Κεφάλαιο 10. Ασύγχρονη JavaScript – διαχείριση συμβάντων

Σύνοψη

Στο κεφάλαιο αυτό θα παρουσιάσουμε τους μηχανισμούς που διαθέτει η JavaScript για ασύγχρονη εκτέλεση κώδικα και στη συνέχεια θα εστιάσουμε στον μηχανισμό διαχείρισης συμβάντων (events).

Η ασύγχρονη εκτέλεση κώδικα είναι απαίτηση που συναντάμε συχνά στον φυλλομετρητή (διαχείριση συμβάντων και ασύγχρονη πρόσβαση σε πόρους του διαδικτύου) αλλά και στο περιβάλλον του εξυπηρετητή, της Node.js, όπως θα δούμε σε επόμενα κεφάλαια.

- *Στην **ακολουθιακή (σύγχρονη)** εκτέλεση ενός προγράμματος η μια 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 JavaScript έχει δύο μηχανισμούς ασύγχρονης εκτέλεσης:

- *Την **κλήση συνάρτησης επιστροφής** (callback function),*
- *Τον **μηχανισμό Promise**, όπως θα δούμε στη συνέχεια. Ο μηχανισμός αυτός έχει επιπλέον υλοποιηθεί ως μηχανισμός **async/await**, όπως θα δούμε στο επόμενο κεφάλαιο.*

Στο κεφάλαιο αυτό θα ολοκληρώσουμε την επισκόπηση της JavaScript. Θα δούμε ότι οι λειτουργίες της γλώσσας αποτελούν υπόβαθρο για τα επόμενα κεφάλαια, στα οποία θα ασχοληθούμε με το πώς μπορούμε να εξυπηρετήσουμε αιτήματα στην πλευρά του εξυπηρετητή με την JavaScript.

Προαπαιτούμενη γνώση

Είναι απαραίτητη η εξοικείωση με τα κεφάλαια 7, 8 και 9.

10.1 Ασύγχρονη εκτέλεση κώδικα

Η JavaScript είναι μονο-νηματική γλώσσα εκ κατασκευής. Δηλαδή, στην τυπική λειτουργία της εκτελεί τις εντολές τη μια μετά την άλλη ακολουθιακά. Η λειτουργία αυτή λέγεται **σύγχρονη λειτουργία**.

Όμως, η JavaScript έχει συχνά ανάγκη για διαχείριση ασύγχρονων λειτουργιών, όπως το να εκτελεστεί ένα τμήμα του κώδικα μετά παρέλευση κάποιου χρόνου είτε μετά από ένα συμβάν το οποίο θα προκληθεί από κάποιον έξω προς το περιβάλλον της γλώσσας, π.χ. από τον χρήστη, από το δίκτυο ή από άλλες διεργασίες του λειτουργικού συστήματος.

Δεν θα πρέπει να ξεχνάμε πως η τυπική λειτουργία ενός κώδικα JavaScript στον φυλλομετρητή, όπως ήδη περιγράψαμε, είναι ότι, αφού περάσει αρχικά από τη φάση φορτώματος του κώδικα και της αρχικής εκτέλεσης, εισέρχεται και παραμένει σε κατάσταση αναμονής συμβάντων (**ασύγχρονη λειτουργία**).

Παραδείγματα ασύγχρονης εκτέλεσης κώδικα που θα συζητήσουμε στη συνέχεια ή που έχουμε ήδη δει είναι:

- Με κλήση των μεθόδων του αντικείμενου window **setTimeout(f, t)**, η οποία επιτρέπει την εκτέλεση μιας συνάρτησης f μετά από παρέλευση ορισμένου χρόνου t, ή της **setInterval(f, t)** η οποία επιτρέπει την εκτέλεση της συνάρτησης f επαναληπτικά κάθε t ms.
- Με τον ορισμό **χειριστών συμβάντων**, π.χ. button.addEventListener(event, handler), όπως έχουμε ήδη δει σε προηγούμενα παραδείγματα.
- Με τη χρήση της διεπαφής **fetch()** που χρησιμεύει για ανάκτηση δεδομένων από εξυπηρετητή, με χρήση του μηχανισμού ασύγχρονης λειτουργίας *Promise*.
- Με την κλήση του **requestAnimationFrame** για επαναληπτική εκτέλεση κώδικα ώστε να επιτύχουμε κίνηση γραφικών στοιχείων (animations).

10.1.1 Συναρτήσεις setTimeout() και setInterval()

Κλήση συνάρτησης επιστροφής με καθυστέρηση

Μπορούμε να καλέσουμε μια συνάρτηση επιστροφής μετά από κάποια καθυστέρηση με κλήση της μεθόδου setTimeout() του global object **window**.

```
setTimeout(συνάρτηση επιστροφής, delayInMsec);
```

Για παράδειγμα, στον παρακάτω κώδικα η συνάρτηση console.log() που περνάμε ως όρισμα στην setTimeout θα εκτελεστεί μετά παρέλευση 2000 msec, με αποτέλεσμα να εμφανιστούν πρώτα τα μηνύματα: «Πριν», «Μετά», και μετά με καθυστέρηση 2΄΄ τυπώνεται το μήνυμα «Με καθυστέρηση 2sec».

```
console.log('Πριν');
setTimeout( () => {
    console.log('Με καθυστέρηση 2sec');
}, 2000)
console.log('Μετά');
```

Στο παράδειγμα η συνάρτηση επιστροφής ορίστηκε ως ανώνυμη συνάρτηση, ως πρώτο όρισμα της setTimeout(), ενώ το δεύτερο όρισμα ορίζει την καθυστέρηση σε msec.

Εναλλακτικά, μπορούμε να περάσουμε μια επώνυμη συνάρτηση ως όρισμα της setTimeout. Στην περίπτωση αυτή τα ορίσματα της συνάρτησης, αν υπάρχουν, θα ακολουθούν το όρισμα *delay*. Για παράδειγμα, εναλλακτικά ο παραπάνω κώδικας μπορεί να γραφτεί ως:

```
console.log('Πριν');
setTimeout( console.log, 2000, 'Με καθυστέρηση 2sec');
console.log('Μετά');
```

Άσκηση

Ποιο το αποτέλεσμα του παρακάτω κώδικα;

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

Απάντηση

```
αρχή
τέλος
timeout2
timeout1
```

Επαναληπτική κλήση συνάρτησης με κάποιο διάλειμμα

Η δεύτερη μέθοδος του global object η οποία επιτρέπει την κλήση συνάρτησης με καθυστέρηση είναι η setInterval().

Η μέθοδος αυτή παίρνει δύο ορίσματα, το πρώτο είναι η συνάρτηση επιστροφής η οποία καλείται επαναληπτικά και το δεύτερο είναι το διάστημα σε ms, ανάμεσα σε δύο διαδοχικές κλήσεις. Η συνάρτηση αυτή είναι χρήσιμη για επαναληπτικές διαδικασίες, και για τον λόγο αυτό χρησιμοποιείται σε κίνηση αντικειμένων (animations).

Η setInterval() επιστρέφει μια μοναδική ταυτότητα του interval και αυτή η ταυτότητα μπορεί να χρησιμοποιηθεί για τη διαγραφή του με κλήση της clearInterval().

```
const id = setInterval(func, interval);
clearInterval(id);
```

Άσκηση

Ποιο το αποτέλεσμα του παρακάτω κώδικα;

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

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

Απάντηση

```
1 'Γεια '
2 'Γεια '
3 'Γεια '
4 'Γεια '
```

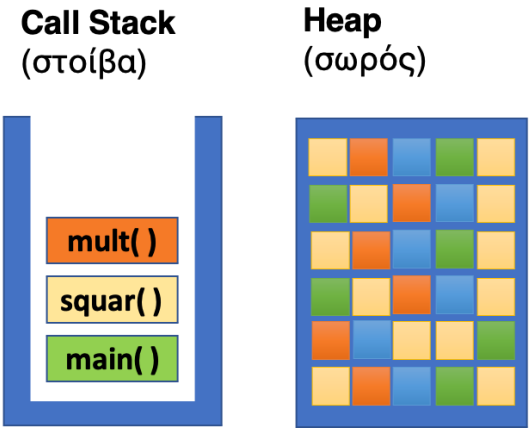
Πριν προχωρήσουμε σε πιο λεπτομερή περιγραφή των μηχανισμών ασύγχρονης λειτουργίας που διαθέτει η γλώσσα, θα πρέπει να περιγράψουμε τον μηχανισμό που ονομάζεται βρόχος ελέγχου συμβάντων (event loop).

10.1.2 Ο βρόχος ελέγχου συμβάντων

Ο **βρόχος ελέγχου συμβάντων** (event loop) είναι ο μηχανισμός της JavaScript ο οποίος επιτρέπει την επιλογή εργασιών από την **ουρά κλήσεων συναρτήσεων επιστροφής (callback queue)** μόνο όταν η **στοίβα (call stack)** είναι κενή (δεν υπάρχουν εργασίες για εκτέλεση).

Η σύγχρονη εκτέλεση κώδικα στηρίζεται σε δύο δομές της μνήμης του υπολογιστή μας (όπως και σε πολλές άλλες γλώσσες προγραμματισμού): Στον **σωρό (heap)** και στη **στοίβα κλήσεων (call stack)**.

Ο **σωρός** είναι ο χώρος της μνήμης όπου αποθηκεύονται τα δεδομένα που χειρίζεται το πρόγραμμά μας. Εκεί υπάρχουν ο κοινός χώρος διευθύνσεων (global object), τα αντικείμενα και οι άλλοι τύποι δεδομένων στα οποία αναφέρεται το πρόγραμμά μας. Η **στοίβα κλήσεων** είναι μια δομή στην οποία τοποθετούνται η μια μετά την άλλη οι συναρτήσεις με τη σειρά που καλούνται. Όταν το πρόγραμμα καλεί μια νέα συνάρτηση, ένα νέο πλαίσιο κώδικα (frame) τοποθετείται στην κορυφή της στοίβας κλήσεων. Στο κάθε πλαίσιο υπάρχουν οι τοπικές μεταβλητές της συνάρτησης. Η στοίβα είναι δομή LIFO (last-in first-out). Η διαδοχική κλήση συναρτήσεων (π.χ. όταν μια συνάρτηση καλεί μια άλλη συνάρτηση) συσσωρεύει διαδοχικά πλαίσια, όπως φαίνεται στην **Εικόνα *10.1***:



Εικόνα 10.1 *Δομές μνήμης κατά την εκτέλεση ενός προγράμματος JavaScript: στοίβα κλήσεων και σωρός. Η main() έχει καλέσει τη squar και αυτή με τη σειρά της τη mult(), η οποία εκτελείται τώρα. Όταν τελειώσει η εκτέλεσή της, το πλαίσιο της mult() θα αφαιρεθεί από τη στοίβα κλήσεων και ο έλεγχος θα περάσει στη squar().*

Ας υποθέσουμε ότι εκτελείται ένα πρόγραμμα χωρίς ασύγχρονες λειτουργίες όπως το παρακάτω:

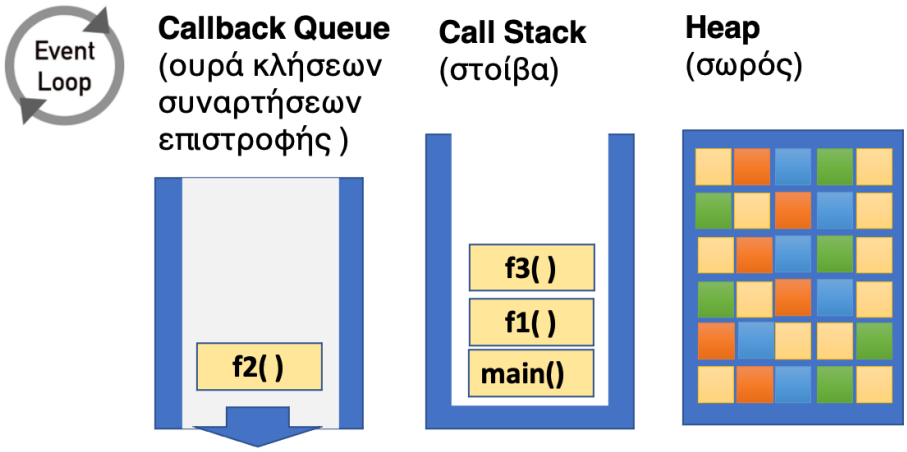
```
function mult(a,b) {
    return a*b
};

function squar(n) {
    return mult(n,n)
};
```

```
console.log(squar(5));
```

Στην ***Εικόνα 10.2*** φαίνεται ότι στη στοίβα κλήσεων έχει φορτωθεί αρχικά το πλαίσιο main(), το οποίο κάλεσε τη συνάρτηση squar() και η οποία με τη σειρά της κάλεσε τη συνάρτηση mult() που εκτελείται κατά τη συγκεκριμένη στιγμή. Όταν η mult() ολοκληρώσει την εκτέλεσή της, θα ελευθερώσει τη στοίβα κλήσεων και θα περάσει τον έλεγχο στη συνάρτηση squar() μέχρι να αδειάσει η στοίβα κλήσεων.

Στη συνέχεια, θα εξετάσουμε πώς διαφοροποιείται αυτός ο μηχανισμός όταν υπάρχουν ασύγχρονα τμήματα κώδικα.



Εικόνα 10.2 *Δομές μνήμης κατά την εκτέλεση ενός προγράμματος JavaScipt με εκτέλεση ασύγχρονων τμημάτων κώδικα: χρήση ουράς κλήσεων συναρτήσεων επιστροφής.*

Ας παρακολουθήσουμε την εκτέλεση του παρακάτω κώδικα βήμα προς βήμα.

```
function f2() {console.log("f2") };

function f3() {console.log("f3") };

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

f1();
```

Στην περίπτωση αυτή ενεργοποιείται ο βρόχος ελέγχου συμβάντων της JavaScript, αφού στον μηχανισμό έχει προστεθεί ένα ακόμη στοιχείο που είναι η **ουρά κλήσεων συναρτήσεων επιστροφής**. Πρόκειται για μια ουρά FIFO (first-in first-out) στην οποία τοποθετούνται τμήματα του κώδικα τα οποία έχουν κληθεί να εκτελεστούν με ασύγχρονο τρόπο ως συνέπεια κάποιου συμβάντος. Στην ***Εικόνα 10.2*** φαίνεται ότι η συνάρτηση f2() τη στιγμή αυτή είναι σε αναμονή στην ουρά κλήσεων συναρτήσεων επιστροφής. Το ερώτημα είναι πότε θα εκτελεστεί η συνάρτηση αυτή σε σχέση με τα τμήματα του κώδικα που βρίσκονται στη στοίβα.

Ας παρακολουθήσουμε την εκτέλεση του κώδικα. Όταν φορτωθεί το πρόγραμμα (πλαίσιο main()), η πρώτη εντολή είναι η κλήση της συνάρτησης f1() και στη συνέχεια καλείται μέσα από τη συνάρτηση αυτή η setTimeout(f2,0), η οποία άμεσα τοποθετεί τη συνάρτηση f2 στην ουρά κλήσεων συναρτήσεων επιστροφής (callback queue), αφού ο χρόνος αναμονής της είναι 0 ms. Όμως, ο μηχανισμός ασύγχρονης εκτέλεσης της JavaScript δεν επιτρέπει την εκτέλεση κώδικα που βρίσκεται στην ουρά αυτή ενόσω η στοίβα κλήσεων είναι γεμάτη. Στη συνέχεια καλείται η f3() και τοποθετείται στη στοίβα κλήσεων. Τέλος, μετά την ολοκλήρωση εκτέλεσής της, η στοίβα επιτέλους αδειάζει. Τότε μόνο η f(2) μεταφέρεται από την ουρά των κλήσεων συναρτήσεων επιστροφής στη στοίβα και εκτελείται. Συνεπώς, η σειρά εμφάνισης των αποτελεσμάτων του κώδικα αυτού είναι f1, f3, f2.

Στη συνέχεια θα δούμε μια άλλη περίπτωση ορισμού συνάρτησης περιστροφής: τον ορισμό χειριστή συμβάντων.

10.2 Ορισμός χειριστή συμβάντων

Όπως έχει ήδη αναφερθεί, η JavaScript στο περιβάλλον του φυλλομετρητή ακολουθεί το μοντέλο του **προγραμματισμού συμβάντων**, όπως και άλλες γλώσσες προγραμματισμού γραφικών διεπαφών χρήστη.

Ο κώδικας JavaScript αφού φορτωθεί περιμένει. Το περιβάλλον του φυλλομετρητή γεννάει συμβάντα, ως αποτέλεσμα, για παράδειγμα, ενεργειών του χρήστη. Το πρόγραμμά μας έχει καταχωρίσει «χειριστές», συναρτήσεις δηλαδή που περιμένουν αυτά τα συμβάντα. Οι χειριστές καταχωρίζονται σε έναν πίνακα **συμβάντων**. Όταν προκύψει το συμβάν, η JavaScript ανατρέχει στον πίνακα συμβάντων και ανασύρει τον χειριστή που θα πρέπει να ενεργοποιηθεί για το αντίστοιχο συμβάν. Θέτει τον χειριστή στην ουρά κλήσεων συναρτήσεων επιστροφής και ο χειριστής ενεργοποιείται όταν αδειάσει η στοίβα κλήσεων. Ο μηχανισμός ορισμού συμβάντων είναι αρκετά σύνθετος (στηρίζεται στο αντικείμενο **Event** και τις ιδιότητες και μεθόδους του) και θα συζητηθεί πιο εκτενώς στη συνέχεια. Εδώ κάνουμε επισκόπηση των διαφόρων κατηγοριών συμβάντων και διαφόρων τρόπων ορισμού χειριστών τους.

Έχει ενδιαφέρον να παρατηρήσουμε ότι αποτέλεσμα του παραπάνω μηχανισμού (βρόχος χειρισμού συμβάντων) είναι ότι ένα πρόγραμμα JavaScript ποτέ δεν σταματάει την εκτέλεσή του λόγω εξαίρεσης ή σφάλματος. Αν προκύψει ένα σφάλμα σε έναν χειριστή, γεννιέται ένα αντικείμενο **Error** το οποίο παράγει διαγνωστικά μηνύματα, όμως ο χειριστής συμβάντων συνεχίζει να λειτουργεί και τα επόμενα συμβάντα θα προκαλέσουν ασύγχρονη κλήση άλλων χειριστών.

10.2.1 Κατηγορίες συμβάντων

Τα πιο σημαντικά συμβάντα προκύπτουν από ενέργειες του χρήστη, χρήση του ποντικιού ή άλλης δεικτικής συσκευής ή χειρισμοί σε οθόνη αφής, πληκτρολογήσεις κ.λπ. Όμως, υπάρχουν πολλές κατηγορίες συμβάντων, ενώ οι διάφορες εκδόσεις του DOM level 2.0, 3.0 κ.λπ. έχουν οδηγήσει σε διαφοροποιήσεις του μοντέλου συμβάντων μεταξύ των φυλλομετρητών.

Μια πρώτη κατηγοριοποίηση των συμβάντων διακρίνει:

1. **Συμβάντα εξαρτώμενα από συσκευή**. Αυτά τα συμβάντα εξαρτώνται από το είδος της συσκευής, δηλαδή αν πρόκειται για ποντίκι, πληκτρολόγιο ή οθόνη αφής. Τέτοια συμβάντα είναι: mousedown, mousemove, mouseup, touchstart, touchmove, touchend, keydown, keyup κ.λπ.
2. **Συμβάντα ανεξάρτητα από συσκευή**. Το πιο κλασικό παράδειγμα είναι το συμβάν click το οποίο δηλώνει ότι ένας υπερσύνδεσμος ή ένα πλήκτρο έχουν πατηθεί, ενέργεια που μπορεί να γίνει με ποντίκι, πληκτρολόγιο ή το δάχτυλο. Άλλο παράδειγμα είναι το συμβάν input, το οποίο είναι ισοδύναμο του keydown, όμως, εκτός από είσοδο από το πληκτρολόγιο σε ένα στοιχείο εισαγωγής κειμένου, υποστηρίζει και επικόλληση κειμένου. Επίσης, τα συμβάντα pointerdown, pointermove αντιστοιχούν στα αντίστοιχα συμβάντα με ποντίκι ή αφή.
3. **Συμβάντα διεπαφής**. Τα συμβάντα αυτά περιγράφουν την κατάσταση των στοιχείων της διεπαφής. Παραδείγματα είναι τα συμβάντα focus, change (αλλαγή περιεχομένου ενός στοιχείου μιας φόρμας), submit όταν επιλέγεται το πλήκτρο υποβολής μιας φόρμας.
4. **Συμβάντα αλλαγής κατάστασης**. Τα συμβάντα αυτά δεν προκύπτουν κατευθείαν από ενέργειες του χρήστη, αλλά σχετίζονται με την κατάσταση του φυλλομετρητή ή προκύπτουν από το δίκτυο. Παράδειγμα τέτοιων συμβάντων είναι το συμβάν load του αντικείμενου Window, καθώς και το συμβάν DOMContentLoaded του αντικείμενου Document, τα οποία είδαμε στην ενότητα περιγραφής του φορτώματος της ιστοσελίδας στον φυλλομετρητή (7.6). Άλλο παράδειγμα είναι το συμβάν online καθώς και offline που αφορούν σύνδεση και αποσύνδεση του φυλλομετρητή στο διαδίκτυο.
5. **Συμβάντα ειδικών API**. Τέλος, μια άλλη κατηγορία συμβάντων προκύπτουν από προγραμματιστικές διεπαφές, όπως των στοιχείων <video> και <audio> της HTML, του XMLHttpRequest API κ.λπ.

10.2.2 Καταχώριση χειριστών συμβάντων

Υπάρχουν δύο διαφορετικοί τρόποι για να οριστεί ένας χειριστής συμβάντων: είτε ως τιμή στην αντίστοιχη ιδιότητα του σχετικού αντικείμενου (π.χ. του στοιχείου του DOM) είτε, η πιο πρόσφατη προσέγγιση, μέσω της μεθόδου addEventListener() του αντίστοιχου αντικείμενου, στην οποία περνάμε τον χειριστή και τον τύπο του

συμβάντος ως ορίσματά της.

Χειριστής συμβάντων ως ιδιότητα του αντικείμενου

Κατά σύμβαση, οι ιδιότητες των αντικειμένων οι οποίες αντιστοιχούν σε συμβάντα έχουν όνομα "on"+συμβάν. Για παράδειγμα, onclick, onchange, onload. Θέλει προσοχή το γεγονός ότι οι ιδιότητες αυτές δεν ακολουθούν τη συνηθισμένη σύμβαση **camelCase** που έχουμε ως τώρα δει στην JavaScript.

Παραδείγματος χάρη, αν επιθυμούμε να φορτώνεται ένα τμήμα του κώδικα όταν ολοκληρωθεί το φόρτωμα της ιστοσελίδας (όταν δηλαδή προκύψει το συμβάν load στο αντικείμενο window), τότε:

```
function pageScript() {
    // ο κώδικας
}
window.onload = pageScript;
```

ή η πιο συνηθισμένη έκδοση:

```
window.onload = function() {
    // ο κώδικας
}
```

Επίσης, ο ορισμός της ιδιότητας αυτής μπορεί να γίνει μέσα στον κώδικα HTML (όχι καλή πρακτική) ως γνώρισμα του αντίστοιχου στοιχείου HTML:

```
<button onclick= "console.log('ευχαριστώ')">πατήστε</button>
```

Χειριστής συμβάντων με μέθοδο addEventListener()

Ένας εναλλακτικός τρόπος ορισμού ενός χειριστή συμβάντων, που είναι και ο προτεινόμενος τρόπος, είναι με χρήση της μεθόδου addEventListener() του αντικείμενου στο οποίο επισυνάπτουμε τον χειριστή. Έστω ότι επιθυμούμε να ορίσουμε χειριστή για το συμβάν επιλογής ενός πλήκτρου <**button**>.

```
<button>πατήστε</button>
<script>
    function f(event) {
        console.log('πλήκτρο πατήθηκε οκ');
    }
    // ορισμός συνάρτησης επιστροφής, (χειριστής συμβάντος "click")
    document.querySelector('button').addEventListener('click', f);
</script>
```

Εναλλακτικός τρόπος ορισμού της συνάρτησης στα ορίσματα της μεθόδου addEventListener() ως ανώνυμης συνάρτησης βέλους:

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

Θα επανέλθουμε στο τέλος του κεφαλαίου με τη συζήτηση του μηχανισμού διαχείρισης συμβάντων της JavaScript.

Το παράδειγμα πελάτη-κούριερ

Ένα κάπως πιο σύνθετο παράδειγμα ορισμού χειριστών συμβάντων που έχει ως συνέπεια την επικοινωνία μεταξύ των αντικειμένων δύο κλάσεων που περιμένουν ένα συμβάν για αλλαγή της κατάστασής τους είναι το παράδειγμα προσομοίωσης της καθυστέρησης ενός κούριερ μιας μεταφορικής εταιρείας. Ορίζουμε δύο κλάσεις – στο παράδειγμα αυτό πρώτη την κλάση Pelatis:

```
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>";
    }
}
```

Στη συνέχεια ορίζουμε την κλάση Courier:

```
class Courier{
    // Ο κούριερ που παραδίδει δέμα 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);
    }
}
```

Τέλος, αρχικοποιούμε δύο αντικείμενα των κλάσεων ως εξής:

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

Το αποτέλεσμα που θα προκύψει είναι:

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

Ως τώρα έχουμε δει διάφορα παραδείγματα ορισμού συναρτήσεων επιστροφής που καλούνται είτε μετά παρέλευση ορισμένου χρόνου είτε όταν προκύψει ένα συμβάν που περιμένουν.

Ας δούμε όμως στη συνέχεια προβλήματα που παρουσιάζονται κατά τον προγραμματισμό με χρήση αυτού του μοντέλου ασύγχρονης εκτέλεσης κώδικα.

10.3 Προγραμματισμός με κλήση συναρτήσεων επιστροφής

Έχουμε δει ως τώρα παραδείγματα από συναρτήσεις που καλούνται ασύγχρονα όταν προκύψει κάποιο συμβάν. Ένα πρόβλημα που εμφανίζεται σε αυτό το προγραμματιστικό μοντέλο είναι η δυσκολία που έχουμε να χρησιμοποιήσουμε το αποτέλεσμα που παράγει η συνάρτηση επιστροφής. Το αποτέλεσμα αυτό δεν είναι

διαθέσιμο κατά την κλήση της συνάρτησης, συνεπώς δεν μπορούμε να το εκχωρήσουμε σε μια μεταβλητή, όπως γίνεται με την ακολουθιακή (σύγχρονη) κλήση μιας συνάρτησης.

Ας δούμε ένα παράδειγμα. Έστω η συνάρτηση double(), η οποία μετά την παρέλευση κάποιου χρόνου (2 δευτερόλεπτα) καλεί τη συνάρτηση f() η οποία εκτελεί έναν υπολογισμό και επιστρέφει το αποτέλεσμα.

```
function f(v) {
    return v * 2;
}

function double(value) {
    setTimeout(f(value), 2000);
}

x = double(5);
console.log(x);
```

Το αποτέλεσμα που θα μας επιστρέψει ο παραπάνω κώδικας είναι undefined αφού η double() κατά την κλήση της δεν έχει επιστρέψει το αποτέλεσμά της, αφού εκτελείται ασύγχρονα. Ποιος είναι λοιπόν ο τρόπος για να πάρουμε το αποτέλεσμα της double;

Ένας τρόπος για να λύσουμε το πρόβλημα αυτό είναι να περάσουμε ως όρισμα στην double() μια συνάρτηση που θα αναλάβει να διαχειριστεί το αποτέλεσμα της ασύγχρονης συνάρτησης όταν αυτό είναι διαθέσιμο.

Έστω ότι αυτή είναι η συνάρτηση με όνομα callback(). Ξαναγράφουμε συνεπώς τη συνάρτηση double() που εκτελεί το ασύγχρονο έργο ως εξής:

```
function double(value, callback) {
    setTimeout(
        () => {
            callback(f(value));
        },
        2000);
}
```

Παρατηρούμε ότι αυτή τη φορά περάσαμε στην double() δύο ορίσματα: την αρχική τιμή και τη συνάρτηση callback() που θα πάρει το αποτέλεσμα της f().

Όταν καλέσουμε την double(), θα πρέπει να φροντίσουμε να περάσουμε ως δεύτερο όρισμα μια κατάλληλη συνάρτηση η οποία θα παραλάβει το αποτέλεσμα της f(). Για παράδειγμα:

```
double(5, (x) => {
  console.log(`αποτέλεσμα = ${x}`);
});
```

Αυτή τη φορά όταν τρέξουμε τον κώδικα θα παρατηρήσουμε ότι τίποτα δεν θα συμβεί αρχικά, όμως μετά από περίπου 2 δευτερόλεπτα θα δούμε να εμφανίζεται το μήνυμα:

```
'αποτέλεσμα = 10'
```

Αυτός είναι ένας συνηθισμένος τρόπος χειρισμού αποτελεσμάτων ασύγχρονων συναρτήσεων.

Ένα επόμενο ερώτημα που ανακύπτει είναι πώς να χειριστούμε την περίπτωση που η ασύγχρονη συνάρτηση δεν επιστρέφει το προσδοκώμενο αποτέλεσμα γιατί, για παράδειγμα, ο υπολογισμός απέτυχε (π.χ. ένας εξωτερικός πόρος, όπως ο εξυπηρετητής, δεν είναι διαθέσιμος). Για να καλύψουμε και αυτό το πρόβλημα, θα πρέπει να προβλέψουμε να περάσουμε στην double() μια ακόμη συνάρτηση που θα χειριστεί το σφάλμα, αν αυτό προκύψει.

Μια νεότερη έκδοση της double() που προβλέπει και αυτό το ενδεχόμενο είναι η εξής:

```
function double(value, success, failure) {
    setTimeout(() => {
        const result = f(value);
        if (typeof result !== 'number') {
            failure();
```

```
            } else {
                success(result);
            }
        }, 2000);
    }
```

Η κλήση στη συνάρτηση αυτή τη φορά έχει την εξής μορφή:

```
    double(
        5,
        (x) => {
            console.log(`αποτέλεσμα = ${x}`);
        },
        () => console.log('Σφάλμα: δεν επέστρεψε αποτέλεσμα')
    );
```

Σε αυτό το παράδειγμα έχουμε φροντίσει να περάσουμε ως ορίσματα στην double() δύο συναρτήσεις, που η πρώτη χειρίζεται το αποτέλεσμα που επιστρέφει η ασύγχρονη συνάρτηση και η δεύτερη παράγει ένα μήνυμα σφάλματος αν η ασύγχρονη συνάρτηση αποτύχει στον υπολογισμό.

Τα πράγματα όμως μπορεί να γίνουν ακόμη πιο σύνθετα αν πρέπει να περάσουμε σε μια συνάρτηση τα αποτελέσματα της ασύγχρονης κλήσης που και αυτή καλείται ασύγχρονα κ.ο.κ. Το πρόβλημα αυτό έχει γίνει γνωστό ως η κόλαση των ασύγχρονων κλήσεων, *callback hell*. Μια πρόταση για λύση του προβλήματος αυτού είναι ο μηχανισμός Promise που περιγράφεται στη συνέχεια.

10.4 Ο μηχανισμός Promise

Ο μηχανισμός **Υποσχέσεων (Promise)** εισήχθη στην JavaScript στην έκδοση ES6 για να απλοποιήσει την ασύγχρονη λειτουργία της γλώσσας, να λύσει το πρόβλημα διαδοχικών callbacks και για να επιτρέψει καλύτερη διαχείριση σφαλμάτων που προκύπτουν κατά την ασύγχρονη εκτέλεση κώδικα.

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

Promises χρησιμοποιούνται από την JavaScript και στον εξυπηρετητή και στον φυλλομετρητή. Παράδειγμα το Fetch API και το Service Workers API, ενώ ο πιο πρόσφατος μηχανισμός **async/await**, που θα δούμε στο επόμενο κεφάλαιο, βασίζεται επίσης στα promises.

Το αποτέλεσμα της υπόσχεσης δεν μπορούμε να το πάρουμε, για παράδειγμα, εκχωρώντας την υπόσχεση σε μια μεταβλητή, όπως κάνουμε με τη σύγχρονη εκτέλεση κώδικα.

Για να δημιουργήσουμε μια υπόσχεση, καλούμε τη συνάρτηση-δημιουργό Promise(), στην οποία περνάμε μια συνάρτηση executor() που θα εκτελέσει την ασύγχρονη εργασία. Φροντίζουμε στη συνάρτηση αυτή να περάσουμε ως ορίσματα δύο συναρτήσεις επιστροφής, τις resolve() και reject(), οι οποίες θα πρέπει να κληθούν αν η υπόσχεση ικανοποιηθεί ή όχι αντίστοιχα.

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

10.4.1 Καταστάσεις του αντικείμενου Promise

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

Η ιδιότητα state ορίζει την κατάσταση του αντικείμενου και παίρνει τις τιμές:

- "pending" Ένδειξη ότι η ασύγχρονη εργασία είναι σε εξέλιξη.
- "fulfilled" Ένδειξη ότι η ασύγχρονη εργασία έχει ολοκληρωθεί επιτυχώς.
- "rejected" Ένδειξη ότι η ασύγχρονη εργασία απέτυχε να ολοκληρωθεί.

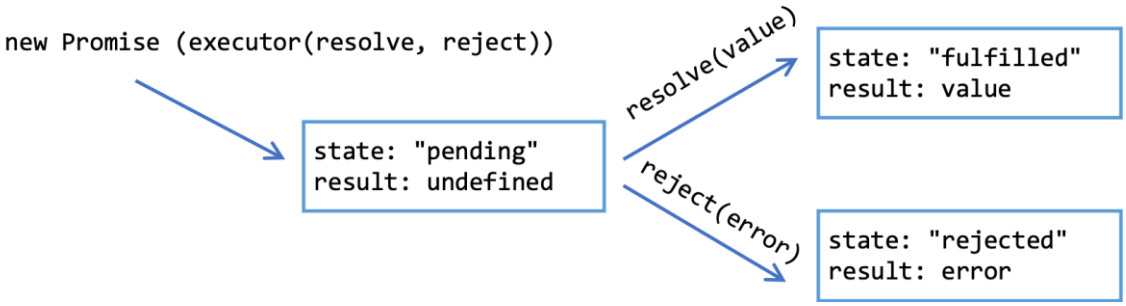
Η ιδιότητα result παίρνει αρχικά την τιμή undefined, ενώ στη συνέχεια παίρνει την τιμή του αποτελέσματος αν η υπόσχεση ικανοποιηθεί ή την τιμή σφάλματος αν όχι.

Θα πρέπει να σημειωθεί ότι, όταν αλλάξει η κατάσταση του αντικείμενου Promise, η resolve() αν ξανακαλεστεί δεν αλλάζει πλέον την κατάσταση.

Η κατάσταση μιας υπόσχεσης μπορεί να αλλάξει σε "fulfilled" ή "rejected", αλλά και να μείνει για πάντα "pending". Ο κώδικάς μας θα πρέπει να προβλέψει όλα αυτά τα ενδεχόμενα. Έχει ενδιαφέρον ότι η κατάσταση

μιας υπόσχεσης είναι ιδιωτική ιδιότητα, στην οποία δεν έχουμε πρόσβαση από την JavaScript και την οποία δεν μπορούμε να αλλάξουμε προγραμματιστικά, εκτός από τη συνάρτηση executor(). Αυτό για να αποφεύγουμε τη σύγχρονη διαχείριση της υπόσχεσης μέσω ελέγχου ή τροποποίησης της κατάστασής της.

Η συνάρτηση εκτέλεσης executor() που περνάμε στον δημιουργό της Promise καλείται αυτόματα από τον δημιουργό του αντικείμενου Promise. Ο ρόλος της συνάρτησης αυτής είναι αφενός να αρχίζει την ασύγχρονη εκτέλεση του κώδικα, αφετέρου να ελέγχει τις αλλαγές κατάστασης. Η αλλαγή της κατάστασης γίνεται με κλήση των δύο συναρτήσεων, παραμέτρων της executor(), τις οποίες ονομάζουμε συνήθως resolve, reject. Όταν κληθεί η resolve() θα αλλάξει η κατάσταση της υπόσχεσης σε "fulfilled", ενώ όταν κληθεί η reject() θα αλλάξει η κατάσταση σε "rejected".



Εικόνα 10.3 *Διάγραμμα καταστάσεων ενός αντικείμενου Promise (υπόσχεση).*

Όταν κληθεί η συνάρτηση resolve(), ως όρισμα περνάμε τα αποτελέσματα ώστε αυτά να αποτελέσουν την τιμή της ιδιότητας result της υπόσχεσης, ενώ αν κληθεί η συνάρτηση reject() σε αυτή περνάμε το αντικείμενο Error ή το μήνυμα σφάλματος, το οποίο στην περίπτωση αυτή αποτελεί την τιμή της ιδιότητας result της υπόσχεσης, όπως φαίνεται στην **Εικόνα *10.3***.

Ας δημιουργήσουμε μια υπόσχεση με συνάρτηση εκτέλεσης που άμεσα καλεί τη συνάρτηση επιστροφής resolve(). Όταν προσπαθήσουμε να εκτυπώσουμε την υπόσχεση στην κονσόλα της JavaScript θα πάρουμε το εξής αποτέλεσμα:

```
let p1 = new Promise((resolve, reject) => resolve(5));
setTimeout(() => console.log(p1), 0);
> Promise {<fulfilled>: 5}
    __proto__: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: 5
```

Αν όμως ορίσουμε ότι η resolve() καλείται με καθυστέρηση 100ms, θα πάρουμε το μήνυμα ότι η κατάσταση της υπόσχεσης είναι <pending>

```
let p1 = new Promise((resolve, reject) =>
    setTimeout(resolve 100 5);
setTimeout(() => console.log(p1), 0);
> Promise {<pending>}
```

Θα πρέπει να σημειωθεί ότι η reject(error) δεν παράγει κατάσταση σφάλματος που μπορεί να ελεγχθεί από τη δομή try/except, η οποία αφορά σφάλματα και σύγχρονη εκτέλεση κώδικα της JavaScript.

Ένα παράδειγμα:

```
try {
    Promise.reject(new Error('Σφάλμα!'));
} catch(e) {
    console.log('διαπιστώθηκε σφάλμα:', e);
}
```

Στον κώδικα αυτόν το τμήμα catch() δεν θα τυπώσει το μήνυμα, ενώ θα πάρουμε το μήνυμα: > Uncaught (in promise) Error: Σφάλμα!

Διαπιστώνουμε εδώ ότι τα σφάλματα στον ασύγχρονο κώδικά μας (στην υπόσχεση) δεν μπορούμε να τα διαχειριστούμε με τον συνήθη τρόπο που έχουμε διαθέσιμο για τον σύγχρονο κώδικα.

Η σύνδεση ανάμεσα στον σύγχρονο και στον ασύγχρονο κώδικα γίνεται με τις μεθόδους καταναλωτές του αντικείμενου Promise, που περιγράφονται στη συνέχεια.

10.4.2 Καταναλωτές υποσχέσεων

Ένας συνηθισμένος τρόπος να πάρουμε τα αποτελέσματα μιας υπόσχεσης είναι να ορίσουμε έναν ή περισσότερους *καταναλωτές* της υπόσχεσης, οι οποίοι υλοποιούνται με τις μεθόδους then(), catch(), finally() του αντικείμενου Promise.

```
p = new Promise(f)
p.then(callbackSuccess, callbackFailure);
```

Σε έναν καταναλωτή then() μπορούμε να περάσουμε δύο συναρτήσεις, την callbackSuccess() για την επεξεργασία του αποτελέσματος της υπόσχεσης και την callbackFailure() για διαχείριση του σφάλματος σε περίπτωση αποτυχίας. Επιτρέπεται σε έναν καταναλωτή then() να περάσουμε μόνο την πρώτη συνάρτηση ως όρισμα, χωρίς να ορίζουμε συνάρτηση διαχείρισης σφάλματος:

```
p.then(callbackSuccess);
```

Ακολουθεί, τέλος, ένα παράδειγμα όπου ορίζουμε μόνο τη συνάρτηση διαχείρισης σφάλματος ως εξής:

```
p.then(null, callbackFailure);
```

Εναλλακτικά, την αποτυχία εκπλήρωσης της υπόσχεσης μπορούμε να την καταναλώσουμε μέσω ενός ειδικού καταναλωτή .catch(errorHandling) ενώ ο καταναλωτής .finally(callback) εκτελείται σε κάθε περίπτωση.

Οι συναρτήσεις αυτές θα ενεργοποιηθούν όταν η υπόσχεση βρεθεί στην αντίστοιχη κατάσταση ("fulfilled" ή "rejected"). Επειδή κάθε υπόσχεση μεταβαίνει προς κάποια από τις δύο αυτές καταστάσεις μία φορά μόνο, είναι προφανές ότι η εκτέλεση των συναρτήσεων callbackSuccess() και callbackFailure() ενός καταναλωτή είναι αμοιβαία αποκλειόμενες.

Θα πρέπει να σημειωθεί ότι σε ένα στιγμιότυπο του αντικείμενο Promise μπορεί να κληθεί ένας από τους παραπάνω καταναλωτές, ο οποίος με τη σειρά του παράγει μια υπόσχεση, συνεπώς, μπορούμε να δημιουργήσουμε μια αλυσίδα καταναλωτών με σημειολογία τελείας p.then().then() .... Να σημειωθεί επίσης ότι, ακόμη και αν στη μέθοδο then() δεν έχει οριστεί συνάρτηση χειριστής, η υπόσχεση περνάει στο επόμενο then της αλυσίδας.

Παράδειγμα χρήσης καταναλωτών

```
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('Επιτυχής ολοκλήρωση')
    }
    else {
        reject('Σφάλμα')
    }
}

// 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
```

Στο παράδειγμα αυτό ορίζουμε μια συνάρτηση promisedFunc() που δέχεται δύο συναρτήσεις ως ορίσματα, τις resolve() και reject(), οι οποίες καλούνται στο σώμα της συνάρτησης διαβιβάζοντάς τους το αποτέλεσμα της υπόσχεσης ή το μήνυμα σφάλματος αντίστοιχα, αφού ενεργοποιούνται στην περίπτωση επιτυχούς ή μη επιτυχούς εκπλήρωσης της υπόσχεσης.

Στη συνέχεια καλούμε τον καταναλωτή then στον οποίο περνάμε δύο συναρτήσεις οι οποίες χειρίζονται το αποτέλεσμα της υπόσχεσης ή το σφάλμα αντίστοιχα, τυπώνοντας το σχετικό μήνυμα. Όταν τρέξουμε κατ' επανάληψη τον παραπάνω κώδικα, θα πάρουμε κάποιες φορές το μήνυμα:

```
> 'Σφάλμα'
```

ή σε άλλες περιπτώσεις το:

```
> 'Επιτυχής ολοκλήρωση'
```

10.4.3 Παράδειγμα αλυσίδας καταναλωτών υποσχέσεων

Όπως είδαμε, η κατανάλωση μιας υπόσχεσης γίνεται κυρίως με την κλήση της μεθόδου .then(f) και η μέθοδος then(), όταν εφαρμόζεται σε ένα αντικείμενο Promise, επιστρέφει Promise, συνεπώς μπορούμε να δημιουργήσουμε αλυσίδα από καταναλωτές υποσχέσεων.

Ας δούμε ένα παράδειγμα αλυσίδας καταναλωτών υποσχέσεων:

```
//αρχή προγράμματος
console.log('αρχή προγράμματος');
new Promise(function (resolve, reject) {
  setTimeout(() => resolve(1), 1000); // η υπόσχεση υλοποιείται σε 1"
})
  .then(function (result) {
    // καλείται ο 1ος καταναλωτής then()
    console.log('καταναλωτής-1: ', result);
    return result * 2; // επιστρέφει την τιμή 2
  })
  .then(function (result) {
    // καλείται ο 2ος καταναλωτής
    console.log('καταναλωτής-2: ', result);
    return result * 2; // επιστρέφει την τιμή 4
  })
  .then(function (result) {
    // καλείται ο 3ος καταναλωτής
    console.log('καταναλωτής-3: ', result);
    return result * 2;
  });
console.log('τέλος προγράμματος');
```

Το αποτέλεσμα από την εκτέλεση του προγράμματος αυτού θα είναι:

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

Άσκηση 1

Ποιο το αποτέλεσμα του παρακάτω κώδικα;

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

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

Απάντηση

Το αποτέλεσμα είναι:

```
20
```

Αυτό γιατί, όταν στην ασύγχρονη συνάρτηση της Promise η resolve() καλείται την πρώτη φορά, αλλάζει η κατάσταση του αντικείμενου, η οποία δεν μπορεί να ξαναλλάξει με επόμενες κλήσεις της resolve().

Άσκηση 2

Στην άσκηση αυτή ζητείται να υλοποιήσετε τη συνάρτηση setTimeout() που είδαμε σε προηγούμενη ενότητα, με χρήση υποσχέσεων. Η συνάρτηση setTimeout(f, d) υπενθυμίζεται ότι εκτελεί τη συνάρτηση f() ασύγχρονα μετά παρέλευση d millisecond. Ζητείται να υλοποιηθεί η ίδια συμπεριφορά μέσω μιας υπόσχεσης Promise, που περιλαμβάνει κλήση της συνάρτησης delay(d), το αποτέλεσμα της οποίας καταναλώνει στη συνέχεια μια συνάρτηση .then(f).

Απάντηση

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

Μπορούμε να καταναλώσουμε τη συνάρτηση με κλήση της then(), όπως φαίνεται στο παράδειγμα:

```
delay(3000).then(() => console.log("εκτελείται μετά από 3 sec"));
```

10.4.4 Promises στον βρόχο ελέγχου συμβάντων

Είδαμε σε προηγούμενη ενότητα τη λειτουργία του βρόχου ελέγχου συμβάντων που περιλαμβάνει την **ουρά των κλήσεων συναρτήσεων αναμονής** και τη λειτουργία του **σωρού** και της **στοίβας κλήσεων συναρτήσεων**.

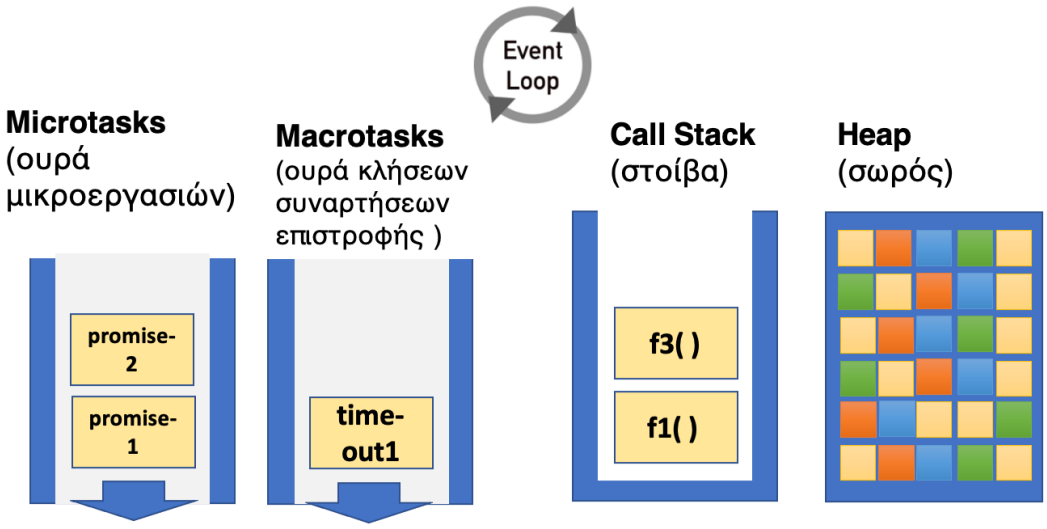
Το ερώτημα που προκύπτει είναι πώς η JavaScript χειρίζεται την ασύγχρονη κλήση συναρτήσεων που γίνεται μέσω του μηχανισμού των Promises.

Έστω, για παράδειγμα, ο παρακάτω κώδικας:

```
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');
```

Το ερώτημα είναι με τι σειρά θα εκτελεστούν οι εντολές και ποιο θα είναι το αποτέλεσμα που θα δούμε στην κονσόλα της JS.

Η νέα αρχιτεκτονική του βρόχου συμβάντων φαίνεται στη συνέχεια.



Εικόνα 10.4 *Δομές μνήμης κατά την εκτέλεση κώδικα JavaScript που περιλαμβάνει ασύγχρονα τμήματα κώδικα και χειρισμό υποσχέσεων.*

Οι Promises εισέρχονται σε μια **ουρά μικροεργασιών**, η οποία εξαντλείται σε κάθε κύκλο του βρόχου ελέγχου συμβάντων.

Κατά συνέπεια, η σειρά εκτέλεσης των Promises θα προηγηθεί της κλήσης συνάρτησης επιστροφής από τη μέθοδο setTimeout(), καθώς και της κλήσης του πρώτου animation frame, ενώ η εκτέλεση του τμήματος του προγράμματος που δεν περιλαμβάνει ασύγχρονη εκτέλεση κώδικα θα προηγηθεί όλων:

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

Θα πρέπει να διευκρινιστεί ότι σε κάθε κύκλο του βρόχου ασύγχρονης εκτέλεσης εργασιών εκτελούνται διάφορα βήματα, που σχετίζονται με την απεικόνιση πληροφορίας (resize, scroll, animation frames κ.λπ.) και εξαντλούνται τα microtasks (promises), ενώ σε διαφορετικούς φυλλομετρητές η σειρά εκτέλεσης των εργασιών αυτών πιθανόν να μεταβάλλεται.

10.5 Η διεπαφή fetch

Από την εποχή του Internet Explorer 5 το 1998 εισήχθη η δυνατότητα ασύγχρονων κλήσεων προς εξυπηρετητές του διαδικτύου από τον φυλλομετρητή με χρήση κλήσεων του **XMLHttpRequest (AJAX)**. Η σύνταξη της σχετικής κλήσης του αντικείμενου XMLHttpRequest ήταν αρκετά σύνθετη.

Η βιβλιοθήκη jQuery, αργότερα, προσέφερε πιο απλή σύνταξη jQuery.ajax(), jQuery.get().

Σήμερα η πιο διαδεδομένη διεπαφή που χρησιμοποιείται για ανάκτηση δεδομένων από εξυπηρετητές είναι η Fetch, που περιλαμβάνει κλήση της μεθόδου fetch(url) και έχει ενσωματωθεί στο αντικείμενο global window, για παράδειγμα, fetch("/file.json"), γεννάει ένα αίτημα GET της HTTP και διαχειρίζεται με ασύγχρονο τρόπο την υποδοχή της απάντησης (μήνυμα HTTP response).

Η μέθοδος fetch(url) επιστρέφει Promise, άρα το αποτέλεσμα μπορούμε να το χειριστούμε με αλυσίδα από κλήσεις μεθόδων then(). Θα πρέπει να σημειωθεί ότι εναλλακτικά θα μπορούσαμε να χειριστούμε το αντικείμενο Promise με χρήση του μηχανισμού async/await, όπως θα δούμε στο επόμενο κεφάλαιο.

10.5.1 Παράδειγμα χρήσης της fetch

Έστω ότι επιθυμούμε την ανάκτηση δεδομένων τύπου JSON από μια διεύθυνση url.

```
function loadJson(url) {
    return fetch(url)
        .then(response => {
            if (response.status == 200) {
                return response.json();
            } else {
                throw new Error(response.status);
            }
        })
        .then(data => console.log("απόκριση:", data))
}

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

Σ' αυτό το παράδειγμα, στον καταναλωτή then() ελέγχουμε την κατάσταση της απόκρισης μέσω του γνωρίσματος status της απόκρισης που έχουμε λάβει. Αν αυτή είναι 200 ('OK'), τότε καλούμε τη μέθοδο json() του αντικείμενου response. Αυτή η μέθοδος εκτελεί αντίστοιχη λειτουργία με της μεθόδου JSON.parse() που είδαμε σε προηγούμενη ενότητα (9.5.1), δηλαδή μετατρέπει μια συμβολοσειρά σε ένα αντικείμενο JavaScript. Συνεπώς, επιστρέφει αντικείμενο JavaScript που προκύπτει από τη συντακτική ανάλυση του σώματος της απόκρισης. Το αντικείμενο αυτό αποτελεί την απόκριση του promise και μπορεί να καταναλωθεί (π.χ. να εκτυπωθεί στην κονσόλα) από την επόμενη μέθοδο then() της αλυσίδας καταναλωτών.

Ιδιότητες του αντικείμενου που επιστρέφει η fetch

Το αντικείμενο που επιστρέφει η fetch ανήκει στην κλάση Response. Το αντικείμενο αυτό έχει ιδιότητες και μεθόδους που μας επιτρέπουν να χειριστούμε τα αποτελέσματα της κλήσης. Ήδη είδαμε τη μέθοδο json(). Εδώ θα δούμε μερικές ακόμη.

- **headers**. Μέσω του αντικείμενου αυτού έχουμε πρόσβαση στην κεφαλίδα της απόκρισης του HTTP (response header). Ένα παράδειγμα ανάκτησης των γνωρισμάτων Content-Type και Date:

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

- **status**. Ιδιότητα που παίρνει τιμή έναν ακέραιο αριθμό που ορίζει το HTTP response status. Τα 101, 204, 205 ή 304 σημαίνουν null body status, τα 200 μέχρι 299 ότι είναι OK (success), τα 301, 302, 303, 307 ή 308 redirect. Ένα παράδειγμα ανάκτησης του status:

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

- **statusText**. Ιδιότητα που παίρνει ως τιμή συμβολοσειρά με το status message της απόκρισης. Για παράδειγμα, έχει την τιμή "OK" αν είναι επιτυχής η απόκριση. Παράδειγμα:

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

- **url**. Ιδιότητα που έχει ως τιμή τη διεύθυνση URL του αντικείμενου. Παράδειγμα:

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

- **Body content**. Η απόκριση περιέχει ένα αντικείμενο body, στο οποίο έχουμε πρόσβαση μέσω διαφόρων μεθόδων: Η μέθοδος text() επιστρέφει το body ως συμβολοσειρά, η json() το επιστρέφει ως αντικείμενο JSON-parsed object, blob() ως Blob object (binary large object), η formData() ως FormData object κ.λπ.

Ανάκτηση διαθέσιμων δεδομένων από διεπαφές REST

Η fetch() μπορεί να έχει πολλές χρήσεις, μεταξύ άλλων να χρησιμοποιηθεί για να ανακτηθούν 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 σχετικά το Open Data Initiative (opendatainitiative.github.io).

Παραδείγματα τέτοιων διεπαφών είναι:

- https://restcountries.com/ (γεωγραφικά στοιχεία χωρών)
- http://www.7timer.info/doc.php?lang=en#api (πρόβλεψη καιρού)
- https://www.exchangerate-api.com/ (τιμές συναλλάγματος)
- https://covid19api.com/ (υγειονομική κρίση Covid-19)

10.5.2 Παράδειγμα: Οι καλοί γείτονες

Στην ενότητα αυτή θα δούμε ένα παράδειγμα χρήσης της διεπαφής fetch για την ανάκτηση δεδομένων από την προγραμματιστική διεπαφή https://restcountries.com/.

Η διεπαφή αυτή έχει διάφορα σημεία επαφής, μέσω των οποίων μπορούμε να ανακτήσουμε τα στοιχεία μιας χώρας (πληθυσμό, έκταση, πρωτεύουσα, γλώσσα, άλλες χώρες με τις οποίες συνορεύει κ.λπ.) είτε δίνοντας το όνομα της χώρας είτε δίνοντας έναν κωδικό δύο γραμμάτων της χώρας (π.χ. “GR” για την Ελλάδα) είτε κωδικό τριών γραμμάτων κ.λπ.

Ως άσκηση ζητείται να κάνουμε τα εξής:

- Ανακτούμε τα στοιχεία της χώρας από το αντίστοιχο σημείο επαφής (π.χ. για τη χώρα Italy).
- Αφού μελετήσουμε τη δομή των δεδομένων JSON που επιστρέφει η διεπαφή, εξετάζουμε τους γείτονες της χώρας.
- Αναζητούμε τη σχετική πληροφορία για τις χώρες που συνορεύει (Array: borders). Παρατηρούμε εδώ ότι για τη χώρα αυτή ο πίνακας περιέχει τις χώρες με τις οποίες συνορεύει η Ιταλία, κωδικοποιημένες με κωδικό τριών χαρακτήρων: borders = ['AUT', 'FRA', 'SMR', 'SVN', 'CHE', 'VAT'].
- Για καθεμία από τις χώρες αυτές θα πρέπει να βρούμε τα στοιχεία της με βάση τον κώδικά της και από αυτά να διαλέξουμε το όνομά της. Άρα, να ανακτήσουμε τα στοιχεία καθεμίας από τις χώρες αυτές μέσω του σημείου επαφής για κωδικούς τριών χαρακτήρων.

Για την επίλυση του προβλήματος αυτού θα πρέπει να αναζητήσουμε τρόπους εκτέλεσης παράλληλων ασύγχρονων κλήσεων σε διεπαφή δεδομένων.

Για τον σκοπό αυτό μπορούμε να χρησιμοποιήσουμε τη μέθοδο .all(set-of-promises) του αντικείμενου Promise.

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.

Θα πρέπει εδώ, για πληρότητα της παρουσίασης, να γίνει αναφορά και σε άλλες αντίστοιχες μεθόδους του αντικείμενου 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 ως 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 της άσκησης

Αρχικά ορίζουμε μια συνάρτηση loadCountryNameFromCode η οποία παίρνει ως όρισμα τον κωδικό τριών χαρακτήρων μιας χώρας και επιστρέφει μια Promise που περιλαμβάνει το όνομα της χώρας από τα δεδομένα της αντίστοιχης υπόσχεσης. Είναι σημαντικό ότι η συνάρτηση αυτή θα πρέπει να επιστρέφει υπόσχεση ώστε να δημιουργήσουμε έναν πίνακα υποσχέσεων τον οποίο να περάσουμε ως όρισμα στη μέθοδο .all().

```
const urlCountry = 'https://restcountries.com/v3.1/name/';
const urlCode = 'https://restcountries.com/v3.1/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"
        });
    });
}
```

Το κυρίως πρόγραμμά μας θα είναι η συνάρτηση findBorders(country):

```
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);    });
    }});
}
```

Η συνάρτηση αυτή αρχικά ανακτά μέσω fetch() τα στοιχεία της χώρας από το σημείο επαφής urlCountry.

Εφόσον ο κωδικός του μηνύματος απόκρισης είναι 200 (OK), κάνουμε συντακτική ανάλυση του αποτελέσματος στον πρώτο καταναλωτή της υπόσχεσης then() και εξάγουμε το αποτέλεσμα με τη μορφή αντικείμενου JavaScript.

Στη συνέχεια, ο επόμενος αλυσιδωτά καταναλωτής then() ελέγχει αν το πρώτο από τα στοιχεία που έχουν επιστραφεί (μπορεί πολλές χώρες να ικανοποιούν τη συμβολοσειρά αναζήτησης country) έχει γνώρισμα borders με πλήθος στοιχείων μεγαλύτερο του 0. Αυτό γιατί μπορεί μια χώρα να μην έχει γείτονες (π.χ. μια νησιωτική χώρα όπως η Ιαπωνία).

Στη συνέχεια γεμίζουμε τον πίνακα theCountries με τα αποτελέσματα της κλήσης της συνάρτησης loadCountryNameFromCode(). Η συνάρτηση αυτή, όπως αναφέρθηκε προηγουμένως, επιστρέφει ένα Promise.

Συνεπώς, διαδοχικά γεμίζουμε τον πίνακα αυτό με ασύγχρονες υποσχέσεις ανάκτησης των στοιχείων όλων των χωρών του πίνακα borders.

Τέλος, περνάμε τον πίνακα αυτό ως όρισμα στην all:

```
Promise.all(theCountries)
```

Το αποτέλεσμα της εντολής αυτής είναι, όπως αναφέρθηκε, υπόσχεση, άρα μπορούμε να την καταναλώσουμε με then().

Στον καταναλωτή αυτόν περνάμε το αποτέλεσμα, που είναι ένας πίνακας που περιέχει τα ονόματα των χωρών, δηλαδή το ζητούμενο της άσκησης.

10.6 Διαχείριση συμβάντων

Όπως έχουμε ήδη συζητήσει, η JavaScript είναι γλώσσα που ακολουθεί το παράδειγμα προγραμματισμού συμβάντων. Είδαμε σε προηγούμενη ενότητα τις διάφορες κατηγορίες συμβάντων που μπορεί να προκύψουν, από συσκευές, ανεξάρτητα συσκευής, συμβάντα της διεπαφής ή συμβάντα από APIs.

Επίσης, έχουμε συζητήσει τρόπους να ορίσουμε έναν χειριστή συμβάντων, είτε ως ιδιότητα του αντίστοιχου στοιχείου, element.onclick = χειριστής, είτε με κλήση της μεθόδου element.addEventListener(συμβάν, χειριστής).

Όταν προκύψει το συμβάν στο συγκεκριμένο στοιχείο το οποίο έχει οριστεί χειριστής του, η συνάρτηση-χειριστής καλείται.

10.6.1 Κλήση χειριστή συμβάντος

Κατά την κλήση του χειριστή συμβάντος περνάμε ως όρισμα το αντικείμενο Event. Το αντικείμενο αυτό έχει τις εξής ιδιότητες:

- event.type, τον τύπο του συμβάντος.
- event.target, το αντικείμενο στο οποίο έχει γίνει το συμβάν.
- event.currentTarget, για συμβάντα τα οποία μεταδίδονται, όπως θα δούμε στη συνέχεια, η ιδιότητα αυτή περιγράφει το αντικείμενο στο οποίο έχει συνδεθεί ο χειριστής του συμβάντος, που μπορεί να είναι διαφορετικό από το event.target.
- event.timeStamp, η χρονική στιγμή κατά την οποία έγινε το συμβάν, μετρημένη σε ms από την αρχή φορτώματος της ιστοσελίδας.
- event.isTrusted, ένδειξη αν το συμβάν προέκυψε από τον φυλλομετρητή (true) ή από τον κώδικα JavaScript.

Επιπροσθέτως, το αντικείμενο Event μπορεί να έχει και άλλες ιδιότητες ανάλογα με τον τύπο του. Για παράδειγμα, το συμβάν click ή εν γένει συμβάντα από το ποντίκι ή, πιο γενικά, από τη δεικτική συσκευή έχουν ως ιδιότητες τα event.clientX και event.clientY, με τις συντεταγμένες του σημείου στο παράθυρο. Αν το συμβάν αφορά το πληκτρολόγιο, έχει ως ιδιότητα τον κωδικό του χαρακτήρα που πληκτρολογήθηκε ως event.keyCode κ.λπ.

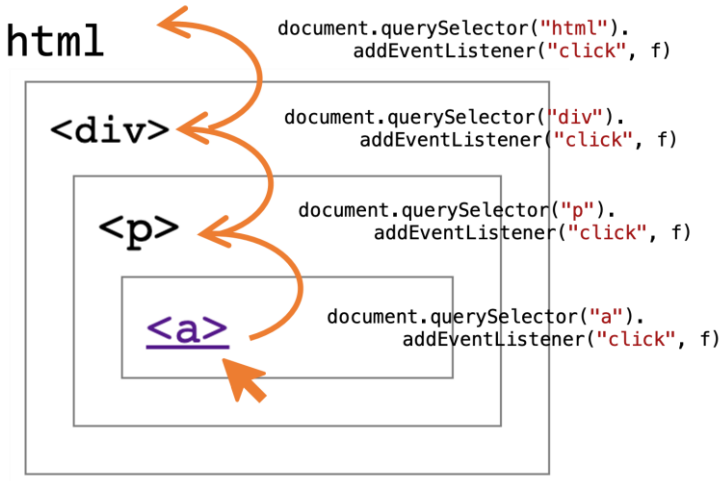
Θα πρέπει να λάβουμε υπόψη μας το γεγονός ότι ένας χειριστής ορίζεται ως τιμή στην ιδιότητα ενός αντικείμενου, συνεπώς ο τελεστής this σε έναν χειριστή αναφέρεται στο αντικείμενο στο οποίο αυτός έχει οριστεί.

10.6.2 Ο μηχανισμός φυσαλίδας

Ένα ενδιαφέρον θέμα είναι η διαχείριση συμβάντων που σχετίζονται με στοιχεία του DOM τα οποία ανήκουν σε μια ιεραρχία. Αν ο χρήστης επιλέξει ένα στοιχείο, επιλέγει και το στοιχείο στο οποίο αυτό ανήκει και εκείνο στο οποίο ανήκει ο πατέρας του κ.ο.κ. Το ερώτημα που γεννάται είναι, αν έχουν οριστεί χειριστές του συμβάντος στα ανώτερα επίπεδα της ιεραρχίας, αυτοί θα αντιδράσουν επίσης στο συμβάν;

Η JavaScript έχει έναν σύνθετο μηχανισμό διάδοσης συμβάντων (ο **μηχανισμός φυσαλίδας**, event bubbling), που περιγράφεται στη συνέχεια, για να αντιμετωπίσει το θέμα αυτό.

Ας υποθέσουμε το παράδειγμα στην **Εικόνα *10.5***:



Εικόνα 10.5 *Παράδειγμα ιεραρχίας στοιχείων και αντίστοιχων χειριστών συμβάντων.*

Ας υποθέσουμε ότι σε ένα έγγραφο HTML έχουμε την ιεραρχία που φαίνεται και στην ***Εικόνα 10.5***:

```
<html>
    <body>
        <div>
            <p>
                <a> </a>
            </p>
        </div>
    </body>
</html>
```

Ας υποθέσουμε επίσης ότι έχουν οριστεί οι παρακάτω χειριστές για το έγγραφο αυτό:

```
function f(event){
    console.log(`φυσαλίδα συμβάντος: ${event.target} this=
${this.tagName}`);
}

document.querySelector("a").addEventListener("click", f)
document.querySelector("p").addEventListener("click", f)
document.querySelector("div").addEventListener("click", f)
document.querySelector("html").addEventListener("click", f)
```

Για τα τέσσερα αυτά στοιχεία της ιεραρχίας έχει οριστεί η συνάρτηση f(), η οποία καλείται αν ο χρήστης κάνει κλικ με τη δεικτική συσκευή στο αντίστοιχο στοιχείο. Επίσης, να σημειώσουμε ότι έχουμε ορίσει ο *χειριστής* να μας γνωρίζει ότι έχει κληθεί τυπώνοντας σχετικό μήνυμα στο οποίο μας δίνει την τιμή της ιδιότητας event.target και της ιδιότητας this.tagName.

Ας εξετάσουμε στη συνέχεια τι θα συμβεί αν ο χρήστης επιλέξει το στοιχείο **<a>**.

Στην περίπτωση αυτή στην κονσόλα θα δούμε τα εξής μηνύματα:

```
> φυσαλίδα συμβάντος: http://localhost/event2.html# this= A
> φυσαλίδα συμβάντος: http://localhost/event2.html# this= P
> φυσαλίδα συμβάντος: http://localhost/event2.html# this= DIV
> φυσαλίδα συμβάντος: http://localhost/event2.html# this= HTML
```

Παρατηρούμε ότι το συμβάν του χρήστη έχει προκαλέσει τέσσερις διαδοχικές κλήσεις του χειριστή, που έχουν προκύψει από τα στοιχεία **<a>**, **<p>**, ** **, **<html>** με αυτή τη σειρά, καθώς το συμβάν διαδόθηκε στην ιεραρχία μέχρι το ανώτερο επίπεδο και ενεργοποιήθηκαν όλοι οι χειριστές της ιεραρχίας αυτής. Τα μηνύματα αυτά έκαναν όλα αναφορά στην ίδια τιμή του event.target (τον υπερσύνδεσμο που επέλεξε ο χρήστης), αλλά σε διαφορετική κάθε φορά τιμή του this.tagName, που σχετίζεται με στοιχείο στο οποίο έχει επισυναφθεί ο

αντίστοιχος χειριστής.

Αν, αντίθετα, κάνουμε κλικ στην περιοχή του στοιχείου <**div**> έξω από τα υπόλοιπα εσωτερικά στοιχεία, θα πάρουμε μόνο τα εξής δύο μηνύματα:

```
> φυσαλίδα συμβάντος: [object HTMLDivElement] this= DIV
> φυσαλίδα συμβάντος: [object HTMLDivElement] this= HTML
```

Συνέπεια του φαινομένου αυτού είναι ότι μπορούμε να ορίσουμε έναν χειριστή για έναν υποδοχέα που περιέχει πολλά στοιχεία και να διαγνώσουμε ποιο συγκεκριμένο στοιχείο είναι το αντικείμενο που έχει επιλέξει ο χρήστης ελέγχοντας την τιμή του event.target.

Θα πρέπει να σημειώσουμε ότι κάποια συμβάντα δεν διαδίδονται με τον μηχανισμό φυσαλίδας, αυτά περιλαμβάνουν τα συμβάντα “focus”, “blur” και “scroll”.

Επίσης, το φαινόμενο διάδοσης ενός συμβάντος είναι ακόμη πιο σύνθετο από τον μηχανισμό φυσαλίδας που ήδη περιγράφηκε. Η πλήρης περιγραφή του μηχανισμού αναλύεται στη συνέχεια.

Φάση σύλληψης συμβάντος

Όταν γίνει ένα συμβάν, ξεκινάει ένας έλεγχος για χειριστές του συγκεκριμένου συμβάντος από τη ρίζα του δένδρου προς τα κάτω, μέχρι να φτάσουμε στο κατώτερο επίπεδο στο οποίο υπάρχει χειριστής. Η φάση αυτή περιγράφεται ως **φάση σύλληψης του συμβάντος** (event capturing), προηγείται της κλήσης του χειριστή συμβάντος και δεν προκαλεί κλήση των χειριστών που θα βρεθούν στην πορεία αυτή.

Όταν ο έλεγχος φτάσει στο κατώτερο επίπεδο που βρίσκεται αντίστοιχος χειριστής, τότε αυτός ενεργοποιείται και αρχίζει η δεύτερη φάση της διάδοσης προς τα πάνω με τον μηχανισμό φυσαλίδας που ήδη αναφέρθηκε.

Το ερώτημα είναι ποιος ο λόγος ύπαρξης της φάσης σύλληψης του συμβάντος. Η χρήση αυτής της φάσης του μηχανισμού είναι περιορισμένη, αλλά μπορεί να φανεί χρήσιμη για να εμποδιστεί ένα συμβάν να διαδοθεί προς τα κάτω. Ένα παράδειγμα είναι, όταν θέλουμε να σύρουμε ένα αντικείμενο στην επιφάνεια εργασίας, δεν επιθυμούμε να προκληθούν συμβάντα που σχετίζονται με το πέρασμα της συσκευής από υποκείμενα στοιχεία της διεπαφής. Πώς όμως θα οριστεί η συμπεριφορά αυτή; Αυτό γίνεται κατά τον ορισμό του χειριστή που θα αναλάβει αυτό το έργο. Ο ορισμός του χειριστή αυτού έχει την εξής σύνταξη:

```
element.addEventListener(συμβάν, χειριστής, {capture: true})
```

Στον χειριστή που έχει οριστεί με τη σύνταξη αυτή μπορεί να κληθεί η μέθοδος κατάργησης του συμβάντος event.stopPropagation() η οποία σταματάει τη διάδοση του συμβάντος προς τα κάτω και συνεπώς την κλήση του χειριστή.

Θα πρέπει να σημειωθεί ότι η μέθοδος αυτή σταματάει τη διάδοση ενός συμβάντος και προς τα πάνω αν βρεθεί σε έναν χειριστή του συμβάντος κατά τη φάση της διάδοσης με τον μηχανισμό φυσαλίδας.

Αποτροπή προκαθορισμένης συμπεριφοράς

Τα συμβάντα προκαλούν προκαθορισμένη συμπεριφορά σε μια ιστοσελίδα.

Για παράδειγμα, όταν ο χρήστης επιλέξει έναν υπερσύνδεσμο ο φυλλομετρητής ξεκινάει κλήση για φόρτωμα της ιστοσελίδας στόχου, όταν ο χρήστης πληκτρολογήσει έναν 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.

Ένας χειριστής ενός τέτοιου συμβάντος μπορεί να αποτρέψει τον φυλλομετρητή από το να ξεκινήσει την προκαθορισμένη ενέργεια με κλήση της μεθόδου event.preventDefault().

Μια συνηθισμένη χρήση της μεθόδου αυτής είναι σε μια φόρμα, αν ορίσουμε έναν χειριστή του συμβάντος “submit” με σκοπό να κάνουμε έλεγχο εγκυρότητας των δεδομένων που έχει εισαγάγει ο χρήστης. Αν ο έλεγχος αυτός διαπιστώσει σφάλματα στα στοιχεία της φόρμας, τότε καλείται η μέθοδος event.preventDefault(), ώστε να αποτραπεί η υποβολή της φόρμας και να δοθεί η ευκαιρία μέσω κατάλληλων υποδείξεων στον χρήστη να διορθώσει τα στοιχεία πριν υποβληθούν. Για τη χρήση της preventDefault() μπορείτε να ανατρέξετε και στο παράδειγμα της ενότητας **6.7.1 Έλεγχος εγκυρότητας με την Bootstrap.**

10.7 Ερωτήσεις αυτοαξιολόγησης

1. Η JavaScript έχει διάφορους μηχανισμούς για ασύγχρονη εκτέλεση ενός τμήματος του προγράμματος, σημειώστε όλους όσους ισχύουν.
    1. Ο μηχανισμός κλήσεων συναρτήσεων επιστροφής (call back functions).
    2. Ο μηχανισμός της στοίβας κλήσεων (call stack).
    3. Η μέθοδος fetch.
    4. Ο μηχανισμός υποσχέσεων Promises.
2. Η fetch με ποιον μηχανισμό ασύγχρονης εκτέλεσης προγράμματος υλοποιείται;
    1. Με τον μηχανισμό Promise.
    2. Με τον μηχανισμό κλήσης συνάρτησης επιστροφής callback.
    3. Και με τον μηχανισμό Promise και με τον μηχανισμό callback.
3. Ποιος ο ρόλος της στοίβας (call stack) κατά την εκτέλεση ενός προγράμματος JavaScript;
    1. Είναι ένας μηχανισμός για τον έλεγχο της κλήσης και εκτέλεσης συναρτήσεων.
    2. Είναι ένας χώρος αποθήκευσης δεδομένων των συναρτήσεων.
    3. Είναι ένας χώρος ελέγχου εκτέλεσης συναρτήσεων επιστροφής (callbacks).
4. Η JavaScript είναι πολυ-νηματική (multi-threaded) γλώσσα προγραμματισμού.
    – Σωστό/Λάθος
5. Οι συναρτήσεις που εκτελούνται με ασύγχρονο τρόπο τίθενται:
    1. στην ουρά κλήσεων συναρτήσεων επιστροφής (callback queue).
    2. στη στοίβα κλήσεων (call stack).
    3. στον σωρό (heap).
6. Προβλέψτε το αποτέλεσμα εκτέλεσης του παρακάτω κώδικα:

```
console.log('1');
setTimeout(() => {console.log('2')}, 10);
setTimeout(() => {console.log('3')}, 0);
console.log('4');
```

    1. σειρά 1, 2, 3, 4
    2. σειρά 1, 4, 2, 3
    3. σειρά 1, 4, 3, 2
    4. σειρά 1, 3, 2, 4
7. Έστω ο παρακάτω κώδικας:

```
let counter = 0;
const i = setInterval(() => {console.log(++counter)}, 100);
setTimeout(clearInterval, 300, i);
```

    Πόσες διαφορετικές τιμές θα τυπωθούν κατά την εκτέλεσή του;
    Απάντηση: _____
8. Ο μηχανισμός Promise έχει εισαχθεί στην JavaScript από το 2015 (έκδοση ES6).
    – Σωστό/Λάθος
9. Καταναλωτές ενός Promise μπορεί να είναι οι εξής μέθοδοι:
    1. try()
    2. catch()
    3. then()
    4. finally()
10. Το όρισμα φ της κλάσης Promise(φ) είναι:
    1. μια συνάρτηση που εκτελεί την ασύγχρονη λειτουργία.
    2. δύο συναρτήσεις που η μια καλείται σε περίπτωση επιτυχούς ολοκλήρωσης και η άλλη σε περίπτωση αποτυχίας.
    3. δύο μεταβλητές status και result οι οποίες στην αρχή έχουν την τιμή “pending” και “undifined”.
11. Η ουρά μικροεργασιών (microtask queue) διαχειρίζεται τους καταναλωτές υποσχέσεων που βρίσκονται σε αναμονή.
    – Σωστό/Λάθος
12. Ποια η διαφορά μεταξύ ουράς μικροεργασιών και ουράς μακροεργασιών; Σημειώστε όλα όσα ισχύουν.

1. Η ουρά των μικροεργασιών σε κάθε κύκλο του βρόχου ασύγχρονης εκτέλεσης αδειάζει, ενώ από την ουρά μακροεργασιών εκτελείται το πρώτο στοιχείο της ουράς.
2. Η ουρά μικροεργασιών είναι πιο μικρή από την ουρά μακροεργασιών.
3. Η ουρά μακροεργασιών τροφοδοτείται από συναρτήσεις και τμήματα του κώδικα που προκύπτουν από κλήσεις επιστροφής (call back), ενώ η ουρά μικροεργασιών από καταναλωτές υποσχέσεων (promise).

13. Ποιο το αποτέλεσμα του παρακάτω προγράμματος;

```
new Promise((res, rej) => {
    setTimeout(() => res(10), 2000);
    setTimeout(() => res(20), 3000);
    setTimeout(() => rej(30), 1000);
    setTimeout(() => rej(50), 4000);
})
.then((r) => {console.log("result:"+r);})
.catch((e) => {console.log("error:"+e);});
```

1. Διαδοχικά θα τυπωθούν error:30, result:10, result:20, error:50
2. Διαδοχικά θα τυπωθούν result:10, result:20
3. Θα τυπωθεί result:10
4. Θα τυπωθεί error:30
5. Διαδοχικά θα τυπωθούν error:30, error:50

14. Ποιο το αποτέλεσμα εκτέλεσης του προγράμματος;

```
console.log('αρχή προγράμματος');
setTimeout(() => {console.log('setTimeout');}, 0);
const p = new Promise((resolve, reject) => resolve());
p.then(setTimeout(() => {console.log('promise1');}, 100));
p.then(setTimeout(() => {console.log('promise2');}, 100));
console.log('τέλος προγράμματος');
```

1. Διαδοχικά θα τυπωθούν: ‘αρχή προγράμματος’, ‘setTimeout’, ‘promise1’, ‘promise2’, ‘τέλος προγράμματος’.
2. Διαδοχικά θα τυπωθούν: ‘αρχή προγράμματος’, ‘τέλος προγράμματος’, ‘setTimeout’, ‘promise1’, ‘promise2’.
3. Διαδοχικά θα τυπωθούν: ‘αρχή προγράμματος’, ‘τέλος προγράμματος’, ‘promise1’, ‘promise2’, ‘setTimeout’.

15. Τι επιστρέφει η μέθοδος then(φ,ψ) του αντικείμενου Promise;
1. Το αποτέλεσμα της επεξεργασίας του αποτελέσματος της υπόσχεσης.
2. Ένα αντικείμενο τύπου Promise.
3. true αν είναι επιτυχή τα αποτελέσματα της υπόσχεσης ή false αν είναι ανεπιτυχή.

16. Ποια η απόκριση του παρακάτω κώδικα;

```
p = new Promise((res, rej) => {
if (Math.random() > 0.5) res(1);
else rej(0);});
p.then(
(r) => {console.log('r1:' + r++)},
(e) => {console.log('e1:' + e++)});
p.then(
(r) => {console.log('r2:' + r++)},
(e) => {console.log('e2:' + e++)});
```

1. Διαδοχικά r1:1, e1:0, r2:2, e2:1
2. Διαδοχικά r1:1, r2:1, e1:0, e2:0
3. Είτε r1:1, r2:1 είτε e1:0, e2:0
4. Είτε r1:1, r2:2 είτε e1:0, e2:1

17. Ποια η απόκριση του παρακάτω κώδικα;

```
p = new Promise((resolve, reject) => {
if (Math.random() > 0.5) resolve(1);
else reject(0);
});
p.then((r) => {
if (r) {
    console.log('r1:' + r++);
    return r;
}
})
.then((r) => {
    console.log('r2:' + r++);
})
.catch((e) => {
    console.log('e1:' + e++);
});
```

1. Διαδοχικά r1:1, e1:0, r2:2
2. Διαδοχικά r1:1, r2:2, e1:0
3. Είτε r1:1, r2:1 είτε e1:0

18. Η δυνατότητα ασύγχρονης κλήσης μια ιστοσελίδας στον εξυπηρετητή εισήχθη για πρώτη φορά το 2015 με την έκδοση ES6 της JavaScript.
    – Σωστό/Λάθος
19. Η μέθοδος fetch() ανήκει στο αντικείμενο…
    Απάντηση: _____
20. Η μέθοδος fetch() επιστρέφει ένα αντικείμενο:
    1. JSON
    2. HTTP response
    3. Promise
21. Στον παρακάτω κώδικα ποια η απόκριση σε περίπτωση επιτυχούς κλήσης της fetch;
    fetch("/api.json").then((r) => ({console.log(r.statusText)}))
    Απάντηση: _____
22. Αν η μέθοδος fetch(url) επιστρέψει στο body δεδομένα σε μορφή JSON, η επεξεργασία τους με την ασύγχρονη μέθοδο .json() είναι αντίστοιχη με ποια από τις μεθόδους που έχουμε δει στη σύγχρονη επεξεργασία δεδομένων;
    1. JSON.parse(body)
    2. JSON.stringify(body)
    3. JSON.process(body)
23. Ποιο το αποτέλεσμα του παρακάτω κώδικα;

```
const thePromises = [];
for (let i = 0; i < 3; i++) {
thePromises.push(
    new Promise((resolve, reject) => {
    const res = Math.floor(Math.random() * 10 + 1);
    if (Math.random() > 0.1) {
        setTimeout(() => resolve('OK'), res * 10);
    } else {
        reject('error...');
    }
    })
);
}
Promise.all(thePromises)
.then((res) => {console.log(res)})
.catch((er) => {console.log(er)});
```

1. Είτε [‘OK’, ‘OK’, ‘OK’] είτε [‘error…’, ‘error…’, ‘error…’]

2. Ένα array 3 στοιχείων που περιλαμβάνει στοιχεία είτε ‘OK’ είτε ‘error…’
3. Είτε [‘OK’, ‘OK’, ‘OK’] είτε ‘error…’

24. Η διαφορά της Promise.any() από την Promise.race() είναι:
    1. Η Promise.any() επιστρέφει μια τυχαία υπόσχεση που έχει ολοκληρωθεί, ενώ η Promise.race() την πρώτη που έχει ολοκληρωθεί.
    2. Η Promise.any() επιστρέφει την πρώτη υπόσχεση που έχει ολοκληρωθεί επιτυχώς, ενώ η Promise.race() την πρώτη που έχει ολοκληρωθεί ανεξάρτητα αποτελέσματος.
    3. Η Promise.any() επιστρέφει στοιχεία για όλες τις υποσχέσεις, ενώ η Promise.race() μόνο για αυτή που τερμάτισε πρώτη.
25. Ποια είναι η μέθοδος του αντικείμενου Promise η οποία δέχεται ένα σύνολο από υποσχέσεις και επιστρέφει μια υπόσχεση με όλα τα αποτελέσματα, είτε αυτά είναι επιτυχή είτε όχι; Promise([p1,p2,...]). _____()
    Απάντηση: _____

10.8 Βιβλιογραφία και Αναφορές

Το πρότυπο EcmaScript (ECMA-262) συντηρείται από τον οργανισμό ecma.

Η βιβλιογραφία του κεφαλαίου αυτού μόνο εν μέρει καλύπτεται από τη βιβλιογραφία των προηγούμενων κεφαλαίων.

Οι πηγές για εκμάθηση της JavaScript στο διαδίκτυο καλύπτουν και τα θέματα αυτού του κεφαλαίου. Η MDN είναι μια καλή πηγή για εισαγωγικά και προχωρημένα μαθήματα, όπως και για την HTML και CSS. Επίσης, η w3schools περιέχει μαθήματα, ενώ η JavaScript.info περιλαμβάνει μια πλήρη σειρά μαθημάτων για την JavaScript με παραδείγματα, ξεκινώντας από τη γλώσσα και προχωρώντας στη διεπαφή της με τον φυλλομετρητή.

Τα θέματα του κεφαλαίου αυτού καλύπτονται επίσης από τα εγχειρίδια της JavaScript. Στις πηγές αυτές συχνά περιλαμβάνονται και κεφάλαια που αφορούν τη λειτουργία της γλώσσας στο περιβάλλον node.js. Αυτή την προσέγγιση ακολουθεί το βιβλίο του Λιακέα (2021). Από τη διεθνή βιβλιογραφία παρόμοια προσέγγιση ακολουθούν πολλά εγχειρίδια, όπως του Flanagan (2020), του Frisbie (2020) κ.λπ.

Επιπλέον, οι συγγραφείς αυτού του βιβλίου έχουν δημιουργήσει *ανοιχτά διαδικτυακά μαθήματα* στην πλατφόρμα mathesis, υλικό από τα οποία έχει χρησιμοποιηθεί και σε αυτό το σύγγραμμα. Για αυτό το κεφάλαιο το σχετικό μάθημα είναι οι διαλέξεις του μαθήματος «Προχωρημένα θέματα ανάπτυξης ιστοσελίδων».

Α. Ξενόγλωσσες

Flanagan, D. (2020). *JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language* (7th ed.). O'Reilly Media, Inc.

Frisbie, M. (2020). *Professional JavaScript for Web Developers*. Wiley.

Β. Ελληνόγλωσσες

Αβούρης, Ν., & Σιντόρης, Χ. (2020). Προχωρημένα θέματα ανάπτυξης ιστοσελίδων. Ανοικτό διαδικτυακό μάθημα, https://mathesis.cup.gr

Λιακέας, Γ. (2021). *Η γλώσσα JavaScript* (3η έκδ.). Κλειδάριθμος. Κωδικός βιβλίου στον Εύδοξο: 102070465.