιμή 2
}) .then(function(result) { // καλείται ο 2ος καταναλωτής
      display.innerHTML += "καταναλωτής-2: "+result+"<br>";
      return result  2; // επιστρέφει την τιμή 4
}).then(function(result) { // καλείται ο 3ος καταναλωτής
      display.innerHTML += "καταναλωτής-3: "+result+"<br>";
      return result  2;
});
display.innerHTML += 'τέλος προγράμματος<br>';
```

άσκηση – ποιο το αποτέλεσμα;

```
// 2ο παράδειγμα
display.innerHTML += 'αρχή <br>';
let promise = new Promise(function(resolve, reject) {
        setTimeout(() => resolve(1), 1000);
});

promise.then(function(result) {
        display.innerHTML += "then-1: "+result+"<br>";
        return result * 2;
});

promise.then(function(result) {
        display.innerHTML += "then-2: "+result+"<br>";
        return result * 2;
});

promise.then(function(result) {
        display.innerHTML += "then-3: "+result+"<br>";
        return result * 2;
});

display.innerHTML += 'τέλος <br>';
```

απάντηση

Α

αρχή
then-1 1
then-2 2
then-3 4
τέλος

Β

αρχή
τέλος
then-1 1
then-2 1
then-3 1

Γ

αρχή
τέλος
then-1 1
then-2 2
then-3 4

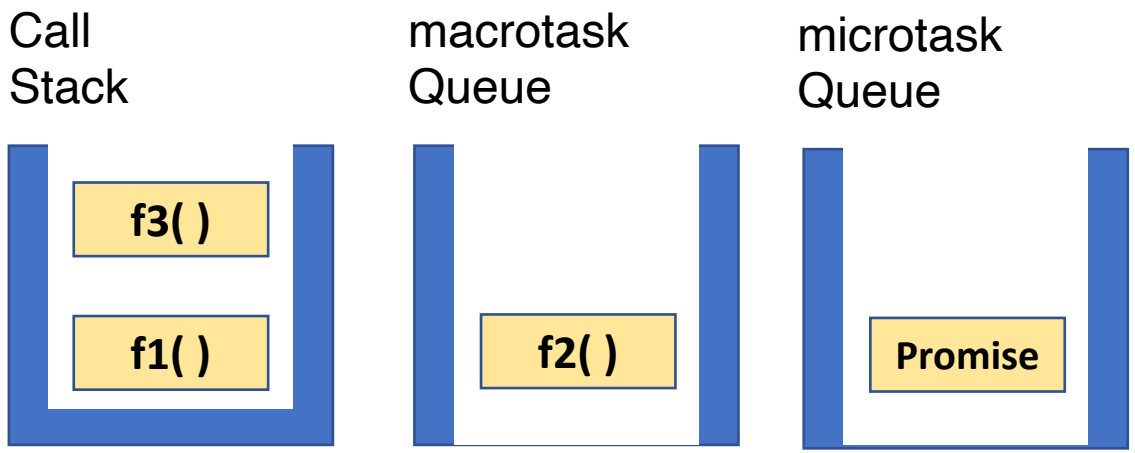
Η ουρά promise (promise jobs queue)

```
const f2 = () => console.log("f2");
const f3 = () => console.log("f3");

const f1 = () => {
    console.log("f1");
    setTimeout(f2, 0);
    new Promise((resolve, reject) =>
    resolve("promised")
    ).then(resolve => console.log(resolve));
    f3();
};
f1();
```

```
>

f1
f3
promised
f2
```



Ο μηχανισμός async / await

Η λέξη-κλειδί async πριν από μια συνάρτηση σημαίνει ότι η συνάρτηση επιστρέφει μια υπόσχεση promise

```
async function f() {
return 1;}

f().then((result) => {console.log(result)}
);
```

Αυτό είναι ισοδύναμο με:

```
function f() {
        return Promise.resolve(1);
}
```

await σε συνάρτηση async

Η λέξη κλειδί **await** χρησιμοποιείται μόνο μέσα σε μια συνάρτηση **async**.

Κάνει τη JavaScript να περιμένει έως ότου διευθετηθεί η υπόσχεση και μετά συνεχίζει με το αποτέλεσμα. Είναι η ίδια λειτουργία με το μηχανισμό promise με καλύτερη σύνταξη από το .then().

```
async function f() {
        let promise = new Promise((resolve, reject) => {
                setTimeout(() => resolve("done!"), 1000);
        });
        let result = await promise;
        console.log(result); // "done!"
}
f();
```

Η διεπαφή fetch

Η διεπαφή **fetch** – ασύγχρονη κλήση στον φυλλομετρητή

- Από την εποχή του Internet Explorer5 το 1998, κάνουμε ασύγχρονες κλήσεις στο δίκτυο μέσα από τον φυλλομετρητή με χρήση κλήσεων του XMLHttpRequest (AJAX)
- Η jQuery προσέφερε πιο απλή σύνταξη jQuery.ajax(), jQuery.get(),
- Η διεπαφή **fetch(url)** έχει ενσωματωθεί στο αντικείμενο global window : `fetch("/file.json")`
- Η `fetch(url)` επιστρέφει Promise, άρα το αποτέλεσμα το χειριζόμαστε με αλυσίδα από then()

https://www.youtube.com/watch?v=tc8DU14qX6I

ιδιότητες του αντικειμένου που επιστρέφει η fetch

- **headers** Μέσω της ιδιότητας αυτής έχουμε πρόσβαση στην κεφαλίδα του HTTP response

```
fetch('./file.json').then(response => { console.log(response.headers.get('Content-Type'))
console.log(response.headers.get('Date')) })
```

- **status** Ακέραιος αριθμός που ορίζει το HTTP response status. 101, 204, 205, ή 304 σημαίνουν null body status, 200 μέχρι 299, είναι οκ, (success), 301, 302, 303, 307, ή 308 redirect

```
fetch('./file.json').then(response => console.log(response.status))
```

- **statusText** Συμβολοσειρά με το status message της απόκρισης. "OK" αν επιτυχής απόκριση.

```
fetch('./file.json').then(response => console.log(response.statusText))
```

- **url** Αναπαριστά τη διεύθυνση URL του αντικείμενου.

```
fetch('./file.json').then(response => console.log(response.url))
```

- **Body content** Η απόκριση έχει ένα body, στο οποίο έχουμε πρόσβαση μέσω διαφόρων μεθόδων: Η μέθοδος .**text**() επιστρέφει το body ως string , η **. json()** το επιστρέφει ως JSON-parsed object, **.blob()** ως Blob object, **formData**() ως FormData object, κλπ.

Παράδειγμα fetch

```
function loadJson(url) {
    return fetch(url)
        .then(response => {
                if (response.status == 200) {
                        return response.json();
                } else {
                        throw new Error(response.status);
                }
        })
}

loadJson(url)
.catch(alert); // Error: 404
```

Η fetch επιστρέφει αντικείμενο promise

Σύνδεση με διεπαφή REST με fetch

- Η fetch μπορεί να έχει πολλές χρήσεις, μεταξύ των άλλων να χρησιμοποιηθεί για να ανακτήσει η ιστοσελίδα πληροφορίες από ένα σημείο επαφής (endpoint) μιας διεπαφής REST(Representation State Transfer) συνήθως μέσω αιτήματος GET.
- Υπάρχουν πολλοί φορείς που διαθέτουν δημόσια δεδομένα και πληροφορίες, κάποιοι χωρίς να χρειάζεται εγγραφή. Βλέπε **open data initiative**.
- Παραδείγματα:
  - https://restcountries.com/ (στοιχεία για χώρες)
  - http://www.7timer.info/doc.php?lang=en#api (πρόβλεψη καιρού)
  - https://www.exchangerate-api.com/ (τιμές συναλλάγματος)
  - https://covid19api.com/ (υγειονομική κρίση)

https://github.com/public-apis/public-apis

παράδειγμα fetch με await

```
async function loadJson(url) { // change to async
	let response = await fetch(url); // await instead of promise
	if (response.status == 200) {
		let json = await response.json(); // await for json if ok
		return json;
	}
	throw new Error(response.status);
}

loadJson('no-such-user.json')
.catch(alert); // Error: 404 --- no await since outside async
```

axios module υλοποιήσης ασύγχρονων κλήσεων στον φυλλομετρητή

npm install axios

στον browser

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

Παράδειγμα: οι καλοί γείτονες

Νίκος Αβούρης
Πανεπιστήμιο Πατρών

Άσκηση: Οι καλοί γείτονες με ποιες χώρες συνορεύει... (πχ. η Ιταλία);

- Χρησιμοποιήστε την διεπαφή https://restcountries.com/
- Ανακτήστε τα στοιχεία της χώρας (Italy)
- Μελετήστε τη δομή των δεδομένων JSON που επιστρέφει η διεπαφή.
- Αναζητήσετε την πληροφορία σχετικά με τις χώρες που συνορεύει, (Array: borders)
- Για κάθε μια από τις χώρες αυτές βρείτε τα στοιχεία της με βάση τον κώδικά της, από αυτά διαλέξτε το όνομά της.
- Γενικεύσετε τη λύση.

Promise.all(set-of-promises)

- Η μέθοδος `all` του αντικείμενου Promise δέχεται ως όρισμα ένα σύνολο από υποσχέσεις και επιστρέφει μια υπόσχεση η οποία εκπληρώνεται όταν όλες οι υποσχέσεις έχουν εκπληρωθεί.
- Η Promise.all() είναι χρήσιμη για περιπτώσεις που έχουμε ένα σύνολο υποσχέσεων που θα θέλαμε όλες να εκπληρωθούν πριν προχωρήσουμε στην παρουσίαση των αποτελεσμάτων.

άλλες μέθοδοι του αντικειμένου Promise

- **Promise.all()** σταματάει στην πρώτη άρνηση υπόσχεσης και επιστρέφει error, ενώ επιστρέφει υπόσχεση με τα αποτελέσματα αν όλες εκπληρωθούν.
- **Promise.race()** σταματάει στην πρώτη που ολοκληρώνεται είτε ως εκπληρωμένη υπόσχεση, είτε σφάλμα και την επιστρέφει.
- Promise.any() σταματάει στην πρώτη που επιστρέφει εκπληρωμένη υπόσχεση και την επιστρέφει
- Promise.allSettled() επιστρέφει όλες τις υποσχέσεις, {status: 'fulfilled', value: result} – για όσες έχουν εκπληρωθεί και {status: 'rejected', reason: error} για όσες όχι.

λύση:

για κάθε γείτονα μαθαίνουμε τον κωδικό της χώρας (array borders =
['AUT', 'FRA', 'SMR', 'SVN', 'CHE', 'VAT']

και για καθένα από τα μέλη του, επιθυμούμε να βρούμε το όνομα της αντίστοιχης χώρας

```
Promise.all(theCountries)
.then ((results)=> render(results))
```

https://codeburst.io/javascript-making-asynchronous-calls-inside-a-loop-and-pause-block-loop-execution-1cb713fbdf2d

Οι καλοί γείτονες - λύση

(βλέπε: https://restcountries.eu/)

```
const urlCountry = 'https://restcountries.eu/rest/v2/name/';
const urlCode = 'https://restcountries.eu/rest/v2/alpha/';

function loadCountryNameFromCode(code) {
    // επιστρέφει Promise του αποτελέσματος – του ονόματος της χώρας
    return new Promise((resolve, reject) => {
        fetch(urlCode + code)
        .then((resp) => {
                if (resp.status == 200) {
                    return resp.json();
                } else reject(new Error(response.status));
        })
        .then((data) => {
                resolve(data.name.common); // όνομα από κωδικό "Greece" από "GR"
        });
    });
}
```

https://codeburst.io/javascript-making-asynchronous-calls-inside-a-loop-and-pause-block-loop-execution-1cb713fbdf2d

οι καλοί γείτονες

```
findBorders('Italy');
```

```
function findBorders(country) {
        fetch(urlCountry + country)
        .then((response) => {
                if (response.status === 200) {
                        return response.json();
                } else throw new Error(response.status);
        })
        .then((data) => {
                if (data[0].borders.length > 0) { // οι κωδικοί των γειτόνων
                        const theCountries = [];
                        data[0].borders.forEach((item) => {
                                theCountries.push(loadCountryNameFromCode(item));
                        });
                        Promise.all(theCountries) //  array από Promises
                        .then((allCountriesNames) => {
                                console.log(
        Η χώρα ${country} συνορεύει με τις εξής χώρες: ${allCountriesNames}
                        ); // render the result
        })
.catch((error) => { console.log(error); });
}});}
```

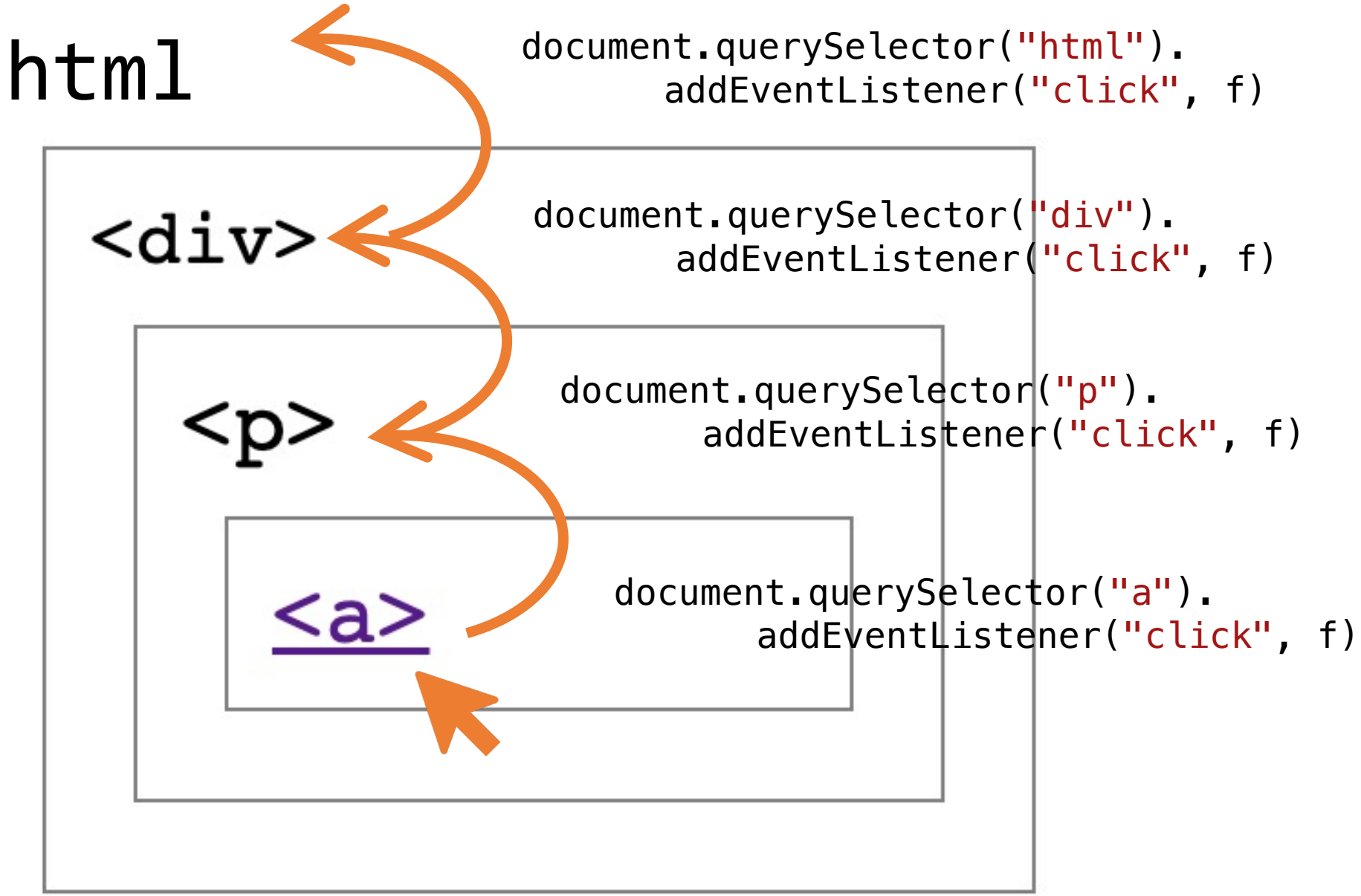
Διαχείριση συμβάντων

Κλήση χειριστή συμβάντος

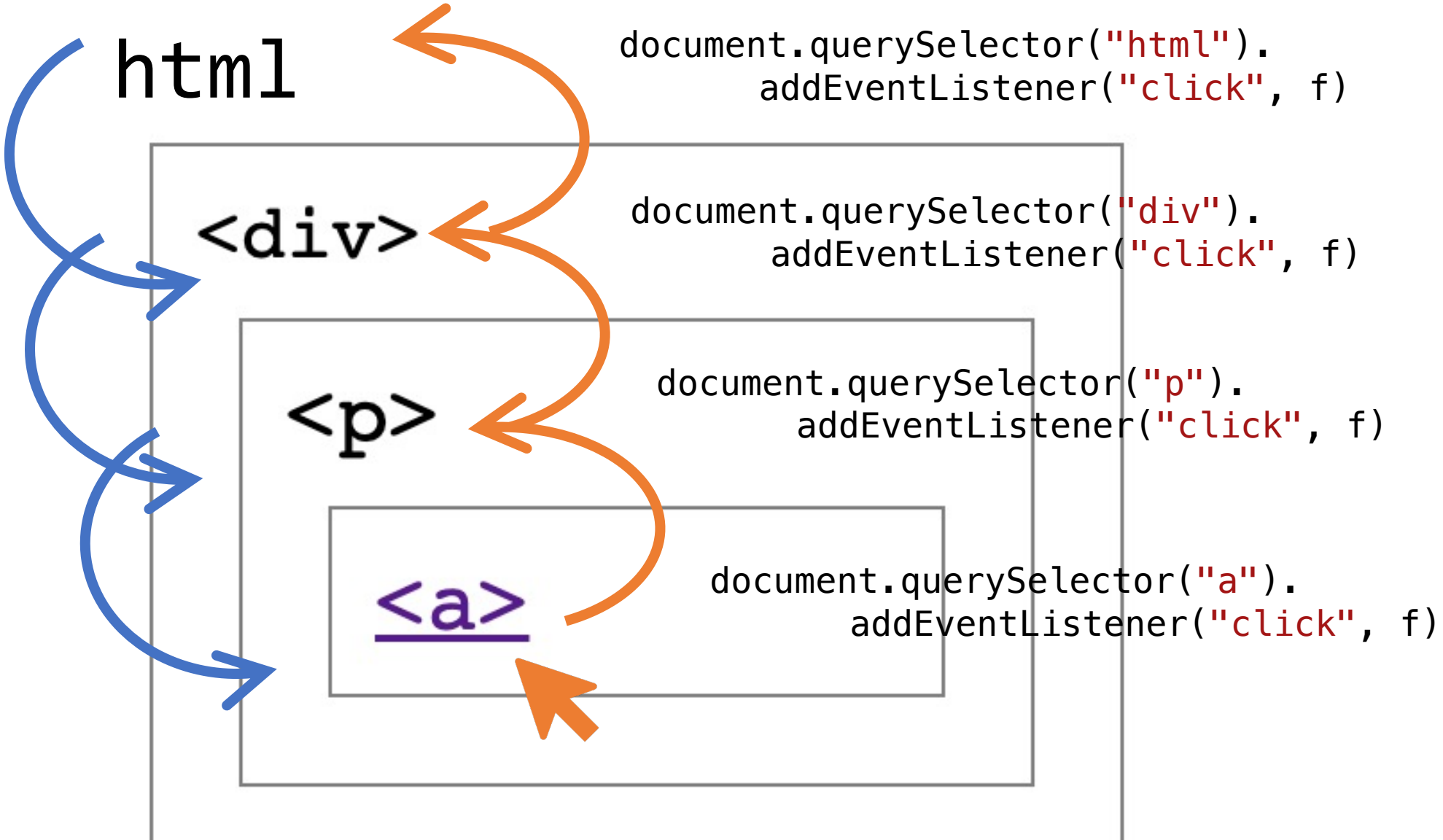
Κατά την κλήση του χειριστή συμβάντος περνάμε ως όρισμα το αντικείμενο event. ιδιότητες:
 event.type τον τύπο του συμβάντος.
 event.target το αντικείμενο στο οποίο έχει γίνει το συμβάν.
 event.currentTarget το αντικείμενο στο οποίο έχει συνδεθεί ο χειριστής.
 event.timeStamp Η χρονική στιγμή κατά την οποία έγινε το συμβάν,
 event.isTrusted (true) αν προέκυψε από φυλλομετρητή (false) από JavaScript.

Επιπροσθέτως το αντικείμενο event μπορεί να έχει και άλλες ιδιότητες ανάλογα με τον τύπο του.
Για παράδειγμα το συμβάν click : event.clientX και event.clientY
keydown: event.keyCode, κλπ.

Ο μηχανισμός φυσαλίδας

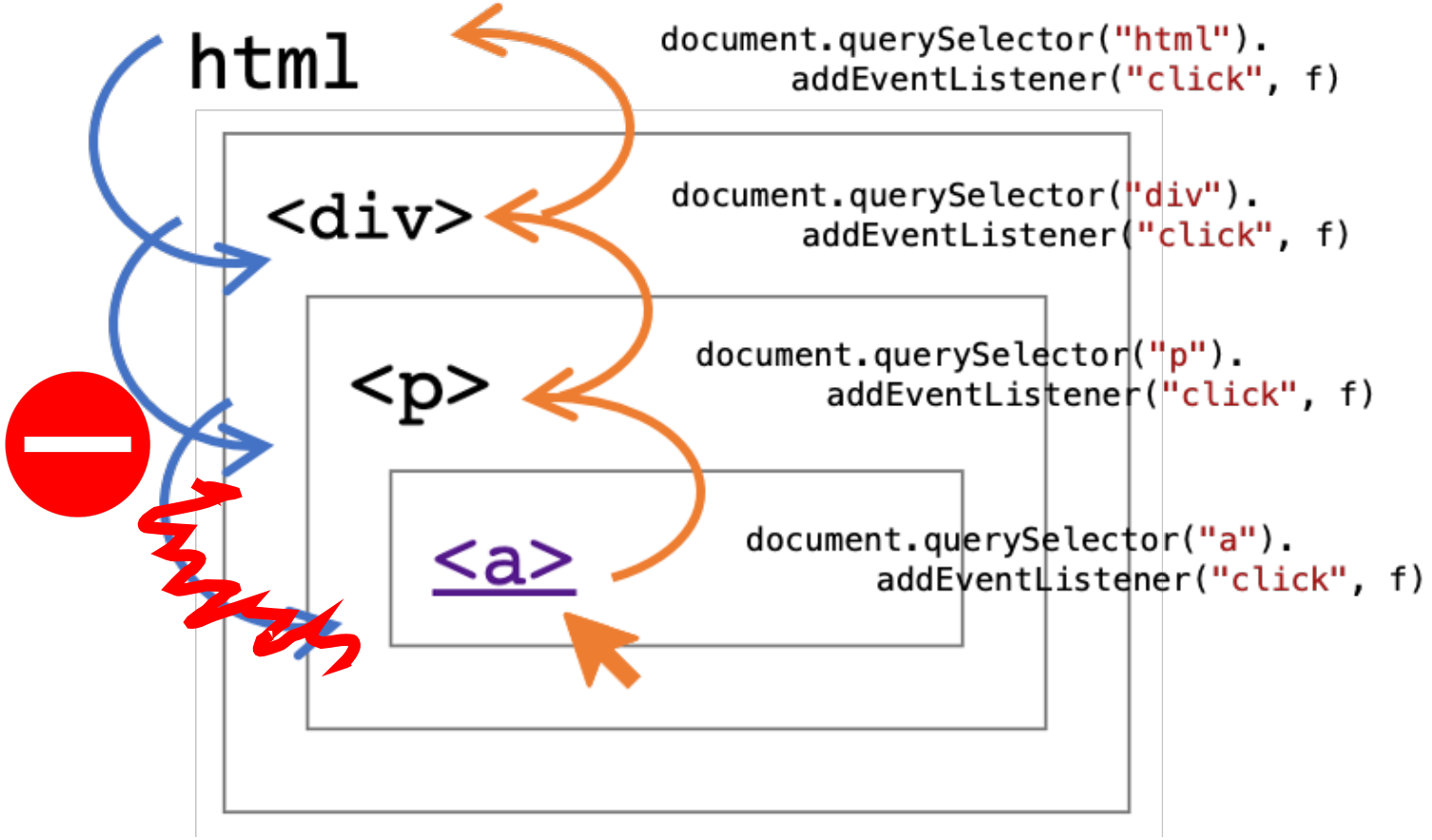


Φάση σύλληψης συμβάντος (event capturing)

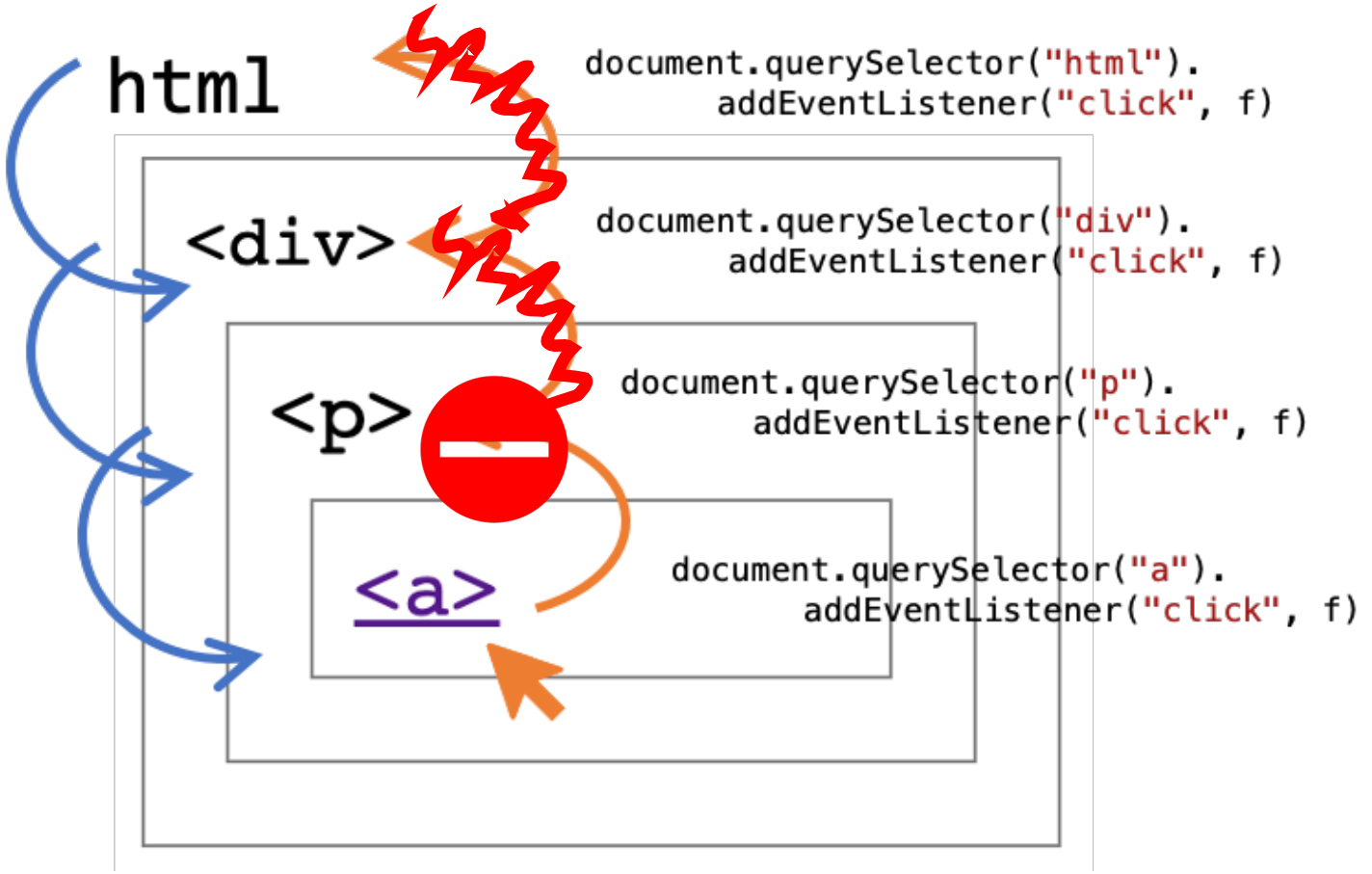


διακοπή της φάσης σύλληψης συμβάντος

```
event.stopPropagation()
```



διακοπή στη διάδοση φυσαλίδας



event.stopPropagation()

Αποτροπή προκαθορισμένης συμπεριφοράς

event.preventDefault()

Παραδείγματα προκαθορισμένης συμπεριφοράς

- όταν ο χρήστης επιλέξει ένα υπερσύνδεσμο ο φυλλομετρητής ξεκινάει κλήση για φόρτωμα της ιστοσελίδας στόχου,
- όταν ο χρήστης πληκτρολογήσει ένα χαρακτήρα σε ένα πλαίσιο κειμένου, ο φυλλομετρητής θα εμφανίσει τον χαρακτήρα στο πλαίσιο,
- όταν σύρει το δάχτυλό του σε μια οθόνη αφής θα προκληθεί κύλιση της οθόνης ή μετάβαση σε προηγούμενη σελίδα,
- όταν επιλέξει το πλήκτρο "Υποβολή" σε μια φόρμα, θα σταλθεί το περιεχόμενο της φόρμας στον ορισμένο αποδέκτη.

throw new Error

Η εντολή throw επιστρέφει αντικείμενο τύπου Error με συμβολοσειρά ως μήνυμα σφάλματος.

```
<script>
function myFunction() {
  var message, x;
  message = document.getElementById("p01");
  message.innerHTML = "";
  x = document.getElementById("demo").value;
  try {
    if(x == "") throw "empty";
    if(isNaN(x)) throw "not a number";
    x = Number(x);
    if(x < 5) throw "too low";
    if(x > 10) throw "too high";
  }
  catch(err) {
    message.innerHTML = "Input is " + err;
  }
}
</script>
```