Προγραμματισμός Διαδικτύου

Νικόλαος Αβούρης, Χρήστος Σιντόρης

2021-03-27

Περιεχόμενα

Κατάλογος πινάκων

Κατάλογος σχημάτων

1 Εισαγωγή στην JavaScript

Στο κεφάλαιο αυτό ξεκινάμε την εισαγωγή στη γλώσσα προγραμματισμού **JavaScript**, που αποτελεί μια βασική τεχνολογία του διαδικτυακού προγραμματισμού και μια από τις γλώσσες προγραμματισμού που έχουν ευρύτατη χρήση όχι μόνο στον φυλλομετρητή αλλά και έξω από αυτόν, για προγραμματισμό του εξυπηρετητή (περιβάλλον node.js), καθώς και για ανάπτυξη μη διαδικτυακών διαδραστικών εφαρμογών στην επιφάνεια εργασίας του υπολογιστή μας (περιβάλλον electron.js).

Η γλώσσα JavaScript είναι αντικείμενο αυτού και των επόμενων κεφαλαίων. Θα δούμε αρχικά τη χρήση της στον φυλλομετρητή και αργότερα στον εξυπηρετητή, στο περιβάλλον *node.js*.

1.1 Εισαγωγή - ιστορικό σημείωμα

Στην πρώτη αυτή ενότητα θα δώσουμε μία γενική εισαγωγή με κύρια την ιστορική διάσταση και τον ρόλο της JavaScript. Η JavaScript είναι μία γλώσσα προγραμματισμού υψηλού επιπέδου, όπως η Python, η Java, κλπ. Ακολουθεί ένα πρότυπο, το πρότυπο ECMAScript. Το επίσημο όνομά της είναι **ECMAScript** αλλά έχει καθιερωθεί να χρησιμοποιούμε το εμπορικό όνομα που αρχικά της δόθηκε, JavaScript.

Η ιστορία της αρχίζει το 1995 από τον μηχανικό της εταιρείας Netscape, Brendan Eich. Ο Netscape ήταν την εποχή εκείνη ο πιο δημοφιλής φυλλομετρητής, στο πλαίσιο του αναπτύχθηκε η πρώιμη έκδοση της JavaScript ως γλώσσα προγραμματισμού μιας ιστοσελίδας, αφού είχε προκύψει η ανάγκη οι ιστοσελίδες να γίνουν πιο διαδραστικές.

Η JavaScript μαζί με την HTML και τη CSS που είδαμε σε προηγούμενα κεφάλαια, είναι οι βασικές τεχνολογίες για να αναπτύσσουμε εφαρμογές στον παγκόσμιο ιστό, στο διαδίκτυο. Σύντομα θα δούμε πώς συνδέεται η JavaScript με τις προηγούμενες δύο τεχνολογίες που έχουμε ήδη δεί.

Όταν κατεβάζουμε ένα αρχείο HTML που συνοδεύεται από ένα φύλλο μορφοποίησης CSS, είτε σε ξεχωριστό αρχείο, είτε στο ίδιο το αρχείο, μπορεί μέσα στο αρχείο ή σε ένα συνοδευτικό διασυνδεδεμένο αρχείο να υπάρχει κώδικας JavaScript. Ο κώδικας αυτός λοιπόν πρέπει να εκτελεστεί. Αυτό γίνεται μέσα στον ίδιο τον φυλλομετρητή ο οποίος διαθέτει μία ``μηχανή'' που μπορεί να κάνει συντακτική ανάλυση και μεταγλώττιση της JavaScript σε μία γλώσσα που εκτελείται από τον υπολογιστή (γλώσσα μηχανής).

Επίσης θα πρέπει να σημειώσουμε ότι η εξαιρετική δημοφιλία αυτής της τεχνολογίας έχει οδηγήσει ώστε αυτή να έχει αποκτήσει και άλλες χρήσεις εκτός από γλώσσα που χρησιμοποιείται στον φυλλομετρητή στις εφαρμογές του διαδικτύου.

Για παράδειγμα με το runtime περιβάλλον `node.js` η γλώσσα αυτή χρησιμοποιείται σαν μία παραδοσιακή γλώσσα, στην ουσία για εφαρμογές στον εξυπηρετητή, να δρομολογεί αιτήματα για παροχή ιστοσελίδων, να συνδέει τις εφαρμογές αυτές με μία βάση δεδομένων, το σύστημα αρχείων κλπ.

1.1.1 Κύρια χαρακτηριστικά της JavaScript

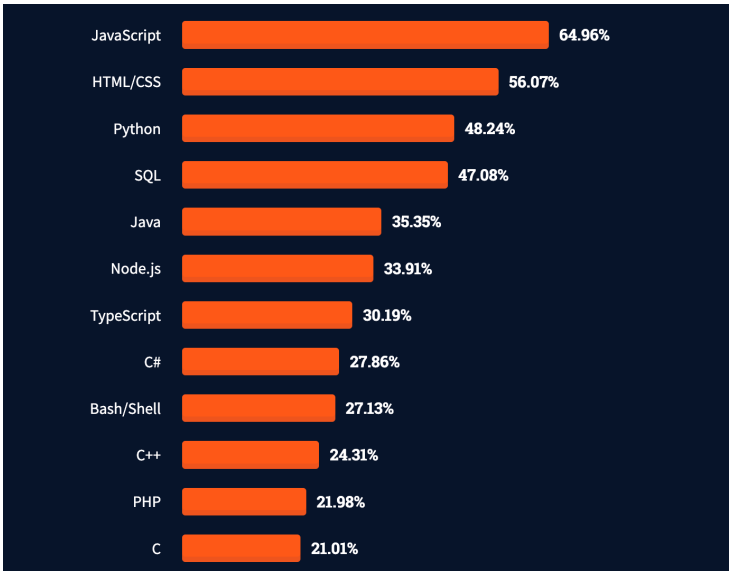
Ας δούμε στη συνέχει τα κύρια χαρακτηριστικά της JavaScript.

- Χαρακτηρίζεται καταρχάς ως δυναμική γλώσσα με **ασθενή διαχείριση τύπων δεδομένων** (weakly typed language). Η JavaScript δεν είναι πολύ αυστηρή ως προς τους τύπους δεδομένων, και συχνά παραβλέπει ασυμβατότητα τύπων σε εκφράσεις κάνοντας μετατροπές τύπων. Όπως θα δούμε η έκφραση `5 + "10"` στις περισσότερες γλώσσες προγραμματισμού θα οδηγήσει σε σφάλμα, αφού δεν επιτρέπεται η πρόσθεση ασύμβατων τύπων δεδομένων (στην Python θα παράγει TypeError). Στην JavaScript θα δώσει το αποτέλεσμα `"510"`, που έχει παραχθεί από την μετατροπή του αριθμού 5 στη συμβολοσειρά ``5'' και στη συνέχεια στην συνένωση των δύο συμβολοσειρών.
- Συχνά αυτό το χαρακτηριστικό της JavaScript είναι αιτία κριτικής στη γλώσσα, αφού μπορεί να προκαλέσει ανεπιθύμητατα σφάλματα που είναι δύσκολο να διαγνωστούν, και μάλιστα έχουν δημιουργηθεί διάλεκτοι της γλώσσας, όπως η **TypeScript** που χειρίζονται τα δεδομένα με πιο αυστηρό τρόπο.
- Η JavaScript είναι αντικειμενοστρεφής γλώσσα ως προς την αρχιτεκτονική της, τα περισσότερα δεδομένα της είναι αντικείμενα, όμως ακολουθεί ένα ιδιαίτερο μηχανισμό κληρονομικότητας που στηρίζεται σε ``πρωτότυπα'', όπως θα περιγραφεί σε επόμενο κεφάλαιο.
- Επίσης υποστηρίζει διαφορετικά προγραμματιστικά παραδείγματα, όπως τον προγραμματισμό με συμβάντα (**event-based programming**), συναρτησιασκό προγραμματισμό (**functional programming**), επίσης είναι αντικειμενοστρεφής και διαθέτει δομές για να προγραμματίσουμε με αντικείμενα (**object-oriented programming**).
- Η JS διαθέτει προγραμματιστικές διεπαφές (APIs), για διαχείριση κειμένων, διαχείριση πινάκων, ημερομηνιών, τυπικές εκφράσεις (regular expressions) καθώς και, κάτι που μας ενδιαφέρει ιδιαίτερα, διαθέτει προγραμματιστική διεπαφή προς το DOM (Document Object Model), δηλαδή προς αντικείμενα που αντιστοιχούν στα στοιχεία ενός HTML αρχείου.
- Δεν περιλαμβάνει διεπαφή για αλληλεπίδραση απευθείας με το χρήστη, δηλαδή δεν υπάρχει εντολή `print()`, επίσης δεν διαθέτει διεπαφή για δικτύωση, για μόνιμη αποθήκευση στο σύστημα αρχείων ή σε βάση δεδομένων ή γραφική διεπαφή.

1.1.2 Χρήση της JavaScript

Για να έχουμε μια ιδέα του πόσο δημοφιλής είναι η JavaScript, δεν έχουμε παρά να εξετάσουμε ένα πρόσφατο ετήσιο developer survey της δημοφιλούς ιστσελίδας StackOverflow, που έχει ευρύτατη χρήση στην κοινότητα

των προγραμματιστών. Στο ερώτημα για την πιο δημοφιλή τεχνολογία οι απαντήσεις που δόθηκαν κατά την ετήσια επισκόπηση του 2021, φαίνονται στην εικόνα1.1.



Σχήμα 1.1: Δημοφιλία της JavaScript σύμφωνα με την ετήσια επισκόπηση του StackOverflow.

Μάλιστα σύμφωνα με την ετήσια αυτή έρευνα, η γλώσσα JavaScript παραμένει στην θέση της πιο δημοφιλούς τεχνολογίας για 9η συνεχή χρονιά, με βάση τις απαντήσεις των μελών του StackOverflow. Ενώ θα πρέπει ακόμη να παρατηρηθεί ότι το περιβάλλον Node.js, που στην ουσία αφορά προγραμματιστικό περιβάλλον στην ίδια γλώσσα προγραμματισμού, και η TypeScript που αποτελεί διάλεκτο της JavaScript, βρίσκονται επίσης στην πρώτη δεκάδα των πιο δημοφιλών τεχνολογιών.

Αν εξετάσουμε ιστορικά την εξέλιξη της JavaScript θα παρατηρήσουμε ότι αρχικά η γλώσσα αυτή δημιουργήθηκε για να υποστηρίζει και να επιτρέπει διαδραστικότητα σε έγγραφα HTML (ιστοσελίδες). Παραδείγματος χάρη χρησιμοποιούσαμε τα script της JavaScript για να ελέγχουμε την εγκυρότητα δεδομένων που δίνει ο χρήστης σε μία φόρμα, να υπάρχει διαδραστικότητα σε συμβάντα που προκαλεί ο χρήστης, όταν χειρίζεται το ποντίκι, το πληκτρολόγιο, την οθόνη αφής κλπ.

Σήμερα εκτός από αυτή τη χρήση που παραμένει πολύ σημαντική, υπάρχει και μία άλλη χρήση που θα τη δούμε περισσότερο στη συνέχεια. Από το τέλος της δεκαετίας του '90 και μετά, με την έλευση της AJAX (``*Asynchronous JavaScript and XML*'') που επέτρεπε μια ιστοσελίδα να κάνει ασύγχρονες κλήσεις προς τον εξυπηρετητή, άρχισε η ανάπτυξη νέου τύπου διαδικτυακών εφαρμογών, που ονομάζονται εφαρμογές μοναδικής ιστοσελίδας (single page applications, SPA), οι οποίες στέλνουν και λαμβάνουν δεδομένα από τον εξυπηρετητή χωρίς να απαιτείται ξαναφόρτωμα της σελίδας. Τέτοιες εφαρμογές συναντάμε όλο και πιο συχνά στο διαδίκτυο, είναι εφαρμογές κοινωνικών δικτύων, εφαρμογές χαρτογραφικού περιεχομένου, εφαρμογές σχεδίασης, διαχείρισης κειμένων και φύλλων εργασίας, που μοιάζουν όλο και περισσότερο με διαδραστικές εφαρμογές που έχουμε συνηθίσει στην επιφάνεια εργασίας των υπολογιστών μας.

Αυτές δεν ακολουθούν το μοντέλο των ιστοσελίδων που φορτώνονται διαδοχικά στον φυλλομετρητή, αλλά συνήθως είναι σελίδες που κατεβαίνουν, εγκαθίστανται και με διαδοχικές κλήσεις στον εξυπηρετητή ανανεώνεται το περιεχόμενό τους. Αυτού του τύπου οι εφαρμογές είναι γραμμένες σε JavaScript και είναι συνήθως αρκετά σύνθετες, με πολλές γραμμές κώδικα. Το τμήμα της εφαρμογής που τρέχει στον εξυπηρετητή παίζει σε αυτή την περίπτωση λιγότερο σημαντικό ρόλο, και κύρια παρέχει υπηρεσίες όπως σύνδεση με βάση δεδομένων και άλλες πηγές μόνιμης αποθήκευσης, κλπ.

1.1.3 Εκδόσεις της JavaScript

Ας εξετάσουμε ιστορικά τις πιο σημαντικές εκδόσεις της JavaScript ή ECMAScript, όπως είναι το επίσημο όνομά της:

- Η πρώτη σημαντική έκδοση ήταν η **ES3 (JavaScript 3)** που ήταν το πρώτο πλήρες πρότυπο, στα πλαίσια του οποίου εισήχθησαν σημαντικές νέες λειτουργίες όπως κανονικές εκφράσεις (regular expression literals), χειριστής εξαιρέσεων try/catch, κλπ.
- Το 2009 ορίζεται το πρότυπο **ES5 (JavaScript 5)**, που θεωρείται η πιο διαδεδομένη έκδοση της JavaScript σήμερα, αφού την υποστηρίζουν σχεδόν όλες οι εκδόσεις των φυλλομετρητών, και η οποία περιλαμβάνει τις **συναρτήσεις πινάκων** map/reduce/filter/forEach, υποστήριξη JSON, ορισμό getters/setters για πρόσβαση στις ιδιότητες αντικειμένων, κλπ.
- Η πιο σημαντική νέα έκδοση ήταν η **ES6 (ES2015)**, που θεωρείται η πιο ανεπτυγμένη και ώριμη έκδοση της γλώσσας. Η έκδοση αυτή περιλαμβάνει μεγάλες βελτιώσεις όπως τον μηχανισμό υποσχέσεων (*promises*) για ασύγχρονη εκτέλεση, εισαγωγή modules, ορισμό κλάσεων με τη λέξη *class*, ορισμό μεταβλητών με εμβέλεια στο block (*let*), συναρτήσεις βέλη (arrow functions `=> {}`), ορισμό συμβολοσειρών με ενσωμάτωση μεταβλητών (template literal), τον τελεστή ``...'' spread, ορισμό προκαθορισμένων τιμών σε παραμέτρους συναρτήσεων, ορισμό του τύπου symbol, κλπ.
- Έκτοτε σε ετήσια βάση ανακοινώνονται καινούργιες εκδόσεις της γλώσσας που φέρουν το όνομα του έτους, ES2020, ES2021, κλπ. Όμως μετά την έκδοση ES2015 οι αλλαγές δεν είναι αξιοσημείωτες και ούτως ή άλλως η πλήρης υιοθέτηση πρόσφατων αλλαγών από τους φυλλομετρητές απαιτεί χρόνο.

Ουσιαστικά η έκδοση ES6 είναι το τρέχον πρότυπο της JavaScript, και αυτή θα χρησιμοποιήσουμε στο βιβλίο αυτό, όμως επειδή υπάρχει πολύ εγκατεστημένο λογισμικό και σε προηγούμενες εκδόσεις, κύρια την ES5, θα πρέπει να λάβουμε υπόψη μας ποια ήταν η λειτουργία της γλώσσας σε προηγούμενες εκδόσεις. Θα πρέπει να αναφερθεί επίσης ότι υπάρχουν ειδικά εργαλεία για μεταγλώττιση μεταξύ εκδόσεων (transpilers), που επιτρέπουν να γράψουμε κώδικα στην έκδοση ES6 και να τον μεταγλωττίσουμε σε ES5 ώστε να μπορεί να εκτελεστεί σχεδόν σε όλους του φυλλομετρητές.

Η JavaScript σήμερα συνοδεύεται από ένα πλούσιο οικοσύστημα από εργαλεία, βιβλιοθήκες και πλαίσια ανάπτυξης. Η πιο σημαντική βιβλιοθήκη είναι jQuery η οποία υπάρχει από το 2006 αν και τείνει να υποχωρήσει αφού οι νέες εκδόσεις της JavaScript έχουν υποκαταστήσει πολλές από τις λειτουργίες της jQuery. Αξίζει να

κάνουμε επίσης αναφορά σε κάποια από τα framework για το front-end που χρησιμοποιούνται ευρύτατα σήμερα όπως είναι το React, το Angular το Vue.js, και τα οποία βασίζονται στην JavaScript.

1.1.4 Μηχανές εκτέλεσης κώδικα JavaScript

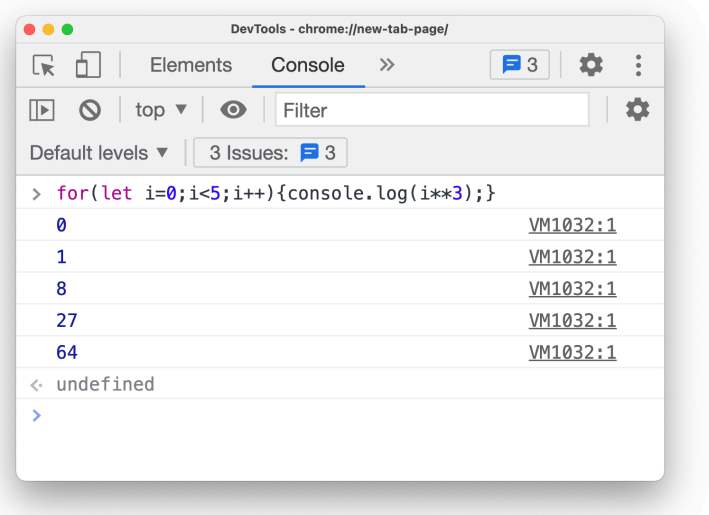
Ας δούμε τις βασικές τεχνολογίες που υπάρχουν για εκτέλεση κώδικα JavaScript. Θα πρέπει να αναφερθεί ότι κάθε φυλλομετρητής έχει διάφορες διεργασίες που εκτελούνται όταν αρχίσει το φόρτωμα μιας ιστοσελίδας. Αρχίζει μια διεργασία που κάνει συντακτική ανάλυση του κώδικα HTML και κτίζει το DOM, όπως είδαμε σε προηγούμενο κεφάλαιο. Αυτή η δεργασία σε συνδυασμό με τον συντακτικό αναλυτή των σχετικών φύλλων CSS, τροφοδοτεί τη διεργασία που ονομάζεται *rendering engine*, που είναι η διεργασία που εμφανίζει την ιστοσελίδα στο παράθυρο του φυλλομετρητή. Επίσης υπάρχει η *JavaScript engine* που αναλαμβάνει τη διερμηνεία και εκτέλεση του κώδικα JavaScript που σχετίζεται με το συγκεκριμένο έγγραφο HTML. Μάλιστα οι σύγχρονοι φυλλομετρητές δεν διερμηνεύουν την JavaScript αλλά την μεταγλωττίζουν με just in time τεχνολογία για καλύτερη απόδοση.

Οι κυρίες μηχανές JavaScript που έχουμε στους φυλλομετρητές σήμερα είναι η **V8** που υπάρχει στον *Chrome* καθώς και στο framework **Node.js**, που είναι το περιβάλλον εκτέλεσης εφαρμογών JavaScript στον εξυπηρετητή. Μια άλλη μηχανή JavaScript είναι η **SpiderMonkey** που είναι ενσωματωμένη στον φυλλομετρητή *FireFox*, καθώς και η **JavaScriptCore** (Nitro) που βρίσκεται στον φυλλομετρητή *Safari*.

1.2 Ανάπτυξη και εκσφαλμάτωση κώδικα JavaScript

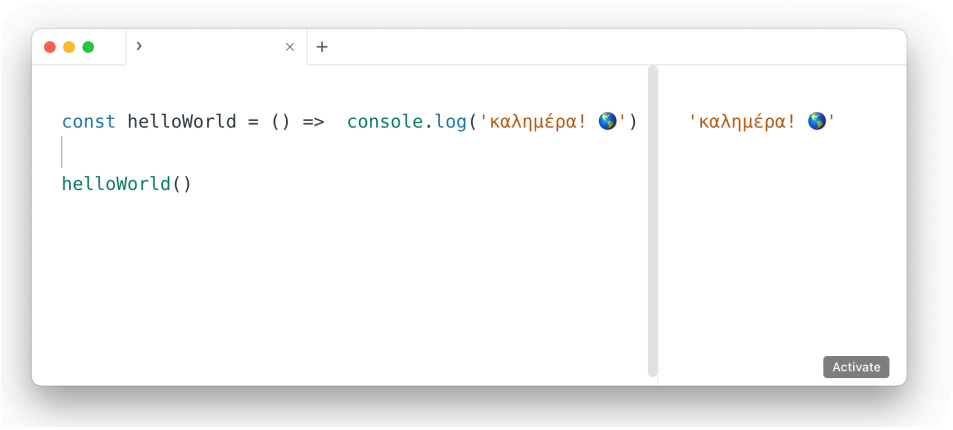
Για την ανάπτυξη και εκσφαλμάτωση κώδικα JavaScript μπορούμε να χρησιμοποιήσουμε οποιοδήποτε περιβάλλον ανάπτυξης, όπως ο *Microsoft VS Code*. Η εκτέλεση του κώδικα θα γίνει στο περιβάλλον του φυλλομετρητή, αφού ενσωματώσουμε τον κώδικα σε μια άδεια σελίδα HTML. Τα μηνύματα σφάλματος εμφανίζονται στην κονσόλα στα εργαλεία προγραμματιστή του φυλλομετρητή (ενεργοποιούνται συνήθως με function-F12).

Αν επιθυμούμε να πειραματιστούμε με τμήματα κώδικα, μπορούμε να χρησιμοποιήσουμε την κονσόλα του φυλλομετρητή, την οποία μάλιστα μπορούμε να αποσπάσουμε σε ξεχωριστό παράθυρο και εκεί να γράψουμε κώδικα, ο οποίος εκτελείται αμέσως. Ένα παράδειγμα εκτέλεσης ενός βρόχου εκτύπωσης των κύβων των αριθμών 0..4 φαίνεται στην παρακάτω εικόνα για την κονσόλα του φυλλομετρητή Chrome.



Σχήμα 1.2: Εκτέλεση κώδικα JavaScript στην κονσόλα του Chrome.

Μια εναλλακτική επιλογή που επιτρέπει δοκιμές με script της JavaScript σε περιβάλλον node.js είναι η χρήση της εφαρμογής RunJS που διατίθεται σε δωρεάν έκδοση. Η εφαρμογή αυτή παρουσιάζει στην τυπική της μορφή δύο παράθυρα, στο ένα γράφουμε κώδικα, και στο δίπλα βλέπουμε το αποτέλεσμα, όπως φαίνεται στην εικόνα που ακολουθεί, για το κλασικό πρόγραμμα helloWorld.



Σχήμα 1.3: Εκτέλεση κώδικα JavaScript στο περιβάλλον RunJS.

Η σύνταξη της γλώσσας είναι κοινή στα δύο αυτά περιβάλλοντα (εξάλλου και τα δύο χρησιμοποιούν την ίδια μηχανή JS, την V8), αν και στη δεύτερη περίπτωση, λείπουν κάποιες διεπαφές, όπως η διεπαφή στο DOM και BOM, που παρέχονται στο περιβάλλον του φυλλομετρητή, όπως θα περιγραφεί στη συνέχεια του κεφαλαίου αυτού.

1.3 Σύνταξη ενός προγράμματος JavaScript

Όπως μπορείτε να διαπιστώσετε από τα παραδείγματα της προηγούμενης ενότητας, η σύνταξη της JavaScript μοιάζει με άλλες γλώσσες προγραμματισμού όπως η C, Java, κλπ. Είναι ευαίσθητη στα κεφαλαία, μικρά γράμματα, και ο κώδικας του προγράμματος γράφεται με κωδικοποίηση UTF-8, κάτι που σημαίνει ότι μπορούμε να εισάγουμε ελληνικούς χαρακτήρες ως χαρακτήρες σε συμβολοσειρές (strings) χωρίς πρόβλημα.

Ο χαρακτήρας ``;'' ορίζεται ως τερματικός χαρακτήρας εντολής, που σημαίνει ότι μπορούν να συνυπάρξουν πολλαπλές εντολές σε μια γραμμή. Όμως σε αντίθεση με άλλες γλώσσες προγραμματισμού η JavaScript δεν θεωρεί τον τερματικό χαρακτήρα υποχρεωτικό και μπορεί να παραληφθεί, αν ο συντακτικός αναλυτής μπορεί να συνάγει σαφώς που τελειώνει η κάθε εντολή. Γενικά είναι καλή πρακτική να χρησιμοποιείται πάντα ο τερματικός χαρακτήρας στο τέλος εντολών, αν και κάποιοι προγραμματιστές προτιμάνε να τον χρησιμοποιούν μόνο όταν είναι απαραίτητος, δηλαδή όπου η παράλλειψη του μπορεί να προκαλέσει σύγχυση στον συντακτικό αναλυτή.

Τα σχόλια μιας γραμμής αρχίζουν με τους χαρακτήρες `"//"` , ενώ σχόλια πολλών γραμμών αρχίζουν με την ακολουθία χαρακτήρων `/*` και τερματίζουν με την ακολουθία `*/`.

```
// σχόλιο μιας γραμμής

/* αυτό είναι ένα
σχόλιο που εκτείνεται
σε πολλές γραμμές */
```

Τα ονόματα σταθερών, μεταβλητών, συναρτήσεων, κλάσεων, γνωρισμάτων αντικειμένων κλπ (identifiers) στην JavaScript πρέπει να ακολουθήσουν κάποιους κανόνες. Τα ονόματα αυτά πρέπει να περιέχουν γράμματα, αριθμούς ή τα σύμβολα `"_"` και `"$"`. Όμως ένα όνομα δεν πρέπει να αρχίζει με αριθμητικό χαρακτήρα.

Συνεπώς επιτρεπτά ονόματα μεταβλητών είναι τα `_, $, _1, $1, a1` ενώ δεν είναι επιτρεπτά τα `1a, 1$, 12, 1_`.

Επίσης υπάρχουν κάποιες δεσμευμένες λέξεις που δεν μπορούν να χρησιμοποιηθούν, όπως τα ονόματα εντολών ή δομών `if, else, const, null, continue, this, while, false, return, throw, with, break, in, true, case, for, instanceof, try, catch, let, super, typeof, class, function, new, switch, var`, κλπ.

Θεωρητικά θα μπορούσε να χρησιμοποιηθεί οποιοσδήποτε χαρακτήρας UTF-8 στο όνομα μιας μεταβλητής, άρα δεν είναι λάθος να γράψουμε:

```
μεταβλητή = 10;
console.log(μεταβλητή)
```

Όμως δεν θεωρείται καλή πρακτική για την μεταφερσιμότητα του κώδικά μας να χρησιμοποιήσουμε χαρακτήρες εκτός ASCII (λατινικό αλφάβητο) στα ονόματα μεταβλητών.

Κατά σύμβαση τα ονόματα των μεταβλητών, γνωρισμάτων αντικειμένων και συναρτήσεων ακολουθούν τη σημειογραφία καμήλας `camelNotation`, δηλαδή να αρχίζουν με μικρό γράμμα και όταν περιέχουν πολλές λέξεις

(συνήθως της Αγγλικής γλώσσας) κάθε αρχικό των επόμενων λέξεων γράφεται με κεφαλαίο, γραφή που θυμίζει τις καμπούρες της καμήλας, εξού και το όνομα.

Ο τύπος των δεδομένων που θα εκχωρηθούν σε μια μεταβλητή ή σταθερά δεν ορίζεται κατά τη δήλωση της μεταβλητής, αλλά προκύπτει από την αρχική τιμή που η μεταβλητή θα πάρει. Η διάλεκτος Typescript προσπαθεί να διορθώσει αυτό το πρόβλημα.

Θα πρέπει να δηλώσουμε τις μεταβλητές ή τις σταθερές πριν τις χρησιμοποιήσουμε με τις λέξεις let/const/var όπως θα δούμε στη συνέχεια.

1.3.1 Δήλωση μεταβλητών let/const/var

Σε αυτή την ενότητα θα δούμε τρόπους που διαθέτουμε για δήλωση μεταβλητών ή σταθερών.

Μέχρι την έκδοση ES5 της Javascript είχαμε μόνο ένα τρόπο να δηλώνουμε μεταβλητές, με τη λέξη κλειδί **var** (για variable).

```
var myName = 'Nikos';
var myAge =  75;
```

Με την έκδοση ES6 προστέθηκαν δύο ακόμη λέξεις κλειδιά για δήλωση μεταβλητών, η **let** (let it be...) και η **const** (constant).

```
let myName = 'Nikos';
let myAge =  75;
const pi =  3.14;
```

Ποιος ο λόγος για εισαγωγή των δύο αυτών τρόπων ορισμού μεταβλητών, υπάρχουν διαφορές μεταξύ τους;

Η απάντηση είναι ότι υπάρχουν σημαντικές διαφορές και η πρόταση εξ αρχής που έχουμε να κάνουμε είναι να ξεχάσουμε τη `var`, και να χρησιμοποιούμε την `const` στις περισσότερες περιπτώσεις, και την `let` στις υπόλοιπες, που είναι μόνο για τις περιπτώσεις μεταβλητών που αναφέρονται σε **πρωτογενείς τύπους δεδομένων** (Number, String, Boolean, κλπ), των οποίων η τιμή πρόκειται να αλλάξει.

Ας εξετάσουμε στη συνέχεια τις κύριες διαφορές μεταξύ let και var. Αυτές εντοπίζονται στα εξής:

- Διαφορές στη δυνατότητα επαναδήλωσης μιας μεταβλητής (στη var επιτρέπεται, στη let όχι)
- Διαφορές στην εμβέλεια (η var έχει εμβέλεια και έξω από το μπλοκ στο οποίο δηλώνεται, ενώ η let όχι)
- Διαφορές στην δυνατότητα πρόσβασης στην μεταβλητή πριν τη δήλωσή της (στη var επιτρέπεται, η τιμή είναι αρχικά undefined, στη let αυτό δεν επιτρέπεται).

Ας δούμε μερικά σχετικά παραδείγματα:

1.3.2 Πολλαπλές δηλώσεις

ο παρακάτω κώδικας είναι αποδεκτός:

```
var myName = 'Nikos';
var myName = 'Kostas';
```

αντίθετα ο παρακάτω κώδικας δίνει error:

```
let myName = 'Nikos';
let myName = 'Kostas';
> SyntaxError: Identifier 'myname' has already been declared
```

Βεβαίως μπορούμε να αλλάξουμε την τιμή μιας μεταβλητής που έχει δηλωθεί με `let` χωρίς όμως να την ξαναδηλώσουμε:

```
let myName = 'Nikos';
myName = 'Kostas';
```

Αντίθετα δεν μπορούμε να αλλάξουμε μια μεταβλητή που έχει δηλωθεί με την λέξη κλειδί `const` αν η τιμή της είναι πρωτογενούς τύπου.

```
const myName = 'Nikos';
myName = 'Kostas';
> TypeError: Assignment to constant variable.
```

Θα πρέπει να προσέξουμε όμως ότι η δήλωση με τη λέξη κλειδί `const` μεταβλητών που αναφέρονται σε **τύπους δεδομένων αναφοράς**, όπως τα αντικείμενα (object) ή οι πίνακες, δεν μας αποκλείει από τη δυνατότητα να τροποποιήσουμε στη συνέχεια το περιεχόμενο αυτών των αντικειμένων.

Για παράδειγμα ο παρακάτω κώδικας είναι απόλυτα αποδεκτός:

```
const ourNames = ['Nikos', 'Kostas', 'Maria'];
ourNames[0] = 'Katerina';
console.log(ourNames);
> ["Katerina", "Kostas", "Maria"]
```

1.3.3 Εμβέλεια μεταβλητών let

Μια σημαντική διαφορά μεταξύ let και var είναι ότι η let ορίζει εμβέλεια της μεταβλητής μόνο στο μπλοκ στο οποίο έχει δηλωθεί (μπλοκ μπορεί να θεωρηθεί μια συνάρτηση αλλά και μια εντολή if, for, κλπ), ενώ η var έχει εμβέλεια σε ολόκληρο τον κώδικα.

Ας συγκρίνουμε το αποτέλεσμα των δύο αυτών παραδειγμάτων:

```
for(var i = 0; i<10; i++) {
}
console.log(i)
> 10

for(let i = 0; i<10; i++) {
}
console.log(i)
> ReferenceError: i is not defined
```

Στο δεύτερο παράδειγμα που η μεταβλητή i ορίστηκε με χρήση της λέξης κλειδί let, αυτή δεν είχε εμβέλεια εκτός του μπλοκ for μέσα στο οποίο ορίστηκε, και για αυτό πήραμε μήνυμα σφάλματος.

1.3.4 Κλήση μεταβλητής πριν τη δήλωσή της

Η let δεν μπορεί να χρησιμοποιηθεί πριν δηλωθεί, ενώ η var μπορεί να χρησιμοποιηθεί (*hoisting*), και έχει την τιμή undefined.

```
console.log('x=', typeof x, x); // x undefined NaN
console.log('y=', typeof y, y); // ReferenceError
var x=5;
let y=10;
```

1.3.5 Τελεστές και εκφράσεις

Έχουμε ήδη δώσει παραδείγματα εντολών εκχώρησης τιμής σε μεταβλητή. Αυτή γίνεται με χρήση του τελεστή ``=":

```
μεταβλητή = έκφραση;
```

Σε μια έκφραση μπορούν να χρησιμοποιηθούν οι δυαδικοί τελεστές αριθμητικών πράξεων που συναντιώνται και σε άλλες γλώσσες προγραμματισμού: ``+" για πρόσθεση, ``-" για αφαίρεση, "*" για πολλαπλασιαμό, ``/" για διαίρεση, ``%" για υπόλοιπο διαίρεσης, ενώ από την έκδοση ES2016 προστέθηκε ο τελεστής "**" για ύψωση σε δύναμη. Επίσης σε μια έκφραση μπορεί να χρησιμοποιηθούν οι μοναδιαίοι τελεστές:

- **a**++ για απόδοση τιμής και αύξηση κατά 1
- **a**-- για απόδοση τιμής και μείωση κατά 1
- ++**a** για αύξηση τιμής κατά 1 και απόδοση τιμής στη συνέχεια
- --**a** για μείωση τιμής κατά 1 και απόδοση τιμής στη συνέχεια

καθώς και οι συντομογραφίες εκφράσεων:

- **a += b** Αύξηση της τιμής του a κατά b, δηλαδή a = a + b
- **a -= b** Μείωση της τιμής του a κατά b, δηλαδή a = a - b;

1.3.6 Τελεστές πράξεων δυαδικών αριθμών

Αν έχουμε δύο μεταβλητές a,b που εκφράζουν δυαδικούς αριθμούς, υπάρχουν οι εξής τελεστές που επιτρέπουν πράξεις μεταξύ τους:

- `a & b` Bitwise AND ανάμεσα στις μεταβλητές
- `a | b` Bitwise OR ανάμεσα στις μεταβλητές
- `a ^ b` Bitwise XOR ανάμεσα στις μεταβλητές
- `~a` Bitwise NOT της μεταβλητής
- `a << b` Αριστερή ολίσθηση κατά b θέσεις των bit της μεταβλητής a
- `a >> b` Δεξιά ολίσθηση κατά b θέσεις των bit της μεταβλητής a

Θυμίζουμε ότι η αριστερή ολίσθηση στους δυαδικούς αριθμούς είναι ισοδύναμη με πολλαπλασιασμό με τη βάση 2, ενώ η δεξιά ολίσθηση είναι ισοδύναμε με διαίρεση με 2.

Στο επόμενο κεφάλαιο θα δούμε άλλους τελεστές που μπορούμε να χρησιμοποιήσουμε σε μια έκφραση, όπως λογικούς τελεστές, αλλά και τον τριαδικό τελεστή, καθώς και τελεστές σύγκρισης που χρησιμοποιούνται σε εκφράσεις που συνηθίζονται στις εντολές επιλογής.

1.3.6.1 Άσκηση

Ποιο το αποτέλεσμα;

```
let c = 5;
console.log(c++);
console.log(++c);
```

Απάντηση:

```
> 5
> 7
```

Αυτό γιατί η πρώτη εντολή εκτύπωσης στην κονσόλα, τυπώνει την τιμή της μεταβλητής c πριν την αύξησή της, ενώ η δεύτερη μετά την νέα αύξησή της.

1.3.7 Διεπαφή με τον χρήστη

Όπως ήδη αναφέρθηκε η JavaScript δεν έχει εγγενώς διεπαφή επικοινωνία με τον χρήστη, όπως άλλες γλώσσες προγραμματισμού. Εξάλλου είναι προορισμένη να συνεργάζεται στενά με την HTML/CSS για το σκοπό αυτό. Όμως συχνά χρειαζόμαστε είτε στο περιβάλλον του εξυπηρετητή είτε στο περιβάλλον του φυλλομετρητή το πρόγραμμά μας να τυπώσει την κατάστασή του, ή να ζητήσει από τον χρήστη κάποια δεδομένα. Στη συνέχεια κάνουμε μια σύντομη αναφορά σε τρόπους που μπορεί να χρησιμοποιήσουμε για τον σκοπό αυτό.

1.3.7.1 console.log(μήνυμα)

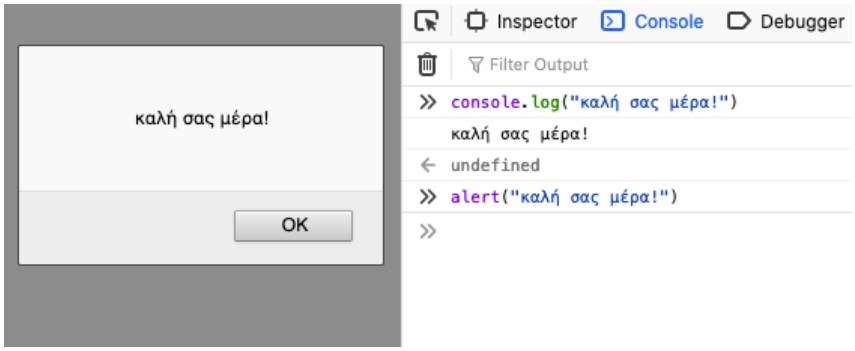
Ο πιο συνηθισμένος τρόπος είναι να χρησιμοποιήσουμε το αντικείμενο `console` που ανήκει στο global object και να καλέσουμε τη μέθοδο `console.log()`. Το αντικείμενο αυτό έχει οριστεί όπως θα δούμε στη συνέχεια και στο καθολικό επίπεδο του node.js με παρόμοια συμπεριφορά με αυτή του φυλλομετρητή.

Στα ορίσματα της συνάρτησης αυτής μπορούμε να περάσουμε εκφράσεις ή συμβολοσειρές και να πάρουμε τα αποτελέσματα στην κονσόλα. Η κονσόλα βρίσκεται στα εργαλεία ανάπτυξης όλων των φυλλομετρητών (Chrome, Firefox, Safari, κλπ.).

```
console.log("Καλημέρα");
> "Καλημέρα"
```

1.3.7.2 alert(μήνυμα)

Ένας εναλλακτικός τρόπος, μόνο για το περιβάλλον του φυλλομετρητή όμως, είναι να χρησιμοποιήσουμε τη μέθοδο `alert()` του global object.

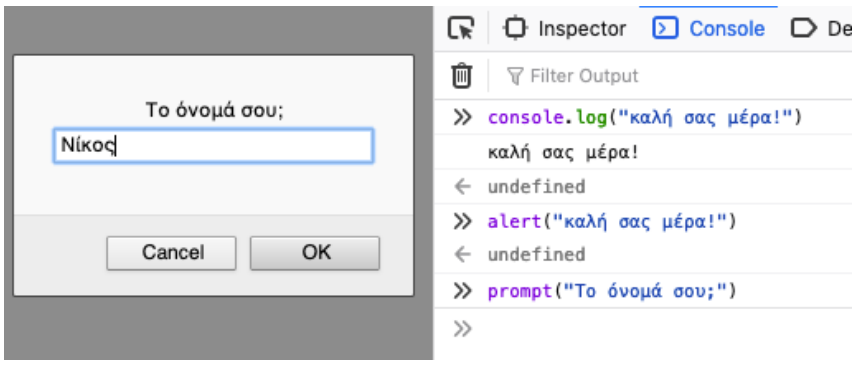


Σχήμα 1.4: Η `alert()' όπως εκτελείται στο φυλλομετρητή Firefox.

Αυτή παράγει ένα αναδυόμενο μονοτροπικό παράθυρο (modal) με το μήνυμα, όπως βλέπουμε, στην εικόνα 1.4, από την κονσόλα του Firefox.

1.3.7.3 prompt()

Παρόμοια είναι η χρήση της μεθόδου `prompt()` του global object, η οποία μέσω ενός αναδυόμενου μονοτροπικού παράθυρου ζητάει μια τιμή από τον χρήστη. Παράδειγμα στην εικόνα 1.5.



Σχήμα 1.5: Η `prompt()' όπως εκτελείται στο φυλλομετρητή Firefox.

1.3.7.4 Επικοινωνία μέσω της HTML

Εκτός από τους παραπάνω, υπάρχουν πολλοί τρόποι να χρησιμοποιηθεί η ίδια η ιστοσελίδα για επικοινωνία με τον χρήστη, όπως για παράδειγμα με τροποποίηση ενός στοιχείου, με την ιδιότητα `element.textContent` ή `element.innerHTML`, αλλά και να πάρουμε στοιχεία από τον χρήστη μέσω στοιχείων `<input>` μιας φόρμας, όπως έχουμε δει στο κεφάλαιο της HTML.

Αυτές είναι οι βασικές τεχνικές που διαθέτει η JavaScript για επικοινωνία με τον χρήστη. Στη συνέχεια του κεφαλαίου θα επιχειρήσουμε μια επισκόπηση της αρχιτεκτονικής του φυλλομετρητή και του τρόπου που εκετελείται ο κώδικας JavaScript στο πλαίσιο του, ενώ θα εξετάσουμε τη διεπαφή της JavaScript με το περιβάλλον της ιστοσελίδας και του φυλλομετρητή.

1.4 Η JavaScript στον φυλλομετρητή

Ο κώδικας JavaScript μιας ιστοσελίδας σχετίζεται με το αρχείο `.html`. Μια ιστοσελίδα μπορεί να συνοδεύεται από πολλά τμήματα κώδικα JavaScript που μπορεί να βρίσκονται είτε ενσωματωμένα στο ίδιο το αρχείο ή σε εξωτερικά αρχεία που φορτώνονται από το αρχείο `.html`.

Όλα αυτά τα τμήματα του κώδικα συνυπάρχουν στον ίδιο κοινό χώρο, συναποτελώντας το ``*πρόγραμμα JavaScript*'' της σελίδας. Μάλιστα όλα αυτά τα τμήματα κώδικα μοιράζονται τον κοινό χώρο ονομάτων μεταβλητών που ονομάζεται **global object**.

Ο κώδικας JavaScript που προέρχεται από το ίδιο το αρχείο `.html` εμφανίζεται ως περιεχόμενο στοιχείων `<script>`.

```
<script>
    // κώδικας JavaScript
</script>
```

Το στοιχείο αυτό μπορεί να βρίσκεται είτε στο τμήμα <head> του εγγράφου HTML, ή οπουδήποτε στο τμήμα <body>.

Ένας δεύτερος τρόπος να φορτώσουμε στη σελίδα κώδικα JavaScript είναι από εξωτερικό αρχείο, στο οποίο γίνεται αναφορά μέσω του γνωρίσματος src του στοιχείου <script>.

Για παράδειγμα ένα εξωτερικό αρχείο ``scriptfile.js'' φορτώνεται ως εξής:

```
<script src="scriptfile.js"></script>
```

Σε αυτή την περίπτωση το στοιχείο μπορεί να έχει μια από τις ιδιότητες async ή defer, οι ιδιότητες αυτές δεν παίρνουν τιμή, είναι λογικές ιδιότητες και καθορίζουν τη σειρά φορτώματος και εκτέλεσης του αντίστοιχου κώδικα JavaScript.

- Η ιδιότητα async ορίζει ασύγχρονο φόρτωμα και εκτέλεση του κώδικα JavaScript, που μπορεί να γίνει ακομη και πριν την ολοκλήρωση φορτώματος της HTML.
- Η ιδιότητα defer ορίζει ότι το φόρτωμα του κώδικα θα γίνει ασύγχρονα αλλά η εκτέλεση του κώδικα JavaScript θα γίνει μετά τη φόρτωση HTML ή άλλου κώδικα JS που επίσης έχει την ίδια ιδιότητα αλλά προηγείται.

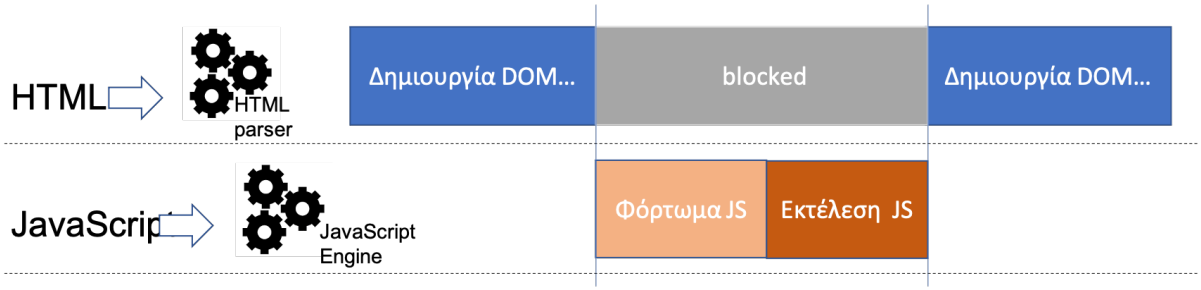
Για παράδειγμα αν στη σελίδα μας περιέχεται ο παρακάτω κώδικας:

```
<script src="js/script2.js" defer ></script>
<script src="js/script3.js" defer ></script>
```

το script3.js θα εκτελεστεί μετά το script2.js.

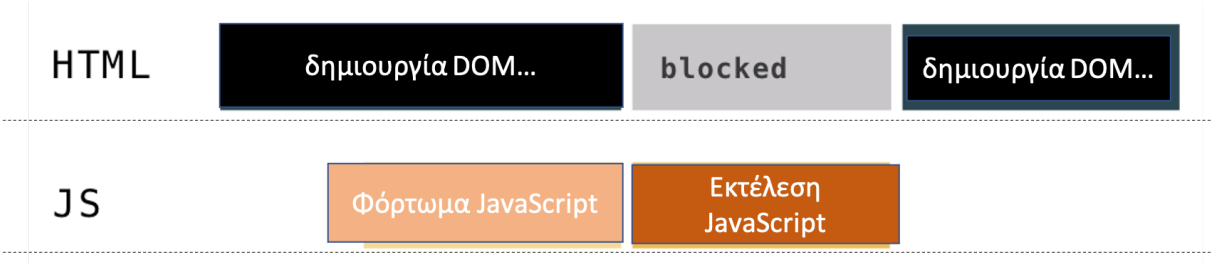
Αν δεν χρησιμοποιηθεί καμιά από αυτές τις ιδιότητες, ο κώδικας JavaScript μπλοκάρει το φόρτωμα της HTML και άρα το χτίσιμο του DOM. Αυτό γίνεται γιατί μέσα στον κώδικα JS μπορεί να κρύβονται εντολές που συμβάλουν στο κτίσιμο της HTML, πχ με την εντολή document.write() που επιτρέπει εισαγωγή κώδικα HTML στο σημείο της συγκεκριμένης εντολής. Η χρήση αυτής της δυνατότητας δεν προτείνεται, όμως οι φυλλομετρητές πρέπει να φροντίσουν και για αυτό το ενδεχόμενο.

Η παρακάτω εικόνα 1.6 περιγράφει αυτό το σενάριο.



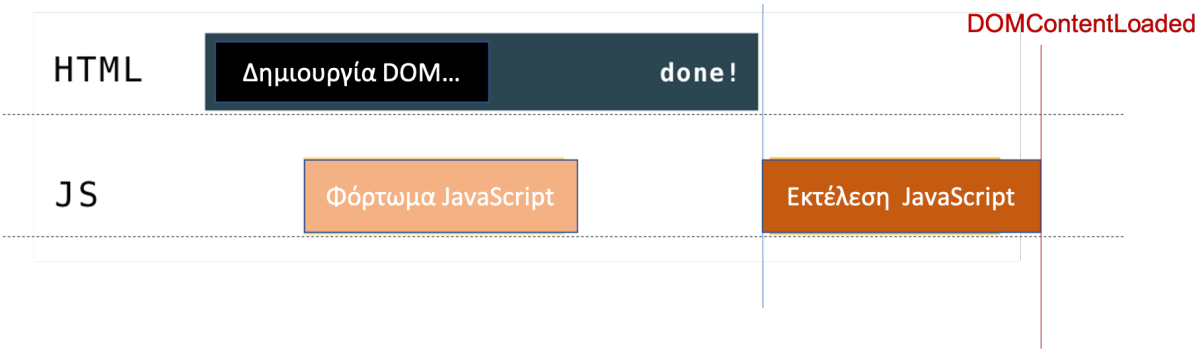
Σχήμα 1.6: Φόρτωμα κώδικα JavaScript με μπλοκάρισμα φορτώματος της HTML/DOM.

Η περίπτωση φορτώματος κώδικα JavaScript με την ιδιότητα async φαίνεται στην εικόνα 1.7.



Σχήμα 1.7: Φόρτωμα κώδικα JavaScript με την ιδιότητα async, ασύγχρονα με την HTML.

Τέλος η περίπτωση φορτώματος κώδικα JavaScript με την ιδιότητα defer φαίνεται στην επόμενη εικόνα 1.8.



Σχήμα 1.8: Φόρτωμα κώδικα JavaScript με την ιδιότητα defer, η εκτέλεση του κώδικα μεταφέρεται μετά την ολοκλήρωση του DOM.

Ποια είναι η καλύτερη πολιτική για φόρτωμα κώδικα JS; Γενικά τοποθετούμε το <script> μέσα στο <head> και χρησιμοποιούμε τα γνωρίσματα async και defer ώστε η JS να μην μπλοκάρει το κατέβασμα της σελίδας.

Χρησιμοποιούμε <script async …> αν το περιεχόμενο της σελίδας είναι ανεξάρτητο από τον κώδικα JavaScript.

Χρησιμοποιούμε <script defer … > αν επιθυμούμε ο κώδικας να τρέξει μετά την ολοκλήρωση φορτώματος της HTML, αν για παράδειγμα ο κώδικάς μας κάνει αναφορά σε στοιχεία του DOM.

Επίσης μπορούμε να θέσουμε το <script> στο τέλος του <body> ώστε το φόρτωμα και η εκτέλεση του κώδικα να μην καθυστερεί το φόρτωμα της σελίδας, ιδιαίτερα αν ο κώδικας JavaScript είναι εκτενής.

Εναλλακτικά, μπρούμε να βάλουμε τμήμα του κώδικα μέσα σε μια συνάρτηση η οποία να κληθεί μετά από ένα συμβάν, πχ. το συμβαν DOMContentLoaded που σηματοδοτεί την ολοκλήρωση φόρτωσης του DOM.

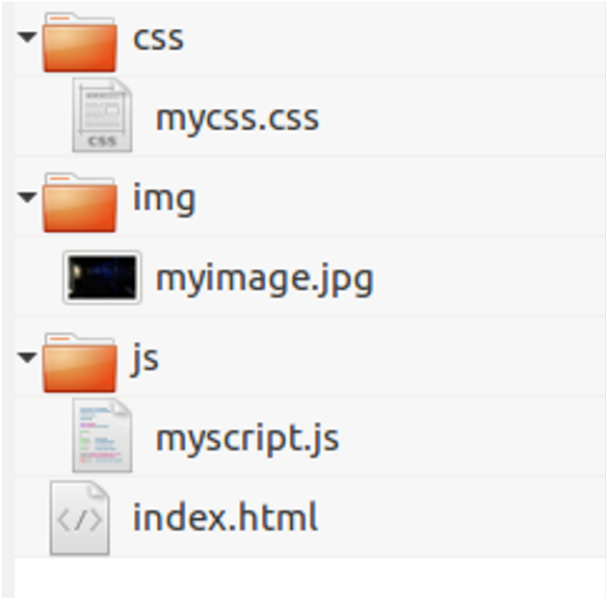
Στη συνέχεια θα κάνουμε ιδιαίτερη αναφορά στα συμβάντα που σχετίζονται με το φόρτωμα μιας σελίδας.

1.5 Διαχωρισμός κώδικα HMTL, CSS, JavaScript

Σε μια σύγχρονη διαδικτυακή εφαρμογή, μια ιστοσελίδα περιλαμβάνει ένα σύνολο από αρχεία, εκτός από το κυρίως αρχείο HTML (πχ index.html). Αυτά τυπικά περιλαμβάνουν αρχείο ή αρχεία CSS, αρχεία JavaScript καθώς και αρχεία εικόνων και άλλων πολυμέσων. Ο τρόπος που συνδέονται αυτά τα αρχεία, όπως έχει ήδη αναφερθεί είναι αυτός που φαίνεται στο παράδειγμα.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="css/mycss.css">
  </head>
  <body>
    <h1 id="first">Καλή σας μέρα</h1>
    <img src="img/myimage.jpg"></img>
    <script src="js/myscript.js"></script>
  </body>
</html>
```

Η οργάνωση των αρχείων του πρότζεκτ ανάπτυξης της ιστοσελίδας συνήθως έχει την παρακάτω δομή:



Σχήμα 1.9: Συνήθης οργάνωση των αρχείων ενός πρότζεκτ ανάπτυξης ιστοσελίδας.

1.6 Η ακολουθία συμβάντων κατά το φόρτωμα μιας σελίδας

Η εκτέλεση της JavaScript στον φυλλομετρητή ακολουθεί τη λογική των γλωσσών προγραμματισμού που στηρίζονται στο **μοντέλο χειρισμού συμβάντων (event-based model)**. Τέτοιες είναι οι γλώσσες που λειτουργούν σε διαδραστικές συνθήκες, όπως σε γραφικά περιβάλλοντα αλληλεπίδρασης με τον χρήστη. Το μοντέλο αυτό περιλαμβάνει φόρτωμα του κώδικα και αρχικοποίηση των μεταβλητών και αντικειμένων, ορισμό των χειριστών συμβάντων, δηλαδή των τμημάτων του κώδικα (συναρτήσεων) που θα εκτελεστούν όταν συμβούν καθορισμένα συμβάντα (events). Στη συνέχεια το πρόγραμμα βρίσκεται σε αναμονή των συμβάντων αυτών, τα οποία προκύπτουν συνήθως από ενέργειες του χρήστη, αλλά και συμβάντων του συστήματος, όπως ανάκτηση δεδομένων από άλλες πηγές, ολοκλήρωση φόρτωσης τμημάτων του κώδικα, κλπ. Όταν συμβεί ένα αναμενόμενο συμβαν, ενεργοποιείται ο αντίστοιχος χειριστής.

Ας ακολουθήσουμε στη συνέχεια την ακολουθία φάσεων και αντίστοιχων συμβάντων κατά το φόρτωμα της ιστοσελίδας.

Φάση 1: Φόρτωμα DOM. Ο φυλλομετρητής δημιουργεί ένα αντικείμενο **document** και αρχίζει την συντακτική ανάλυση της ιστοσελίδας, με προσθήκη κόμβων στο DOM καθώς αναλύει ένα προς ένα τα στοιχεία HTML και το περιεχόμενό τους. Η ιδιότητα `document.readyState` έχει την τιμή *`loading`* σε αυτή τη φάση. Αν κατά τη διάρκεια της συντακτικής ανάλυσης βρεθεί ετικέτα `<script>` χωρίς async, defer, φορτώνεται και εκτελείται ο κώδικας JavaScript, ενώ διακόπτεται το φόρτωμα της HTML. Αν κατά την φάση αυτή ο αναλυτής συναντήσει ετικέτα `<script async ...>` με ασύγχρονο τρόπο προχωράει στο φόρτωμα και εκτέλεση του κώδικα JavaScript.

Φάση 2: Η ανάλυση της HTML ολοκληρώνεται, η ιδιότητα `document.readyState` παίρνει την τιμή *`interactive`*. Σε αυτή τη φάση τα τμήματα κώδικα JavaScript που ήταν σε κατάσταση `defer` εκτελούνται διαδοχικά το ένα μετά το άλλο. Μετά την ολοκλήρωση εκτέλεσης των τμημάτων αυτών του κώδικα, ενεργοποιείται το συμβάν `DOMContentLoaded`, που σηματοδοτεί τη μεταβαση της JavaScript από κατάσταση φορτώματος και εκτέλεσης κώδικα, στην φάση της διαδραστικής εκτέλεσης, σε αναμονή συμβάντων.

Φάση 3: Η JavaScript έχει μπει σε κατάσταση αναμονής συμβάντων. Η σελίδα έχει φορτωθεί πλήρως, αν και τμήματα των πρόσθετων (εικόνες κλπ.), ίσως ακόμη φορτώνονται. Επίσης πιθανόν τμήματα κώδικα JavaScript σε κατάσταση `async` να είναι ακόμη σε κατάσταση φορτώματος/εκτέλεσης. Η φάση αυτή ολοκληρώνεται όταν όλα τα πρόσθετα φορτωθούν και ολοκληρωθεί η εκτέλεση του κώδικα `async`. Τότε εμφανίζεται το συμβάν `window.load`, ενώ η ιδιότητα `document.readyState = "complete"`. Με την ολοκλήρωση αυτής της φάσης ξεκινάει η ασύγχρονη λειτουργία της JavaScript.

1.6.1 Ένα παράδειγμα

Στο παράδειγμα αυτό θα φορτώσουμε μια ιστοσελίδα η οποία μάς γνωστοποιεί τη χρονική διάρκεια κάθε φάσης φορτώματος όπως προκύπτει από τα σχετικά συμβάντα. Η σελίδα φορτώνει μια σειρά από φωτογραφίες διαφορετικής ανάλυσης, από την ιστοσελίδα διάθεσης φωτογραφιών https://picsum.photos. Ο κώδικας

καταγράφει το χρόνο που απαιτείται για τα εξής συμβάντα: (α) Η αλλαγή της ιδιότητας readyState σε interactive (ολοκλήρωση φορτώματος του DOM), (β) η ενεργοποίηση του συμβάντος DOMContentLoaded που σηματοδοτεί την ολοκλήρωση εκτέλεσης του κώδικα JavaScritp, (γ) ενεργοποίηση του συμβάντος window.load που σηματοδοτεί την ολοκλήρωση φορτώματος των πρόσθετων (εικόνων).

Η ιστοσελίδα φαίνεται στη συνέχεια. θα πρέπει να σημειωθεί ότι το παράδειγμα περιέχει κώδικα JavaScript με συναρτήσεις και δομές που δεν έχουμε αναφέρει ακόμη, όμως παρουσιάζεται κύρια για το αποτέλεσμα που παράγει, το οποίο φαίνεται στην εικόνα που ακολουθεί.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>ex0</title>
    <script>
        const round = (v) => (v/1000).toFixed(2);
        report = "";
        function checkReady(e){
            report += `readyState=${document.readyState} -- ${round(e.timeStamp)}sec <br>`;
        }
        document.addEventListener('readystatechange', checkReady);
        document.addEventListener('DOMContentLoaded', (e) => {
            report += `DOMContentLoaded -- ${round(e.timeStamp)}sec :φόρτωμα DOM<br>`;
        });
        window.addEventListener('load', (e) => {
            report += `window.load -- ${round(e.timeStamp)}sec :φόρτωμα σελίδας`;
            document.querySelector("#report").innerHTML = report;
            console.log(report);
        });
    </script>
</head>
<body>
    <div style="display: flex"></div>
    <div id="report" style="margin:10px;font-family:sans-serif;
    font-size:2rem;"></div>
    <script>
        const container = document.querySelector("div");
        for (let i=0;i<5;i++){
            const d = document.createElement("div");
            d.style.padding = "5px";
            const im = document.createElement("img");
```

```
            im.src = `https://picsum.photos/${i+2}00`
            d.appendChild(im);
            container.append(d);
        }
    </script>

</body>
</html>
```

Το αποτέλεσμα από μια τυπική φόρτωση της σελίδας φαίνεται στη συνέχεια:

Σχήμα 1.10: Τυπική φόρτωση της σελίδας.

Προκύπτει ότι το φόρτωμα της σελίδας που περιλαμβάνει τις 5 εικόνες που φορτώνονται από το https://picsum.photos απαίτησε στη συγκεκριμένη περίπτωση συνολικά 0.42sec.

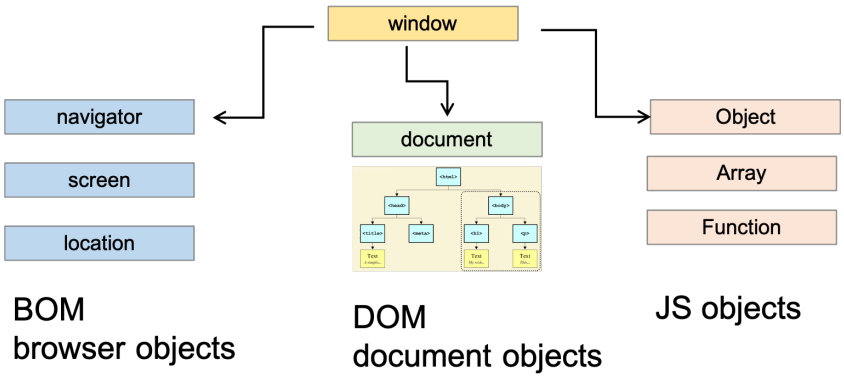
Συγκεκριμένα, το συμβάν ολοκλήρωσης της συντακτικής ανάλυσης της HTML ολοκληρώθηκε σε 0.18sec ενώ αμέσως ολοκληρώθηκε και το φόρτωμα του JavaScript κώδικα με κατάσταση defer (συμβάν DOMContentLoaded), αφού δεν υπήρχαν τέτοια τμήματα κώδικα JS. Τέλος η ολοκλήρωση φορτώματος των εικόνων που προκαλεί την αλλαγή της document.readyState = ``complete'' καθώς και του συμβάντος window.load απαιτεί όπως προκύπτει, χρόνο μέχρι 0.42sec.

Μελετήστε τον κώδικα του παραδείγματος και ιδιαίτερα την ιδιότητα του συμβάντος που επιτρέπει την καταγραφή χρόνων, που χρησιμοποιούν οι *χειριστές συμβάντων* για να μάς παρουσιάσουν τη σχετική πληροφορία.

1.7 Το περιβάλλον εκτέλεσης της JavaScript

Στο περιβάλλον του φυλλομετρητή το πρόγραμμα JavaScript που φορτώθηκε με τη διαδικασία που περιγράφτηκε στην προηγούμενη παράγραφο, έχει πρόσβαση στον κοινό χώρο ονομάτων (global scope) που λέγεται *global object*. Το αντικείμενο αυτό είναι το αντικείμενα `window`, στο οποίο μάλιστα μπορούμε να αναφερθούμε με χρήση της λέξης-κλειδί `this`. Ιδιότητες του αντικειμένου αυτού είναι τα βασικά στοιχεία της γλώσσας (built-in objects), τα οποία μπορεί να χρησιμοποιηθούν είτε με σημειογραφία τελείας `window.αντικείμενο`, είτε παραλείποντας την αναφορά στο αντικείμενο window.

Οι κατηγορίες αντικειμένων που έχει πρόσβαση η JavaScript μέσω του global object φαίνονται στο παρακάτω διάγραμμα 1.11.



Σχήμα 1.11: Κατηγορίες αντικειμένων που έχει πρόσβαση η JavaScript.

- JS objects. Μια πρώτη κατηγορία είναι τα αντικείμενα της γλώσσας, είτε τα συστατικά της στοιχεία, είτε τα ανικείμενα που δημιουργεί ο κώδικας του χρήστη.
- DOM. Μια δεύτερη κατηγορία είναι τα αντικείμενα του *Document Object Model* που ανήκουν στο αντικείμενο `document`. Σε επόμενη ενότητα θα κάνουμε ιδιαίτερη αναφορά στη **διεπαφή DOM**, δηλαδή το σύνολο των μεθόδων που διαθέτει η JavaScript για αλληλεπίδραση με τα στοιχεία του DOM.
- Browser objects. Συχνά τα στοιχεία αυτά αναφέρονται ως BOM (browser object model), περιλαμβάνουν μεταξύ των άλλων το αντικείμενο navigator που περιέχει στοιχεία για το περιβάλλον του φυλλομετρητή και του λειτουργικού συστήματος του χρήστη, το αντικείμενο screen που αφορά την οθόνη του υπολογιστή του χρήστη, το αντικείμενο location που αφορά το URL της σελίδας, κλπ.

1.7.1 Δραστηριότητα

Μεταβείτε στην κονσόλα του φυλλομετητή σας και πληκτρολογήσετε `window` ή `this`. Επιθεωρήστε τις ιδιότητες του αντικειμένου αυτού. Ιδιαίτερα εξετάστε τα αντικείμενα του φυλλομετρητή που αναφέρθηκαν καθώς και το περιεχόμενο του DOM.

1.8 Βασικά στοιχεία του αντικειμένου window

Αν επιθεωρήσουμε τις ιδιότητες του αντικειμένου `window` μεταξύ αλλων θα βρούμε αντικείμενα με ιδιότητες και μεθόδους που χρησιμοποιούνται ευρύτατα στην JS.

Θα κάνουμε σύντομη αναφορά σε αυτά, ενώ σε επόμενες ενότητες θα γίνει ιδιαίτερη μνεία στις κυριότερες από αυτές.

1.8.1 Συναρτήσεις

- **eval(έκφραση)** : υπολογισμός του αποτελέσματος της έκφρασης, θεωρείται κακή πρακτική η χρήση της για λόγους ασφάλειας.
- **isFinite(εκφραση)** : επιστρέφει `false` αν το αποτέλεσμα της έκφρασης είναι +Infinity, -Infinity, NaN ή undefined, αλλιώς `true`
- **isNaN(τιμή)** : επιστρέφει `true` αν η *τιμή* επιστρέφει NaN, αλλιώς `false`.
- **parseFloat(συμβολοσειρά)** : επιστρέφει την αριθμητική τιμή ως δεκαδικό αριθμό. Αν όμως η συμβολοσειρά περιέχει μη αριθμητικούς χαρακτήρες από ένα χαρακτήρα και μετά, λαμβάνει υπόψη του το ως τότε τμήμα της συμβολοσειράς. Αν όμως η συμβολοσειρά αρχίζει με μη αριθμητικό χαρακτήρα επιστρέφει NaN
- **parseInt(συμβολοσειρά, βάση)** : παρόμοια με την parseFloat() για ακέραιο αριθμό, παίρνει ως δεύτερο προαιρετικό όρισμα τη βάση του αριθμητικού συστήματος (πχ 16 για το δεκαεξαδικό, κλπ.)
- **encodeURI(URI)** : Κωδικοποίηση δεδομένων με την τεχνική URL Encoding, βάσει της κωδικοσελίδας UTF-8, λαμβάνοντας υπόψη ειδικούς χαρακτήρες όπως `;,/?:@&=+$#`, για παράδειγμα κωδικοποιεί τα κενά ως %20.
- **encodeURIComponent()** : όπως η προηγούμενη μόνο που αφορά δεδομένα που τα τοποθετηθούν σε μεταβλητές PUT GET
- **decodeURI()** : αποκωδικοποίηση της encodeURI()
- **decodeURIComponent()** : αποκωδικοποίήση της encodeURIComponent()

1.8.2 Βασικά Αντικείμενα

- **Object** : το αρχέτυπο αντικείμενο, τα περισσότερα αντικείμενα της JavaScript κληρονομούν από αυτό.
- **Function** : η κλάση των συναρτήσεων της JavaScript που είναι αντικείμενα πρώτης τάξης.
- **Boolean** : αντικείμενο που αντιστοιχεί σε λογικές τιμές
- **Symbol** : αντικείμενο που αντιστοιχεί στα δεδομένα τύπου Symbol

1.8.3 Αντικείμενα Αριθμών

- **Number** : αντικείμενο που αντιστοιχεί στον πρωτογενή τύπο δεδομένων Number

- **BigInt** : αντικείμενο που αντιστοιχεί στον πρωτογενή τύπο δεδομένων BigInt για άριθμους μεγαλύτερους από 2^{53} - 1
- **Math** : αντικείμενο με ιδιότητες και μεθόδους για μαθηματικές συναρτήσεις και σταθερές.
- **Date** : αντικείμενο που εκφράζει χρονική στιγμή σε milliseconds με αφετηρία μέτρησης 1 Ιανουαρίου 1970 UTC (τύπου Number)

1.8.4 Κείμενο

- **String** : αντικείμενο που αφορά την αναπαράσταση και μεθόδους που αφορούν συμβολοσειρές
- **RegExp** : αντικείμενο που επιτρέπει την αντιστοίχιση προτύπων χαρακτήρων (κανονικές εκφράσεις) με συμβολοσειρές

1.8.5 Συλλογές αντικειμένων

- **Array** : αντικείμενο που αφορά την κλάση αντικειμένων τύπου Array (πίνακες), να σημειωθεί ότι υπάρχουν ακόμη αντικείμενα για typed Arrays, πίνακες που δέχονται ορισμένου τύπου δεδομένα, όπως Float32Array, κλπ.
- **Map** : αντικείμενο που αφορά την κλάση Map, ακολουθία ζευγών κλειδί: τιμή που θυμάται τη σειρά δημιουργίας της.
- **Set** : αντικείμενο που αφορά την κλάση Set, που επιτρέπει την αποθήκευση συλλογής μοναδικών στοιχείων

1.8.6 Σφάλματα

- **Error** : αντικείμενα τα οποία δημιουργούνται όταν συμβούν σφάλματα κατά τη διάρκεια εκτέλεσης του κώδικα, υπάρχουν ειδικοί τύποι σφαλμάτων, πχ ReferenceError, κλπ.

Υπάρχουν πολλά ακόμη αντικείμενα ως ιδιότητες του αντικειμένου window.

Ακόμη, το αντικείμενο window πέραν του να ορίζει τον καθολικό χώρο ονομάτων και να έχει ως ιδιότητες τα βασικά στοιχεία της γλώσσας (built-ins) έχει ένα ακόμη ρόλο να παίξει, περιέχει πληροφορίες για το παράθυρο του φυλλομετρητή στο οποίο εμφανίζεται η ιστοσελίδα, όπως οι ιδιότητες innerHeight/innerWidth που περιέχουν τις διαστάσεις του παράθυρου.

Για πλήρη περιγραφή των αντικειμένων και ιδιοτήτων του global object μπορείτε να συμβουλευτείτε τη σχετική σελίδα στο MDN.

1.9 Διεπαφή με το Document Object Model (DOM)

Σε αυτή την ενότητα θα εξετάσουμε τη διεπαφή της JavaScript με τα στοιχεία της ιστοσελίδας, μέσω της ιδιότητας `document` του global object, η οποία έχει ως τιμή το αντικείμενο `Document`.

Το αντικείμενο αυτό αναπαριστά τα στοιχεία του εγγράφου HTML, που εμφανίζεται στο παράθυρο ή στην καρτέλα του φυλλομετρητή. Έχει οριστεί μια προγραμματιστική διεπαφή με τα στοιχεία αυτά, που ονομάζεται Document Object Model (DOM). Έχει γίνει ήδη αναφορά στην αναπαράσταση των στοιχείων ενός εγγράφου HTML ως ιεραρχία κόμβων με ρίζα το στοιχείο `<html>`. Το DOM API ορίζει στο περιβάλλον της JavaScript ένα σύνολο από αντικείμενα, που αντανακλούν την ιεραρχία στοιχείων της HTML. Για κάθε στοιχείο HTML υπάρχει ένα αντίστοιχο αντικείμενο (`Element`) JavaScript και για κάθε κείμενο του αρχείου HTML, υπάρχει ένα αντικείμενο κειμένου (`Text`). Τα αντικείμενα αυτά είναι εξειδικεύσεις του αντικειμένου `Node`.

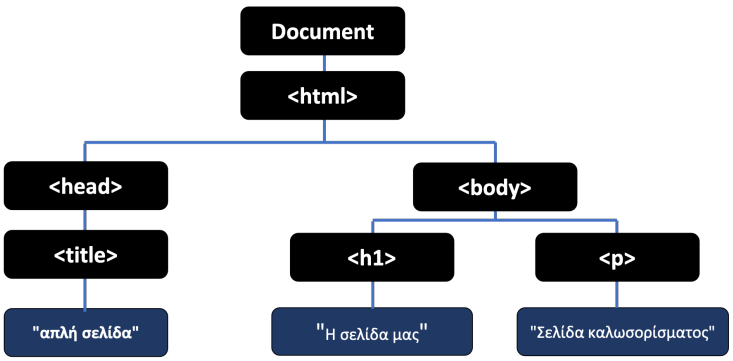
Συνεπώς στην ιεραρχία του DOM εμπεριέχονται σαν αντικείμενα, όλα τα στοιχεία της ιστοσελίδας. Να σημειωθεί ότι *αντικείμενο (object)* είναι μια βασική δομή δεδομένων της JavaScript που θα δούμε στη συνέχεια. Στα αντικείμενα αυτά έχει πρόσβαση ένα πρόγραμμα JavaScript, το οποίο μπορεί να τα τροποποιήσει. Συνεπώς το document object model είναι η προγραμματιστική διεπαφή του εγγράφου HTML και είναι ο μηχανισμός που παρέχει στην JavaScript δυνατότητα πρόσβασης στο περιεχόμενο της ιστοσελίδας.

Ας πάρουμε καταρχάς ένα παράδειγμα, απλού εγγράφου HTML.

```
<!DOCTYPE html>
<html lang="el">
    <head>
        <meta charset="utf-8">
        <title>απλή σελίδα</title>
    </head>
    <body>
        <h1>Η σελίδα μας</h1>
        <p>Σελίδα καλωσορίσματος</p>
    </body>
</html>
```

Σε αυτό το έγγραφο, βλέπουμε ότι η ρίζα είναι το `<html>`, υπάρχει το στοιχείο `<head>` που είναι παιδί του `<html>`, το `<body>` που είναι επίσης ένα άλλο παιδί του `<html>` και αυτά με τη σειρά τους έχουν άλλα στοιχεία, ως παιδιά τους.

Το document object model θα μπορούσαμε να το δούμε ως μία ιεραρχία από κόμβους ως εξής:



Σχήμα 1.12: Το Document Object Model (DOM) είναι μια ιεραρχία από κόμβους.

Στη συνέχεια θα δούμε την προγραμματιστική διεπαφή DOM API, που περιλαμβάνει τις συναρτήσεις που θα χρησιμοποιήσουμε σε ένα πρόγραμμα JavaScript για να επικοινωνήσει με αυτά τα στοιχεία.

1.9.1 Ανάκτηση στοιχείων του DOM

Η διεπαφή περιλαμβάνει διάφορες μεθόδους οι οποίες μας επιτρέπουν την ανάκτηση στοιχείων του δένδρου.

Η πιο σημαντική συνάρτηση είναι η querySelector(). Όπως και όλες οι συναρτήσεις του DOM API είναι μέθοδος του αντικειμένου document γι' αυτό την καλούμε ως `document.querySelector()`. Ως όρισμα περνάμε μία προδιαγραφή για το στοιχείο που αναζητούμε, χρησιμοποιώντας τη σύνταξη των επιλογέων της CSS. Για παράδειγμα αν δώσουμε όρισμα το `"a"`, η συνάρτηση θα μάς επιστρέψει το πρώτο στοιχείο τύπου `<a>`, δηλαδή το πρώτο anchor του εγγράφου. Η κλήση της `document.querySelector(".myClass")` θα επιστρέψει το πρώτο στοιχείο που έχει γνώρισμα `class = "myClass"`, ενώ η `document.querySelector("#myId")` θα επιστρέψει το στοιχείο με γνώρισμα `id = "myId"`, κλπ.

Μία παραλλαγή του querySelector είναι η `document.querySelectorAll()`. Η μέθοδος αυτή επιστρέφει μια συλλογή με στοιχεία, σε αντίθεση με την querySelector() που επιστρέφει ένα μόνο στοιχείο. Θα μας επιστρέψει όλα τα στοιχεία που ικανοποιούν τον επιλογέα, για παράδειγμα η `document.querySelectorAll("a")` θα επιστρέψει όλα τα στοιχεία τύπου anchor `<a>`, που υπάρχουν στην ιστοσελίδα.

Υπάρχουν και άλλες μέθοδοι για αναζήτηση στοιχείων του DOM, αυτές είναι:

- `document.getElementById("myid")` επιστρέφει το στοιχείο που έχει `id="myid"`,
- `document.getElementsByClassName("myClass")` επιστρέφει τη συλλογή στοιχείων με όνομα κλάσης `class = "myClass"`.
- `document.getElementsByTagName("tag")` επιστρέφει συλλογή στοιχείων με όνομα ετικέτας `tag`.

Κατά κάποιο τρόπο οι μέθοδοι αυτές έχουν υπερκαλυφθεί από την διεπαφή querySelector που είδαμε, η οποία παίρνει ως όρισμα έναν επιλογέα που μπορεί να αντιστοιχεί σε ένα element, σε κλάση, attribute, id, κλπ.

1.9.2 Διαχείριση των γνωρισμάτων στοιχείων DOM

Στη συνέχεια θα δούμε τις μεθόδους της διεπαφής DOM που αφορούν τα γνωρίσματα των στοιχείων. Έστω ένα στοιχείο `element`, είναι ένα αντικείμενο που μας επέστρεψε η μέθοδος querySelector. Μπορούμε με την μέθοδο `element.setAttribute(atr, val)` να δώσουμε τιμή σε ένα γνώρισμά του, `element.getAttribute(attr)` να πάρουμε την τιμή ενός γνωρίσματος, ενώ με την `element.removeAttribute(attr)` να καταργήσουμε ένα γνώρισμα.

1.9.3 Διαχείριση περιεχομένου στοιχείων DOM

Για τη διαχείριση του περιεχομένου των στοιχείων του DOM, η διεπαφή διαθέτει δύο ιδιότητες, την `innerHTML`, και την `textContent`.

Η ιδιότητα `innerHTML` έχει δύο χρησεις:

(α) Η `element.innerHTML` επιστρέφει το περιεχόμενο του στοιχείου element, περιλαμβανομένου του κειμένου που αυτό περιέχει, καθώς και του κώδικα HTML.

(β) Επιπροσθέτως, μπορεί να χρησιμοποιηθεί για να δώσουμε τιμή στο περιεχόμενο του στοιχείου element, `element.innerHTML = text`

Αντίστοιχη είναι η ιδιότητα `element.textContent`, η οποία μας επιστρέφει μόνο το κείμενο που περιέχεται στο στοιχείο.

Συνεπώς, θα πρέπει να τονιστεί ότι η διαφορά της ιδιότητας `textContent` από την ιδιότητα `innerHTML` ενός στοιχείου, είναι ότι η πρώτη περιέχει μόνο το κείμενο που περιέχεται στο στοιχείο, ενώ η δεύτερη περιέχει όλον τον κώδικα HTML που περιέχεται στο στοιχείο, τα στοιχεία παιδιά του, κλπ.

1.9.4 Διαχείριση του στυλ των στοιχείων DOM

Η διεπαφή DOM μάς επιτρέπει να έχουμε πρόσβαση και να διαχειριστούμε το στυλ των στοιχείων.

Συγκεκριμένα, η ιδιότητα `element.style` μας επιστρέφει την προδιαγραφή του στυλ του συγκεκριμένου στοιχείου όπως καθορίζεται από τα φύλλα στυλ που σχετίζονται με το στοιχείο αυτό, ενώ επίσης μάς επιτρέπει να τροποποιήσουμε αυτή την προδιαγραφή, αφού μπορούμε να ορίσουμε παραμέτρους στυλ, αλλάζοντας την τιμή τους.

```
element.style.ιδιότητα = τιμή
```

Ένα σημείο που πρέπει να προσέξουμε όμως είναι η έκφραση των ιδιοτήτων στυλ της CSS ως μεταβλητών JavaScript. Επειδή στην CSS έχουμε συχνά μεταβλητές που περιέχουν τον χαρακτήρα παύλα, πχ `font-size`, `background-color`, κλπ, αυτές οι μεταβλητές δεν επιτρέπονται στην JavaScript, αφού ο χαρακτήρας `"-"` δεν επιτρέπεται στο όνομα μεταβλητής. Μία σύμβαση που ισχύει είναι να αναφερόμαστε στην ιδιότητα `font-size` της CSS ως `fontSize` στην JavaScript.

```
element.style.fontSize = "2rem";
```

1.9.5 Διαπέραση δένδρου DOM

Μια άλλη δυνατότητα που μάς παρέχει η διεπαφή DOM είναι να διαπεράσουμε τα στοιχεία του δένδρου.

Ας δούμε κατ' αρχάς τους όρους που θα δούμε στην προγραμματιστική διεπαφή του DOM, είναι όροι που συναντώνται γενικότερα στις δομές δεδομένων δένδρου.

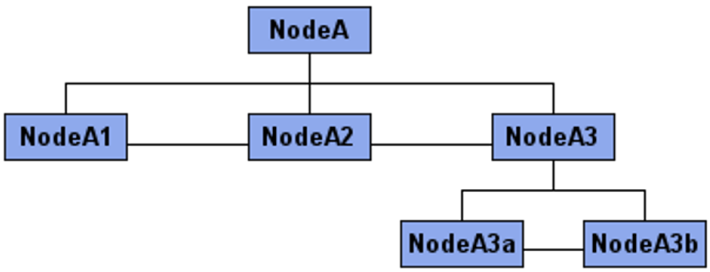
- Το **Element** αντιστοιχεί σε ένα στοιχείο.
- Το **Root** αντιστοιχεί στη ρίζα του δέντρου, το `<html>`.
- **Child** είναι το παιδί ενός κόμβου.
- **Descendant** είναι οποιοδήποτε κόμβος είτε παιδί είτε παιδί παιδιού δηλαδή οποιοσδήποτε κόμβος του υποδέντρου κάτω από ένα κόμβο.
- **Parent** είναι ο πατέρας, το γονικό στοιχείο όπως ονομάζεται.
- **Sibling** είναι τα αδέλφια, είναι άλλα στοιχεία που ανήκουν στο ίδιο επίπεδο.
- **Text** node είναι ένας κόμβος που περιέχει κείμενο.

Ακολουθούν μερικά παραδείγματα διαπέρασης του δένδρου:

- `element.firstChild` επιστρέφει το πρώτο παιδί του στοιχείου element.
- `element.lastChild` επιστρέφει το τελευταίο παιδί του στοιχείου element.
- `element.childNodes` επιστρέφει ως συλλογή τα παιδιά του element.
- `element.childNodes.length` επιστρέφει το πλήθος των παιδιών του element.
- `element.childNodes[0]` επιστρέφει το πρώτο παιδί του element, ισοδύναμο με το `element.firstChild`.
- `element.nextSibling` είναι ο επόμενος αδελφός και `element.prevSibling` ο προηγούμενος αδελφός του element.
- `element.parentNode` είναι ο πατέρας του στοιχείου element.

1.9.5.1 Άσκηση

Έστω το παρακάτω απόσπασμα από ένα δένδρο DOM.



Σχήμα 1.13: Απόσπασμα ενός δένδοου DOM.

Ζητείται να βρείτε τις απαντήσεις στα παρακάτω ερωτήματα:

1. NodeA.firstChild
2. NodeA.lastChild
3. NodeA.childNodes.length
4. NodeA.childNodes[0]
5. NodeA.childNodes[1]
6. NodeA.childNodes[2]
7. NodeA1.parentNode
8. NodeA1.nextSibling
9. NodeA3.prevSibling
10. NodeA3.nextSibling
11. NodeA.lastChild.firstChild
12. NodeA3b.parentNode.parentNode

Απάντηση:

(1) NodeA1, (2) NodeA3, (3) 3, (4) NodeA1, (5) NodeA2, (6) NodeA3, (7) NodeA, (8) NodeA2, (9) NodeA2, (10) null, (11) NodeA3a, (12) NodeA

1.9.6 Εισαγωγή-διαγραφή στοιχείων του DOM

Μια ακόμη χρήσιμη δυνατότητα που μάς παρέχει η διεπαφή του DOM είναι αυτή της δημιουργίας και διαγραφής στοιχείων του δένδρου.

Αυτή η δυνατότητα υποστηρίζεται από τις μεθόδους `createElement()` που δημιουργεί ένα νέο στοιχείο, και στη συνέχεια από τη μέθοδο `element.appendChild()`, για προσθήκη του στοιχείου στην ιεραρχία, ή τη μέθοδο `element.removeChild()` για διαγραφή στοιχείου από την ιεραρχία.

Ας δούμε ένα παράδειγμα. Έστω ότι θέλουμε να δημιουργήσουμε ένα κόμβο `<p>` κάτω από τον κόμβο `<section>` ενός εγγράφου HTML.

```
const sect = document.querySelector('section');
const para = document.createElement('p');
para.textContent = 'Ευχαριστούμε για την επίσκεψη!';
sect.appendChild(para);
```

Παρατηρούμε στο παράδειγμα αυτό ότι αρχικά στην μεταβλητή sect εκχωρήσαμε το στοιχείο `<section>` του εγγράφου (αν υπήρχαν πολλά στοιχεία αυτού του τύπου, μάς επιστρέφει το πρώτο από αυτά), στη συνέχεια δημιουργούμε ένα νέο στοιχείο (μεταβλητή para) του δίνουμε ένα περιεχόμενο κείμενο, και τέλος το επισυνάπτουμε στο στοιχείο sect με τη μέθοδο `appendChild()`.

Ας δούμε ένα ακόμη παράδειγμα, διαγραφής του πρώτου παιδιού ενός κόμβου του δένδρου.

```
const sect = document.querySelector('section');
sect.removeChild(sect.childNodes[0]);
```

Να σημειωθεί οτι η μέθοδος `removeChild()` επιστρέφει ένα αντικείμενο τύπου Node μετά την επιτυχή διαγραφή του στοιχείου, ή null αν το στοιχείο δεν βρέθηκε στο δένδρο.

1.9.6.1 Άσκηση

Να εισάγετε ως πρώτο παιδί του στοιχείου με ταυτότητα id = ``myID'' έναν υπερσύνδεσμο με κείμενο *Πανεπιστήμιο Πατρών* προς την ιστοσελίδα του Πανεπιστημίου Πατρών.

Απάντηση:

```
const el = document.querySelector('#myID');
const link = document.createElement('a');
link.textContent = 'Πανεπιστήμιο Πατρών';
link.href = 'https://www.upatras.gr';
el.appendChild(link)
```

Αυτά ως μία σύντομη εισαγωγή στη διεπαφή προς το DOM και μία ιστοσελίδα. Οι μέθοδοι που είδαμε σε αυτή την ενότητα και ιδιαίτερα η `querySelector` θα είναι το βασικό μας εργαλείο για την επικοινωνία ανάμεσα σε ένα πρόγραμμα JavaScript και την ιστοσελίδα με την οποία θέλει να επικοινωνήσει.

1.10 Σύνοψη

Στο κεφάλαιο αυτό έγινε η πρώτη γνωριμία με τη γλώσσα προγραμματισμού JavaScript. Αφού κάναμε μια ιστορική ανασκόπηση της εξέλιξης της γλώσσας, είδαμε βασικά στοιχεία σύνταξης της γλώσσας, οπως τους τρόπους δήλωσης μεταβλητών και τους βασικούς τελεστές που χρησιμοποιούνται σε μαθηματικές εκφράσεις. Στη συνέχεια είδαμε τρόπους για να ενσωματώσουμε κώδικα JavaScript σε μια ιστοσελίδα. Είδαμε πώς η JavaScript έχει πρόσβαση στις ιδιότητες του global object (window) και στο DOM (document). Στο επόμενο κεφάλαιο θα προχωρήσουμε με τους βασικούς τύπους δεδομένων και τις κύριες εντολές της JavaScript.

2 Πρωτογενείς τύποι δεδομένων και εντολές της JavaScript

Στο δεύτερο αυτό κεφάλαιο με αντικείμενο την εισαγωγή στη γλώσσα προγραμματισμού JavaScript, θα ξεκινήσουμε την επισκόπηση των διαφορετικών πρωτογενών τύπων δεδομένων που υποστηρίζει η γλώσσα. Συγκεκριμένα θα δούμε τους επτά πρωτογενείς τύπους δεδομένων (primitive data types). Επίσης στο κεφάλαιο αυτό θα γίνει η εισαγωγή στον πυρήνα των εντολών της JavaScript που υλοποιούν δομές ελέγχου ροής και επανάληψης που διαθέτει η γλώσσα.

2.1 Τύποι δεδομένων

Υπάρχουν οι εξής πρωτογενείς (primitive) τύποι δεδομένων:

Οι τύποι **number, string, boolean** που συναντιώνται στις περισσότερες γλώσσες προγραμματισμού.

Ακόμη ορίζονται οι ειδικοί τύποι **null** και **undefined** που θα περιγραφούν στη συνέχεια.

Επίσης σε τελευταίες εκδόσεις της γλώσσας έχουν προστεθεί οι τύποι **bigInt** για αναπαράσταση μεγάλων ακέραιων αριθμών μεγαλύτερων από 2^{53}, καθώς και ο τύπος **symbol**.

Εκτός των πρωτογενών τύπων δεδομένων, υπάρχουν οι τύποι **δεδομένων αναφοράς**. Στην κατηγορία αυτή είναι ο τύπος δεδομένων που αφορά όλα τα δεδομένα πλην των ανωτέρω, το **αντικείμενο (object)**. Ένα αντικείμενο της JavaScript είναι μια μη ταξινομημένη συλλογή από ζευγάρια *κλειδιών:τιμών*. Η γλώσσα ορίζει επίσης ένα ειδικό είδος αντικειμένου, τον **πίνακα (array)**, που αντιπροσωπεύει μια διαταγμένη συλλογή αριθμημένων τιμών.

Οι πρωτογενείς τύποι δεδομένων θα συζητηθούν στο παρόν κεφάλαιο, ενώ οι πίνακες και τα αντικείμενα θα συζητηθούν μαζί με τις συναρτήσεις στο επόμενο κεφάλαιο.

https://javascript.info/primitives-methods

2.2 Πρωτογενείς τύποι δεδομένων: Number

Ο τύπος δεδομένων **Number** αφορά τιμές ακεραίων με τιμές $\pm 2^{53}$ και κλασματικών αριθμών μεγέθους μέχρι $\pm 1.7976931348623157 \times 10^{308}$ και μικρών τιμών μέχρι $\pm 5 \times 10^{-324}$. Οι κλασματικοί αριθμοί αναπαρίστανται με

κωδικοποίηση 64-bit IEEE 754, όπως στις περισσότερες σύγχρονες γλώσσες προγραμματισμού (αντίστοιχος τύπου double της Java, C++, float της Python, κλπ).

Για αναπαράσταση πολύ μεγάλων ακέραιων αριθμών εκτός του παραπάνω όριου, από την έκδοση ES2020 έχει εισαχθεί ο τύπος **BigInt** ο οποίος αναπαριστά τον αριθμό ως ακολουθία ψηφίων και δεν έχει άνω όριο. Όμως η υλοποίηση του δεν έχει ολοκληρωθεί στους φυλλομετρητές ακόμη.

Οι τιμές μεταβλητών τύπου Number μπορεί να είναι είτε ακέραιοι, είτε κλασματικοί αριθμοί, είτε αριθμοί σε δεκαεξαδική μορφή, σε δυαδική μορφή, σε μορφή ύψωσης σε δύναμη, κλπ. Παραδείγματα ακολουθούν:

```
let x = 12345 // ακέραιος
let y = 123.456 // δεκαδικός
let z = 1.473E-32 // δύναμη 1.473 × 10⁻³²
let k = 0xBADCAFE // δεκαεξαδικός => 195939070
```

Υπάρχει ακόμη η δυνατότητα από την έκδοση ES6 και μετά, αναπαράστασης οκταδικών αριθμών (αρχίζουν με 0o) και δυαδικών αριθμών (αρχίζουν με 0b).

Με χρήση του τελεστή `typeof` μπορούμε να ελέγξουμε τον τύπο μιας μεταβλητής. Σε όλες τις παραπάνω περιπτώσεις η έκφραση για παράδειγμα `typeof x` επιστρέφει τη συμβολοσειρά `'number'`.

Όπως έχει ήδη αναφερθεί στους αριθμούς μπορούμε να εφαρμόσουμε τελεστές αριθμητικών πράξεων, `+, -, *, /, %, **`, καθώς και τους τελεστές `++, --, +=, -=`, κλπ.

Το global object διαθέτει το αντικείμενο `Math` το οποίο έχει μεθόδους για πιο σύνθετες μαθηματικές συναρτήσεις.

Οι κυριότερες από αυτές είναι:

- `Math.abs(-20)` // 20 απόλυτη τιμή
- `Math.ceil(10.5)` // 11 ο επόμενος ακέραιος
- `Math.E` // 2.718281828459045 η σταθερά e
- `Math.exp(5)` // e**5 = 148.413, δύναμη του e
- `Math.floor(8.9)` // 8 το ακέραιο μέρος
- `Math.log(3)`, // 1.0986122886681096
- `Math.max(10,20)` // 20 , μέγιστη τιμή
- `Math.min(10,20)` // 10 ελάχιστη τιμή,
- `Math.pow(3,4)` // 81 ύψωση του x στην δύναμη y, x ** y
- `Math.round(5.6)` // 6 στρογγύλεμα στον πλησιέστερο ακέραιο
- `Math.sqrt(4)` //2, τετραγωνική ρίζα
- `Math.PI` // π= 3.14

- `Math.sin(θ)` // ημίτονο
- `Math.cos(θ)` // συνημίτονο
- `Math.tan(θ)` // εφαπτομένη

Θα πρέπει να σημειθεί ότι η γωνία θ στις τριγωνομετρικές συναρτήσεις εκφράζεται σε rad, δηλαδή 90° = Math.PI/2, συνεπώς: Math.cos(Math.PI/2) = 0.

Επίσης το αντικείμενο Math διαθέτει αρκετές ακόμη χρήσιμες συναρτήσεις, μεταξύ των οποίων η `Math.random()`, η οποία όταν κληθεί επιστρέφει ένα ψευδο-τυχαίο δεκαδικό αριθμό στο διάστημα από 0 μέχρι 1 χωρίς να περιλαμβάνεται το 1.

Αντίθετα με άλλες γλώσσες δεδομένων, στην JavaScript αν μια μαθηματική έκφραση προκαλέσει υπερχείλιση, (overflow), όπως για παράδειγμα η διαίρεση με το μηδέν, δεν προκύπτει σφάλμα, αλλά το αποτέλεσμα της έκφρασης μπορεί να πάρει μια από τις παρακάτω ειδικές τιμές:

- `Infinity` (θετικό άπειρο)
- `-Infinity` (αρνητικό άπειρο)
- `NaN` όχι αριθμητική τιμή Not-a-Number

Μάλιστα το global object διαθέτει συναρτήσεις για έλεγχο αυτών των τιμών, όπως `isNaN(x)`.

Για παράδειγμα:

```
console.log(-10/0);
> -Infinity
```

επίσης:

```
isNaN("αβγ");
> true
```

2.3 Πρωτεγενείς τύποι δεδομένων: Συμβολοσειρές

Οι συμβολοσειρές (string) είναι ο τύπος δεδομένων που αναπαριστά ακολουθία χαρακτήρων. Οι συμβολοσειρές στην JavaScript περιέχουν χαρακτήρες Unicode, 16bit ο καθένας. Οριοθετούμε συμβολοσειρές με απλά ' ή διπλά " εισαγωγικά, ή με τον χαρακτήρα *backquote* `. Χρησιμοποιούμε το ένα για να οριοθετήσουμε συμβολοσειρά που περιέχει το άλλο.

Ο χαρακτήρας backquote χρησιμοποιείται για συμβολοσειρές που μπορούν να περιέχουν εκφράσεις JavaScript, όπως θα εξηγηθεί στη συνέχεια.

Ο χαρακτήρας διαφυγής μπορεί να χρησιμοποιηθεί για να ακυρώσει τον ειδικό ρόλο κάποιου χαρακτήρα, όπως του χαρακτήρα οριοθέτησης της συμβολοσειράς.

Για παράδειγμα:

```
console.log("του είπαν: \"τι χαμπάρια μάς φέρνεις;\"")


>του είπαν: "τι χαμπάρια μάς φέρνεις;"
```

Ο τελεστής ``+'' επιτρέπει την συνένωση συμβολοσειρών.

```
let one = 'Καλή σας μέρα ';
let two = 'άρχοντες! ';
console.log(one + two);
>Καλή σας μέρα άρχοντες!
```

Στο επόμενο παράδειγμα, όταν ο χρήστης πατήσει το πλήκτρο, προκύπτει παράθυρο με ερώτημα για το όνομά του και ακολουθεί χαιρετισμός με σύνθεση συμβολοσειρών.

```
<button> ok </button>
let button = document.querySelector('button');
button.onclick = function() {
    let name = prompt(Πώς σε λένε;');
    alert('Καλωσήρθες ' + name); }
```

Ένα αξιοπρόσεκτο χαρακτηριστικό της γλώσσας, ένδειξη της αδύναμης υποστήριξης τύπων μεταβλητών είναι οι πράξεις μεταξύ συμβολοσειρών και αριθμών:

```
10 + '20'
> "1020"
10+20
> 30
'abc'+10
> "abc10"
```

2.3.1 Δείκτης συμβολοσειράς

Μπορούμε να αναφερθούμε σε επί μέρους στοιχεία μιας συμβολοσειράς με χρήστη δεικτών που αρχίζουν από 0, μέχρι length-1 όπου length το πλήθος χαρακτήρων της συμβολοσειράς.

Για παράδειγμα:

```
const hero = "Αθανάσιος Διάκος";
console.log(hero[1]);
> 'θ'
```

Το μήκος της συμβολοσειράς μπορούμε να το πάρουμε με χρήση της ιδιότητας length :

```
const hero = "Αθανάσιος Διάκος";
console.log(hero.length);
> 16
```

Επειδή η συμβολοσειρά hero περιέχει 16 χαρακτήρες.

Μια λεπτομέρεια που θα πρέπει να προσέξουμε είναι ότι αν η συμβολοσειρά περιέχει σύμβολα που συντίθενται από ακολουθίες Unicode 16bit, η ιδιότητα length θα μετρήσει το πλήθος αυτών των χαρακτήρων. Για παράδειγμα emoji σημαιών χωρών:

```
const greekFlag = "🇬🇷";
greekFlag.length;
> 4
```

Θα πρέπει όμως να σημειωθεί ότι κείμενα στις περισσότερες γλώσσες, συμπεριλαμβάνομένων των Ευρωπαϊκών γλωσσών αλλά και των ιδεογραμμάτων των ασιατικών γλωσσών περιλαμβάνονται στην κωδικοποίηση Unicode 16bit, συνεπώς το μήκος της συμβολοσειράς μπορεί να μετρηθεί.

```
const hello = "你好" // Nǐ hǎo (γεια σας στα Κινέζικα)
hello.length;
> 2
```

2.3.2 Μετατροπές από συμβολοσειρές σε αριθμούς

Για την μετατροπή μιας συμβολοσειράς με αριθμητικούς χαρακτήρες σε αριθμό μπορούμε να χρησιμοποιήσουμε τον δημιουργό του αντικειμένου `Number()`.

```
// μετατροπή από string σε αριθμό
let myString = '123';
let myNum = Number(myString);
typeof myNum;
>"number"
```

Αν το όρισμα του δημιουργού `Number()` δεν εκφράζει ένα αριθμό, τότε η συνάρτηση επιστρέφει ένα αντικείμενο τύπου `number` με την τιμή `NaN`.

Εναλλακτικά μπορούμε να χρησιμοποιήσουμε τις συναρτήσεις `parseInt()` και `parseFloat()` που ανήκουν στο global object, για μετατροπή συμβολοσειράς σε ακέραιο ή κλασματικό αριθμό αντίστοιχα.

Θα πρέπει να προσέξουμε μια ιδιαιτερότητα των συναρτήσεων αυτών.Αν η συμβολοσειρά αρχίζει με αριθμητικούς χαρακτήρες και στη συνέχεια περιέχει μη αριθμητικούς χαρακτήρες, αγνοεί τους μη αριθμητικούς και επιστρέφει τον αριθμό που προκύπτει, και όχι NaN που επιστρέφει η δημιουργός Number().

```
Number('123abc'); \\ NaN
parseInt('123abc'); \\ 123
```

Η αντίστροφη μετατροπή από αριθμό στην αντίστοιχη συμβολοσειρά γίνεται με κλήση της μεθόδου `toString()` του αντικειμένου Number.

```
// μετατροπή από αριθμό σε string
let myNum = 123;
let myString = myNum.toString();
```

2.3.3 Κύριες μέθοδοι και τελεστές του String

Έχουμε ήδη δει πώς αναφερόμαστε σε επί μέρους χαρακτήρες μια συμβολοσειράς και πώς βρίσκουμε το μήκος μια συμβολοσειράς.

```
let myName = 'Nikos';
myName.length;    // 5

//πρώτος χαρακτήρας
myName[0];     // "N"

//τελευταίος χαρακτήρας
myName[myName.length-1];  // "s"
```

Στη συνέχεια θα δούμε δύο ενδιαφέρουσες μεθόδους του αντικειμένου String, την slice() και την indexOf():

```
//τμήμα, από 0 έως 3
myName.slice(0,3); // returns 'Nik'

//βρες τη θέση της υπο-συμβολοσειράς  -1 αν δεν βρεθεί
myName.indexOf('kos');     // 2

//τεμάχιο συμβολοσειράς από θέση i1, έως i2
myName.slice(3); // επιστρέφει 'os'
myName.slice(1,4); // επιστρέφει 'iko'
```

Όπως φαίνεται από τα παραδείγματα, μπορούμε να αναφερθούμε σε τμήμα της συμβολοσειράς από τη θέση index1 μέχρι τη θέση index2 χωρίς να περιλαμβάνεται η τελευταία με χρήση της `s.slice(index1, index2)`.

Επίσης με την μέθοδο `s.indexOf(υπο-συμβολοσειρά)` βρίσκουμε την θέση που για πρώτη φορά συναντάμε την υπο-συμβολοσειρά στην s. Αν αυτή δεν υπάρχει τότε μάς επιστρέφει την τιμή -1.

Μπορούμε να διαχωρίσουμε μια συμβολοσειρά σε ένα πίνακα των επί μέρους χαρακτήρων της, με χρήση της μεθόδου split(), η μέθοδος μοιάζει με την αντίστοιχη της Python, με μια διαφορά ότι πρέπει να δώσουμε οπωσδήποτε όρισμα, για παράδειγμα:

```
//διαχωρισμός συμβολοσειράς σε πίνακα στοιχείων
//split(delimiter_string)
```

```
'αβγ'.split('');
> [ 'α', 'β', 'γ' ]
'αβγ'.split('β');
> [ 'α', 'γ' ]
```

Άλλες χρήσιμες μέθοδοι του αντικειμένου String είναι:

```
myName.toLowerCase(); // "nikos"
myName.toUpperCase(); // "NIKOS"

myName.replace('kos','na') //"Nina"

myName.charAt(2); // "k"

myName.charCodeAt(2); // 107
```

Επίσης η μέθοδος trim() αποκόπτει τα κενά στην αρχή και στο τέλος μιας συμβολοσειράς με αντίστοιχο τρόπο όπως η μέθοδος strip() στην Python:

```
"       καλή σας μέρα           ".trim();
> 'καλή σας μέρα'
```

2.3.4 Συμβολοσειρές-πρότυπα (template literals)

Ένας ειδικός τύπος συμβολοσειρών εισήχθη στην JavaScript στην έκδοση ES6. Είναι οι **συμβολοσειρές πρότυπα**, αυτές έχουν το χαρακτηριστικό ότι μπορούν να περιλάβουν εκφράσεις JavaScript `${έκφραση}`, οι οποίες αντικαθίστανται από την τιμή τους.

Οι συμβολοσειρές αυτές οριοθετούνται από το σύμβολο backquote (`) .

Ας δούμε ένα παράδειγμα.

```
let a = 5;
let b = 10;
console.log(`a+b=${a+b} ενώ 2a+b=${2*a + b}.`);
>'a+b=15 ενώ 2a+b=20.'
```

2.3.5 Ασκήσεις

2.3.5.1 Άσκηση 1

Έστω

```
let input = "ΘεσΑΛΛΟνίκη";
```

Να γράψετε εντολές JavaScript που παράγουν σωστή συμβολοσειρά για το όνομα της πόλης.

Απάντηση

```
let temp = input.toLowerCase();
let startCharacter = temp[0].toUpperCase();
let result = startCharacter + temp.slice(1);
```

2.3.5.2 Άσκηση 2

Έστω

```
let input = "ATH675847583748sjt567654;Athens El.Venizelos";
```

Ζητείται να γράψετε κώδικα που εξάγει από την παραπάνω συμβολοσειρά την εξής (τους τρείς πρώτους χαρακτήρες και την υπόλοιπη συμβολοσειρά μετά το ;): `ATH:Athens El.Venizelos`

Απάντηση

```
console.log(input.slice(0,3)+ ":" + input.slice(input.indexOf(";")+1))
> 'ATH:Athens El.Venizelos'
```

2.4 Πρωτογενείς τύποι δεδομένων: null και undefined

Στην ενότητα αυτή θα δούμε δύο ακόμη πρωτογενείς τύπους δεδομένων, ειδικού σκοπού.

Ο πρώτος τύπος είναι το `null`, μια λέξη-κλειδί που υποδεικνύει την απουσία τιμής. Η χρήση του τελεστή typeof στο null επιστρέφει τη συμβολοσειρά ``object'', υποδεικνύοντας ότι το `null` μπορεί να θεωρηθεί ως μια ειδική τιμή αντικειμένου που υποδηλώνει την απουσία αντικειμένου. Σύμφωνα με τον ορισμό της γλώσσας, το `null` δεν θεωρείται αντικείμενο αλλά το μοναδικό μέλος ενός ξεχωριστού τύπου δεδομένων που μπορεί να χρησιμοποιηθεί για να δείξει ``μη τιμή'' για αριθμούς και συμβολοσειρές καθώς επίσης για αντικείμενα.

Έστω σε μια ιστοσελίδα χωρίς υπερσυνδέσμους (χωρίς στοιχεία `<a>`), ας δούμε τι θα συμβεί αν αναζητήσουμε στοιχεία αυτού του τύπου στο DOM:

```
let x = document.querySelector("a");
console.log(x);
> null
typeof x;
> "object"
```

Υπάρχει όμως ένας ακόμη τύπος δεδομένων που υποδηλώνει μη τιμή: ο τύπος `undefined`.

Ο τύπος αυτός αποδίδεται σε μεταβλητές που έχουν οριστεί χωρίς να λάβουν τιμή, σε συναρτήσεις που δεν επιστρέφουν δεδομένα (δεν έχουν εντολή return), σε ιδιότητες αντικειμένων που δεν υπάρχουν, σε τιμές ορισμάτων

συναρτήσεων που δεν τους έχει δοθεί τιμή. Η τιμή αυτή επιστρέφει ως τύπος δεδομένων από τον τελεστή typeof. Ας δούμε μερικά παραδείγματα:

```
let x; // μεταβλητή χωρίς αρχική τιμή
typeof x;
> 'undefined'

const ob = {name: "Nikos"} // αντικείμενο
typeof ob.age; // ιδιότητα που δεν υπάρχει
> 'undefined'

function f(){} // συνάρτηση που δεν επιστρέφει τιμή
typeof f();
> 'undefined'
```

2.5 Πρωτογενείς τύποι δεδομένων: Boolean

Ο τύπος δεδομένων Boolean αντιπροσωπεύει λογικές μεταβλητές οι οποίες παίρνουν τιμή αληθές/ψευδές. Οι δεσμευμένες λέξεις `true` και `false` αντιστοιχούν στις τιμές αυτές αντίστοιχα.

Η τιμή αληθές/ψευδές είναι το αποτέλεσμα μιας σύγκρισης, η οποία εισάγεται με τον τελεστή σύγκρισης `===` (ισότητα τιμής και τύπου δεδομένων) ή `==` (ισότητα μόνο τιμής, όχι απαραίτητα και τύπου δεδομένων), ή και άλλων τελεστών σύγκρισης `>`, `<`, `>=`, `<=`, `!==` (έλεγχος ανισότητας τιμής και τύπου δεδομένων), `!=` (έλεγχος ανισότητας τιμής).

Εκφράσεις που περιέχουν σύγκριση συνήθως χρησιμοποιούνται σε εντολές που απαιτούν τιμή αληθές/ψευδές, όπως η εντολή if ή η εντολή while.

Όμως θα πρέπει να σημειωθεί ότι οποιαδήποτε έκφραση της JavaScript ανάλογα με την τιμή που παίρνει, αντιστοιχεί σε αληθές/ψευδές. Ο κανόνας που ισχύει είναι ο εξής: Αν η έκφραση έχει την τιμή: `undefined`, `null`, `0`, `-0`, `NaN`, `""` (κενή συμβολοσειρά), τότε είναι ισοδύναμη με `false`. Σε οποιαδήποτε άλλη περίπτωση είναι `true`.

Οι λογικοί τελεστές της JavaScript είναι:

- `&&` τελεστής AND (και)
- `||` τελεστής OR (ή)
- `!` τελεστής NOT (όχι)

2.5.1 Τιμή λογικών εκφράσεων

Ας δούμε τι επιστρέφουν οι λογικές εκφράσεις, κάτι που συχνά εκμεταλλεύονται οι προγραμματιστές JavaScript για να ελέγξουν τιμή σε μεταβλητές ή να δώσουν μια προκαθορισμένη τιμή σε μια μεταβλητή αν αυτή δεν έχει ήδη οριστεί.

Μια έκφραση λογικής OR, θα λάβει την τιμή της πρώτης μεταβλητής που είναι true, επειδή η λογική OR ικανοποιείται σε αληθές αρκεί ένας όρος να είναι αληθής, ή την τελευταία τιμή αν κανένας όρος δεν είναι αληθής.

Παραδείγματα εκφράσεων OR:

- `0 || undefined || 5` : θα πάρει την τιμή 5 (true)
- `0 || 10 || 5` : θα πάρει την τιμή 10 (true)
- `0 || undefined || null` : θα πάρει την τιμή null (false)

Αντίστοιχα σε μια έκφραση με λογική AND Θα επιστρέψει τον τελευταίο όρο, αν όλοι είναι αληθείς και τον πρώτο ψευδή όρο αν υπάρχει ψευδής όρος. Αυτό γιατί στη λογική AND αρκεί ένας όρος να είναι ψευδής για να πάρει η έκφραση την τιμή *ψευδής*.

Παραδείγματα εκφράσεων AND:

- `0 && 10 && 5` : θα πάρει την τιμή 0 (false)
- `8 && 10 && 5` : θα πάρει την τιμή 5 (true)
- `5 && 10 && null` : θα πάρει την τιμή null (false)

2.6 Προγραμματιστικές δομές της JavaScript

Στη συνέχεια, θα δούμε τις βασικές προγραμματιστικές δομές που υποστηρίζει η JavaScript, που υλοποιούνται ως εντολές για έλεγχο της ροής του προγράμματος όπως η `if` και `switch`, εντολές που επιτρέπουν την επαναληπτική εκτέλεση ενός κώδικα, όπως οι `for` και `while`, και τέλος εντολές που επιτρέπουν τη μετάβαση σε άλλο τμήμα του κώδικα, όπως η `break`, `continue` και η `throw`.

- Θα πρέπει να σημειώσουμε ότι η JavaScript συντακτικά ακολουθεί το παράδειγμα πολλών άλλων γλωσσών προγραμματισμού, όπως η C, C++, Java, ως προς τη σύνταξη των εντολών αυτών, τη χρήση του `";"` ως τερματικού εντολής και τη χρήση των άγκιστρων `{ ... }` για τον ορισμό ενός μπλοκ εντολών.
- Θα πρέπει εδώ να σχολιάσουμε όμως ότι οι ομοιότητες μεταξύ JavaScript και Java σταματούν εδώ, αφού πρόκειται για δύο πολύ διαφορετικές τεχνολογίες και το όνομα JavaScript είναι ατυχές αφού παραπέμπει στην Java, με την οποία η JavaScript δεν έχει ιδιαίτερη σχέση.
- Αν ένα μπλοκ εντολών περιέχει μια μόνο εντολή, τότε τα άγκιστρα μπορούν να παραληφθούν. Όμως πολλοί προγραμματιστές βάζουν πάντα τα άγκιστρα ακόμη και στην περίπτωση του μπλοκ με μια εντολή, για να γίνει πιο σαφής η εμβέλεια κάθε εντολής.
- Επίσης μπορούμε να ορίσουμε ως κενή εντολή την εντολή `;`.
- Ως προς τη στοίχιση των εντολών, αυτή δεν είναι υποχρεωτική, όπως γίνεται για παράδειγμα στην Python, όμως είναι πολύ καλή πρακτική ένα μπλοκ εντολών να στοιχίζεται μέσα στα άγκιστρα που το ορίζουν, κάτι που επεξεργαστές κώδικα όπως ο *VS Code* κάνουν αυτόματα.

2.6.1 Εντολή `if-else`

Η πρώτη δομή που θα δούμε επιτρέπει την υπό-συνθήκη εκτέλεση ενός τμήματος του κώδικα, είναι η εντολή `if` που συναντάται σε όλες τις γλώσσες προγραμματισμού.

Στη γενική περίπτωση αυτή η δομή συντάσσεται ως εξής:

```
if (έκφραση){
    μπλοκ-εντολών-1
}
else {
    μπλοκ-εντολών-2
}
```

Το `μπλοκ-εντολών-1` θα εκτελεστεί αν η `έκφραση` είναι αληθής, ενώ το `μπλοκ-εντολών-2` αν είναι ψευδής.

Θα πρέπει να παρατηρήσουμε εδώ ότι η `έκφραση` θα πρέπει να παίρνει μια τιμή η οποία να αντιστοιχεί είτε σε αληθή, είτε σε ψευδή τιμή, σύμφωνα με τους κανόνες αληθότητας/ψευδότητας που έχουν ήδη αναφερθεί. Υπενθυμίζουμε ότι οι τιμές `undefined`, `null`, 0, -0, `NaN`, `""` (κενή συμβολοσειρά), `false` αντιστοιχούν στην τιμή **``ψευδής''** ενώ οποιαδήποτε άλλη τιμή σε **``αληθής''**. Για παράδειγμα ένας κενός πίνακας [ ] αντιστοιχεί στην τιμή ``αληθής'', το ίδιο και ένα κενό αντικείμενο { }, κάτι που διαφοροποιεί την JavaScript από την Python που η κενή λίστα [ ] αντιστοιχεί στην τιμή ``ψευδής''.

Επίσης θα πρέπει να παρατηρήσουμε ότι το τμήμα else της παραπάνω δομής μπορεί να παραληφθεί:

```
if (έκφραση){
    μπλοκ-εντολών-1
}
```

Ακόμη μπορούμε να κάνουμε διαδοχικούς ελέγχους εισάγοντας στο τμήμα else ένα νέο έλεγχο if:

```
if (συνθήκη1){
    μπλοκ-εντολών-1 // αν συνθήκη1 αληθής
} else if (συνθήκη2) {
    μπλοκ-εντολών-2 // αν όχι συνθήκη1 και συνθήκη2 αληθής
} else if (συνθήκη3) {
    μπλοκ-εντολών-3 // αν όχι συνθήκη1, ούτε συνθήκη 2 και συνθήκη3 αληθής
} else {
    μπλοκ-εντολών-4 // αν όχι συνθήκη1, ούτε συνθήκη2, ούτε συνθήκη3
}
```

Ας δούμε ένα παράδειγμα:

```
let forecast = 'snowing';

if (forecast === 'sunny') {
```

```
  console.log('Πάμε για μπάνιο.');
} else if (forecast === 'rainy') {
  console.log('Να πάρουμε ομπρέλα');
} else if (forecast === 'snowing') {
  console.log('Να πάμε για σκι');
} else {
  console.log('μένουμε σπίτι');
}
> 'Να πάμε για σκι'
```

Στο παράδειγμα η μεταβλητή forecast ελέγχεται για την τιμή της και ανάλογα τυπώνεται ένα μήνυμα στην κονσόλα. Αν η τιμή της μεταβλητής είναι διαφορετική από τις τιμές που ελέγχονται διαδοχικά στη δομή αυτή, πχ `forecast="cloudy"` θα πάρουμε το μήνυμα που τυπώνεται από το σκέλος `else` που ακολουθεί τους ελέγχους.

2.6.2 Εντολή `switch`

Αν πρέπει να κάνουμε μια σειρά από ελέγχους οι οποίοι αφορούν την ίδια μεταβλητή, όπως στο προηγούμενο παράδειγμα, μια καλύτερη επιλογή από διαδοχικά if - else if είναι να χρησιμοποιηθεί η δομή `switch`.

Η εντολή αυτή συντάσσεται ως εξής:

```
switch(n) {
case 1:  // if n === 1
    block-εντολών-1
    break;
case 2: // if n === 2
    block-εντολών-2
    break;
case 3: // if n === 3
    block-εντολών-3
    break;
default: // αν δεν ισχύει καμιά από τις περιπτώσεις
    block-εντολών-4
    break;
}
```

Σύμφωνα με το πρότυπο αυτό το προηγούμενο παράδειγμα θα διαμορφωθεί ως ακολούθως:

```
switch (forecast) {
  case 'sunny':
    console.log('Πάμε για μπάνιο.');
    break;
  case 'rainy':
    console.log('Να πάρουμε ομπρέλα.');
```

```
    break;
  case 'snowing':
    console.log('Να πάμε για σκι.');
    break;
  default:
    console.log('μένουμε σπίτι');
}
```

Να σημειωθεί εδώ ότι οι έλεγχοι που γίνονται σε κάθε case είναι τύπου strict equality "===".

2.6.3 Υπό συνθήκη εκχώρηση τιμής (τριαδικός τελεστής)

Ένας εναλλακτικός τρόπος να εκχωρηθεί διαφορετική τιμή σε μια μεταβλητή ανάλογα με την τιμή μιας έκφρασης είναι με χρήση του τριαδικού τελεστή `a ? b : c`.

Ο τελεστής αυτός, που λέγεται τριαδικός επειδή παίρνει τρεις παραμέτρους, είναι συντομογραφία μας εντολής if-else, όπου a είναι μια έκφραση που παίρνει τιμή αληθής/ψευδής, b η τιμή αν η έκφραση είναι αληθής, και c η τιμή αν η έκφραση είναι ψευδής.

```
(έκφραση) ? τιμή-αν-αληθής : τιμή-αν-ψευδής
```

Ας δούμε ένα παράδειγμα:

Έστω ότι η μεταβλητή result εκφράζει αν ένας φοιτητής πήρε προβιβάσιμο βαθμό ή όχι. Παίρνει την τιμή ``επιτυχία'' αν ο βαθμός είναι μεγαλύτερος ή ισος του 5 και την τιμή ``αποτυχία'' αν ο βαθμός είναι μικρότερος του 5.

Το παρακάτω εντολή if-else αποδίδει τιμή στη μεταβλητή result, ανάλογα με την τιμή της vathmos:

```
let result;
if (vathmos < 5){
    result = "αποτυχία";
}
else {
    result = "επιτυχία";
}
```

Η παραπάνω λογική μπορεί να απλοποιηθεί όμως με χρήση του τριαδικού τελεστή ως εξής:

```
let result = (vathmos<5) ? "αποτυχία" : "επιτυχία";
```

2.6.4 Διαχείριση σφαλμάτων με try/catch

Η JavaScript διαθέτει μηχανισμό για διαχείριση σφαλμάτων ή *εξαιρέσεων (exceptions)* όπως λέγονται. Ο μηχανισμός αυτός έχει πολλά κοινά με την εντολή if/else. Πρόκειται για την εντολή `try/catch/finally`.

Η σύνταξη της εντολής αυτής είναι:

```
try {
    // κώδικας στον οποίο γίνεται έλεγχος εξαίρεσης
}
catch (e) {
    // κώδικας που θα τρέξει αν συμβεί εξαίρεση
}
finally {
    // κώδικας που θα τρέξει σε κάθε περίπτωση
}
```

Η εντολή catch ακολουθείται από ένα μπλοκ κώδικα στον οποίο γίνεται έλεγχος για κάποια εξαίρεση (απρόβλεπτη κατάσταση σφάλματος), αν αυτή συμβεί τότε θα εκτελεστεί ο κώδικας που ακολουθεί την εντολή catch(e). Η παράμετρος e στο catch περιέχει στοιχεία για το σφάλμα, είτε κάποιο μήνυμα, ή κωδικό σφάλματος. Το τμήμα finally είναι προαιρετικό.

Ένα παράδειγμα:

```
try {
    function factorial(num){
        if (num === 0) return 1;
        else return num * factorial( num - 1 );
    }
    let n = Number(prompt("Δώστε θετικό αριθμό", ""));
    let f = factorial(n);
    alert(n + "! = " + f);
}
catch(ex) {
    alert(ex);
}
```

Στο παράδειγμα αυτό, μέσα στο τμήμα `try` ορίζουμε συνάρτηση αναδρομικού υπολογισμού του παραγοντικού n! = n*(n-1)*(n-2) ... * 1 Στο τμήμα αυτό του κώδικα ζητείται από τον χρήστη να δώσει θετικό αριθμό, και το πρόγραμμα τού επιστρέφει μέσω alert την τιμή του παραγοντικού. Αν όμως ο χρήστης δεν δώσει θετικό αριθμό, ενεργοποιείται το τμήμα `catch` που στέλνει στον χρήστη μέσω alert το μήνυμα σφάλματος. Για παράδειγμα αν ο χρήστης δώσει αρνητικό αριθμό το μήνυμα είναι: `"RangeError: Maximum call stack size exceeded"`.

Μια εντολή που συνήθως εμφανίζεται στη δομή try/catch είναι η εντολή `throw`.

Η εντολή αυτή όταν εμφανίζεται σε ένα μπλοκ `try` είναι εντολή εξόδου από το μπλοκ και μετάβασης στο μπλοκ `catch`. Μοιάζει δηλαδή με την εντολή `break` που θα δούμε στους βρόχους επανάληψης στη συνέχεια.

Η σύνταξη της `throw` είναι:

```
throw έκφραση-σφάλματος;
```

Η έκφραση σφάλματος είναι είτε ένα αντικείμενο τύπου Error ή μια συμβολοσειρά που σχετίζεται με τη συγκεκριμένη εξαίρεση, για παράδειγμα:

```
if (x < 0) throw new Error("το x πρέπει να είναι θετικό");
```

Στο παράδειγμα αυτό η throw επιστρέφει ένα αντικείμενο τύπου Error στο οποίο περνάμε μια συμβολοσειρά σφάλματος.

2.7 Δομές επανάληψης

Στην ενότητα αυτή θα δούμε τις προγραμματιστικές δομές που διαθέτει η JavaScript για επαναληπτική εκτέλεση ενός τμήματος του κώδικα. Οι εντολές που επιτρέπουν την επαναληπτική εκτέλεση ενός μπλοκ κώδικα είναι οι `while`, `do/while`, `for`, `for/in`, `for/of`. Η πιο τυπική χρήση μιας δομής επανάληψης είναι όταν ζητείται να διαπεράσουμε τα στοιχεία ενός πίνακα.

2.7.1 Εντολή `while`

Η πιο απλή εντολή επαναληπτικής εκτέλεσης κώδικα, είναι η `while`, που συναντάται σε πολλές γλώσσες προγραμματισμού. Η while συντάσεται ως εξής:

```
while (συνθήκη) {
    μπλοκ-εντολών
}
```

Ο διερμηνευτής της JS μόλις φτάσει στην εντολή αυτή υπολογίζει την τιμή της *συνθήκης*. Αν είναι ψευδής, παραλείπει το μπλοκ-εντολών και προχωράει στην επόμενη εντολή του προγράμματος. Αν η συνθήκη είναι αληθής, εκτελεί το μπλοκ εντολών και επανέρχεται στην κορυφή στον εκ νέου έλεγχο της συνθήκης.

Είναι φανερό ότι αν κάτι δεν αλλάξει στο μπλοκ-εντολών σε σχέση με τη συνθήκη, και η συνθήκη παραμείνει αληθής, προκαλείται ατέρμων βρόχος επανάληψης.

Ας δούμε ένα παράδειγμα μιας εντολής while που επιτρέπει να τυπωθούν οι αριθμοί από 0 μέχρι 4.

```
let count = 0;
while(count < 5) {
    console.log(count++);
}
```

Το πρόγραμμα τυπώνει τους αριθμούς αρχίζοντας από το 0 και αυξάνοντας τον μετρητή count σε κάθε επανάληψη. Όταν τυπωθεί και ο αριθμός 4 και αυξηθεί ο μετρητής count σε 5, τότε η συνθήκη `count < 5` είναι ψευδής και τερματίζει η επαναληπτική εκτέλεση του μπλοκ εντολών.

Μια παραλλαγή της δομής αυτής είναι η `while(true)/break`. Στην περίπτωση αυτή θέτουμε ως συνθήκη μια πάντα αληθή έκφραση και ελέγχουμε την συνθήκη-εξόδου μέσα στο μπλοκ εντολών, όταν δε η συνθήκη-εξόδου ικανοποιηθεί, τότε εκτελούμε την εντολή `break` που μάς στέλνει εκτός του βρόχου. Θυμίζουμε ότι έχουμε δει την εντολή αυτή στην περίπτωση της εντολής case. Παράδειγμα του προηγούμενου προγράμματος με αυτή τη δομή είναι:

```
let count = 0;
while(true) {
    console.log(count++);
    if (count >= 5) break;
}
```

Η `break` χρησιμοποιείται γενικότερα για να μεταφέρουμε τον έλεγχο εκτέλεσης του προγράμματος σε άλλο σημείο, όπως εκτός ενός βρόχου επανάληψης. Μάλιστα μπορούμε να ορίσουμε το όνομα της εντολής στην οποία μεταφέρεται ο έλεγχος του προγράμματος, προς μια εντολή που φέρει όνομα: `όνομα: εντολή;` και να συνοδεύσουμε την εντολή break με το όνομα του στόχου: `break όνομα;` (δομή αντίστοιχη της goto που δεν συναντάται πλέον στις γλώσσες προγραμματισμού). Γενικά όμως θεωρείται όχι καλή πρακτική η χρήση της break, πλην της περίπτωσης εξόδου από ένα βρόχο επανάληψης.

Επίσης μια άλλη εντολή που σχετίζεται με επαναληπτικές δομές είναι η εντολή `continue`. Η εντολή αυτή όταν βρεθεί μέσα σε ένα μπλοκ εντολών ενός βρόχου κάνει τον διερμηνευτή του προγράμματος να παραλείψει τις υπόλοιπες εντολές του βρόχου και να μεταβεί στην επόμενη επανάληψη.

Ένα παράδειγμα χρήσης της break για έξοδο από ένα βρόχο είναι η αναζήτηση μιας τιμής target σε ένα πίνακα:

```
let i = 0;
while (i < ar.length){
    if (ar[i++] === target) break;
}
```

Θα πρέπει να σημειωθεί ότι οι εντολές `break` και `continue` μπορούν να χρησιμοποιηθούν σε όλες τις δομές επανάληψης που περιγράφονται στη συνέχεια.

2.7.2 Εντολή `do/while`

Η `do/while` είναι μια παραλλαγή της while που επίσης συναντάται σε πολλές γλώσσες προγραμματισμού (εξαίρεση η Python που δεν διαθέτει δομή do/while). Η διαφορά από τη while είναι ότι ο έλεγχος της συνθήκης γίνεται στο τέλος του βρόχου επανάληψης, και συνεπώς το μπλοκ εντολών εκτελείται τουλάχιστον μια φορά, ακόμη και αν η συνθήκη είναι ψευδής.

Η σύνταξη της do/while είναι:

```
do {
    μπλοκ-εντολών
} while (συνθήκη);
```

Η χρήση της δομής αυτής είναι πιο σπάνια από της while. Βεβαίως και εδώ ισχύει ο κίνδυνος του ατέρμονα βρόχου σε περίπτωση που η συνθήκη παραμείνει αληθής χωρίς να επηρεάζεται από το μπλοκ εντολών.

Το προηγούμενο παράδειγμα εκτύπωσης των αριθμών 0 μέχρι 4 με αυτή τη δομή είναι:

```
let count = 0;
do {
  console.log(count++);
} while (count < 5);
```

2.7.3 Εντολή for

Η δομή for, η οποία όπως θα δούμε στη συνέχεια εμφανίζεται σε διάφορες παραλλαγές, είναι λίγο πιο σύνθετη από την while αλλά χρησιμοποιείται πολύ πιο συχνά στον προγραμματισμό. Είναι μια δομή που συναντάται σε όλες τις γλώσσες προγραμματισμού.

Η πιο συνήθης έκδοση της for απαιτεί την ύπαρξη μιας βοηθητικής μεταβλητής, μετρητή. Η μεταβλητή αυτή αρχικοποιείται πριν αρχίσει η εκτέλεση των επαναλήψεων, ελέγχεται σύμφωνα με κάποια συνθήκη στην αρχή κάθε επανάληψης, και στο τέλος κάθε επανάληψης μεταβάλεται (συνήθως αυξάνει). Αυτές οι τρεις ενέργειες κωδικοποιούνται σε μια εντολή for ως εξής:

```
for (αρχικοποίηση; έλεγχος; τροποποίηση){
    μπλοκ-εντολών
}
```

Αν προσπαθήσουμε να επαναλάβουμε το παράδειγμα των προηγούμενων ενοτήτων, να τυπώσουμε δηλάδη τους αριθμούς από 0 .. 4, ακολουθεί η υλοποίησή του με χρήση της δομής for:

```
for(let count = 0; count < 5; count++) {
    console.log(count);
}
```

Η πιο συνήθης χρήση της δομής αυτής είναι στη διαπέραση ενός πίνακα, όπως στο παράδειγμα:

```
const fruits = ['Μπανάνα', 'Αχλάδι', 'Μήλο', 'Μάνγκο'];
for (let i = 0; i < fruits.length; i++) {
  console.log(`${i + 1}. ${fruits[i]} `);
}
```

Ο κώδικας αυτός θα διαπεράσει τον πίνακα των φρούτων και θα τα τυπώσει:

```
1. Μπανάνα
2. Αχλάδι
3. Μήλο
4. Μάνγκο
```

Με την ευκαιρία να υπενθυμίσουμε ότι αν η μεταβλητή μετρητής οριστεί με τη λέξη `let` έχει εμβέλεια μόνο μέσα στο μπλοκ for.

Τέλος να αναφέρουμε ότι μπορούμε να παραλείψουμε κάποια από τις εκφράσεις της for ή και ολες τις εκφράσεις, όμως θα πρέπει να διατηρήσουμε τα σύμβολα ``;''. Για παράδειγμα μια δομή που είναι ισοδύναμη με while(true) θα μπορούσε να γραφτεί ως δομή for ως εξής:

```
for (;;){
    μπλοκ-εντολών
}
```

2.7.4 Εντολή `for/of`

Η δομή `for/of` εισήχθη στην JavaScript με την έκδοση ES6. Είναι δομή που θυμίζει την αντίστοιχη δομή for της Python.

Η δομή αυτή επιτρέπει να διαπεράσουμε μια ακολουθία στοιχείων. Μια ακολουθία είναι για παράδειγμα ένας πίνακας, ή μια συμβολοσειρά.

Η σύνταξη της εντολής αυτής είναι:

```
for (let στοιχείο of ακολουθία) {
    μπλοκ εντολών
}
```

Ο βρόχος επαναλαμβάνεται για κάθε στοιχείο της ακολουθίας.

Ένα παράδειγμα:

```
const fruits = ['Μπανάνα', 'Αχλάδι', 'Μήλο', 'Μάνγκο'];
for (let fruit of fruits) {
  console.log(fruit);
}
```

Εδώ θα πάρουμε ένα προς ένα τα ονόματα των φρούτων.

Ας δούμε ένα ακόμη παράδειγμα διαπέρασης των χαρακτήρων μιας συμβολοσειράς στο πρόβλημα καταγραφής συχνότητας εμφάνισης χαρακτήρων.

```
let freq = {}; // αντικείμενο JavaScript
for (let letter of 'Ελλάδα') {
  freq[letter] = (freq[letter] || 0) + 1;
}
freq;
> { 'Ε': 1, 'λ': 2, 'ά': 1, 'δ': 1, 'α': 1 }
```

Μερικές παρατηρήσεις για τη λύση αυτή, χρησιμοποιούμε ένα αντικείμενο, το freq (θα γίνουν οι επίσημες συστάσεις αντικειμένων σε επόμενο κεφάλαιο, για την ώρα θεωρήστε ότι είναι παρόμοιο με ένα λεξικό της Python). Για κάθε γράμμα, αν το γράμμα υπάρχει ήδη στο αντικείμενο freq, προστίθεται στην τιμή της ιδιότητας με τιμή τον χαρακτήρα +1 αν δεν υπάρχει, δημιουργείται η ιδιότητα και αρχικοποιείται σε 1. Αυτό γιατί η έκφραση: `a || b` παίρνει την τιμή του πρώτου στοιχείου της έκφρασης το οποίο είναι αληθές, αλλοιώς του τελευταίου στοιχείου.

2.7.5 Εντολή `for/in`

Η δομή `for/in` μοιάζει με την for/of που είδαμε στην προηγούμενη ενότητα, υπάρχει στην JavaScript από παλαιότερες εκδόσεις και έχει ευρύτερη χρήση από την προηγούμενη, αφού στην περίπτωση αυτή το `στοιχείο` μπορεί να είναι οι ιδιότητες ενός αντικειμένου, όχι απαραίτητα μιας ακολουθίας.

Η σύνταξη είναι:

```
for (let στοιχείο in αντικείμενο) {
    μπλοκ εντολών
}
```

Για τους πίνακες και τις συμβολοσειρές το *στοιχείο* παίρνει τις τιμές του δείκτη που ως γνωστόν παίρνει τιμές από 0 μέχρι length-1.

Συνεπώς το παράδειγμα της προηγούμενης ενότητας παρουσίασης των αγαπημένων μας φρούτων τροποποιείται ως ακολούθως:

```
const fruits = ['Μπανάνα', 'Αχλάδι', 'Μήλο', 'Μάνγκο'];
for (let indx in fruits) {
  console.log(fruits[indx]);
}
```

Είναι αντιληπτό από το παράδειγμα ότι η δομή `for/in` δεν εκφράζει με την ίδια απλότητα την διαπέραση των στοιχείων ενός πίνακα. Για το λόγο αυτό για την περίπτωση πινάκων ή συμβολοσειρών που είναι και η πιο συνήθης περίπτωση χρήσης του βρόχου `for` η νεότερη δομή `for/of` είναι προτιμότερη.

2.8 Σύνοψη

Σε αυτό το κεφάλαιο είδαμε τους πρωτογενείς τύπους δεδομένων της JavaScript, Number, String, Boolean, null, undefined. Στη συνέχεια έγινε ανασκόπηση των βασικών προγραμματιστικών δομών της JavaScript: των εντολών ελέγχου της ροής του προγράμματος όπως η `if` και `switch`, των εντολών που επιτρέπουν την επαναληπτική εκτέλεση ενός κώδικα, όπως οι `for`, `while`, `do/while` και τέλος εντολών που επιτρέπουν τη μετάβαση σε άλλο τμήμα του κώδικα, όπως η `break`, `continue` και η `throw`.

3 Τύποι δεδομένων αναφοράς: Πίνακες, Συναρτήσεις και Αντικείμενα στη JavaScript

3.1 Τύποι δεδομένων αναφοράς

Ως τώρα έχουμε δει τους βασικούς πρωτογενείς τύπους δεδομένων (primitive data types). Στο κεφάλαιο αυτό θα δούμε κάποιους άλλους τύπους δεδομένων που ονομάζονται **τύποι δεδομένων αναφοράς (reference data types)**. Οι τύποι δεδομένων αναφοράς είναι τα αντικείμενα (objects), οι πίνακες (arrays), αλλά και οι συναρτήσεις (functions) που και αυτές όπως θα δούμε είναι αντικείμενα. Υπάρχουν κάποιες σημαντικές διαφορές μεταξύ των δεδομένων αναφοράς και πρωτογενών δεδομένων: Τα πρωτογενή δεδομένα είναι μη τροποποιήσημα (immutable), ενώ τα δεδομένα αναφοράς μπορούν να τροποποιηθούν. Τα δεδομένα αναφοράς έχουν ιδιότητες κάτι που δεν έχουν τα πρωτογενή δεδομένα, επίσης η αντιγραφή ενός πρωτογενούς δεδομένου `a` σε μια νέα μεταβλητή (πχ. `let b = a;`) δημιουργεί νέο αντίγραφο της τιμής του πρωτοτεγενούς τύπου, ενώ στα δεδομένα αναφοράς δημιουργεί μια νέα αναφορά στο ήδη υπάρχον αντικείμενο. Αυτό όπως θα δούμε έχει ιδιαίτερη σημασία στις μεταβλητές που περνάμε στα ορίσματα συναρτήσεων.

Στο κεφάλαιο αυτό θα αρχίσουμε με τους **πίνακες (arrays)**. Στη συνέχεια θα δούμε τον τρόπο που η JavaScript χειρίζεται τις **συναρτήσεις (functions)** και τα **αντικείμενα (objects)**.

Οι συναρτήσεις, οι οποίες και αυτές είναι αντικείμενα πρώτης τάξης εκτός από το ρόλο που παίζουν στη δόμηση ενός προγράμματος, χρησιμοποιούνται ως βασικό συστατικό του *συναρτησιακού μοντέλου προγραμματισμού (functional programming)* το οποίο είναι ιδιαίτερα διαδεδομένο στην JavaScript.

Τέλος τα αντικείμενα αποτελούν θεμελιώδες χαρακτηριστικό της αρχιτεκτονικής της JavaScript και χρησιμοποιούνται εκτενώς σε εφαρμογές που ακολουθούν το *αντικειμενοστρεφές μοντέλο προγραμματισμού (object oriented programming)*.

3.2 Πίνακες

Ο πρώτος τύπος δεδομένων αυτής της κατηγορίας που θα δούμε είναι οι **Πίνακες (Arrays)**. Ένας πίνακας είναι μια διαταγμένη ακολουθία στοιχείων. Κάθε στοιχείο βρίσκεται σε συγκεκριμένη θέση στην ακολουθία. Η θέση του στοιχείου ορίζεται από ένα *δείκτη (index)*, ένα ακέραιο αριθμό που παίρνει τιμές από 0 (το πρώτο στοιχείο) μέχρι length-1, όπου length το πλήθος στοιχείων του πίνακα. Οι πίνακες συναντώνται σε όλες σχεδόν τις γλώσσες

προγραμματισμού, πχ στην Python ο αντίστοιχος τύπος δεδομένων είναι οι λίστες. Οι πίνακες της JavaScript είναι δυναμικοί τύποι δεδομένων, το μέγεθος τους δεν χρειάζεται να οριστεί κατά τη δημιουργία τους, καθώς μπορούν να προστεθούν ή να διαγραφούν στοιχεία (είναι τροποποιήσιμος τύπος δεδομένων).

Τα στοιχεία του πίνακα μπορεί να είναι διαφορετικών τύπων μεταξύ τους, μπορεί να είναι πρωτογενή δεδομένα, ή αντικείμενα, ή και άλλοι πίνακες.

Όπως θα δούμε στη συνέχεια, τα αντικείμενα τύπου Array κληρονομούν μεθόδους (ορίζονται ως ιδιότητες στο Array.prototype) με τις οποίες μπορούμε να διαχειριστούμε τα στοιχεία του πίνακα.

Οι πίνακες μπορεί να δημιουργηθούν είτε με ορισμό τιμής τους (Array literal) ή με κλήση της δημιουργού συνάρτησης νέων αντικειμένων `new Array()`.

3.2.1 Δημιουργία πίνακα με ορισμό της τιμής του

Ας δούμε μερικά παραδείγματα δημιουργίας πινάκων με ορισμό των τιμών τους.

```
const shopping = ['ψωμί', 'γάλα', 'τυρί', 'μήλα'];
const sequence = [1, 1, 2, 3, 5, 8, 13];
const random = ['tree', 795, [0, 1, 2]];
const empty = [];
const sparse = [1, , ,];
```

Όπως βλέπουμε στο παράδειγμα ο πίνακας shopping είναι πίνακας που περιέχει 4 συμβολοσειρές. Το μήκος του πίνακα shopping.length είναι 4. Ο πίνακας sequence είναι πίνακας ακεραίων με 7 στοιχεία, (sequence.length = 7). Ο πίνακας random περιέχει τρία στοιχεία διαφορετικών τύπων μεταξύ τους, μια συμβολοσειρά, ένα ακέραιο και ένα πίνακα, (random.length = 3). Ο πίνακας empty είναι πίνακας χωρίς στοιχεία, συνεπώς empty.length = 0. Τέλος ο πίνακας sparse έχει τρία στοιχεία, εκ των οποίων το πρώτο έχει οριστεί, ενώ τα άλλα δύο όχι, παρόλα αυτά το μήκος του είναι sparse.length = 3.

3.2.2 Δημιουργία πίνακα με new Array()

Ένας εναλλακτικός τρόπος δημιουργίας πίνακα είναι με κλήση της δημιουργού συνάρτησης του αντικειμένου Array:

```
const ar = new Array();
```

Όπως θα δούμε σε επόμενο κεφάλαιο, αυτός είναι ο συνήθης τρόπος δημιουργίας αντικειμένων με κλήση της συνάρτησης-δημιουργού με τη λέξη **new**.

Ο πίνακας ar που δημιουργείται από την παραπάνω εντολή είναι απολύτως ισοδύναμος με την εντολή δημιουργίας πίνακα με ορισμό τιμής:

```
const ar = [];
```

Στη δημιουργό συνάρτηση Array() μπορούμε να περάσουμε ως όρισμα το μήκος του πίνακα που δημιουργούμε.

```
const ar = new Array(50);
ar.length;
> 50
```

Σε αυτή την περίπτωση δημιουργείται ένας κενός πίνακας, με μήκος 50.

Εναλλακτικά στα ορίσματα της συνάρτησης δημιουργού μπορούμε να περάσουμε τα στοιχεία του πίνακα:

```
const ar = new Array(10,20);
ar.length;
> 2
```

Στην περίπτωση αυτή δημιουργείται ένας πίνακας με δύο στοιχεία, άρα μήκους 2.

3.2.3 Αραιοί πίνακες

Μια ιδιαιτερότητα της JavaScript είναι ότι μπορούμε σε ένα πίνακα να εισάγουμε στοιχεία πέραν της τρέχουσας τιμής του δείκτη, και άρα του μήκους του πίνακα.

Ας δούμε ένα παράδειγμα:

```
const ar = [1,2,3];
undefined
ar[10] = 20;
20
console.log(ar)
(11) [1, 2, 3, empty × 7, 20]
ar.length
11
```

Στο παράδειγμα ορίζουμε ένα πίνακα τριών στοιχείων. Στη συνέχεια στην 11η θέση του (δείκτης 10) βάζουμε ένα νέο στοιχείο. Τότε το μήκος του πίνακα γίνεται 11, δηλαδή λαμβάνει υπόψη τη θέση του τελευταίου στοιχείου, αν και ενδιάμεσα υπάρχουν 7 άδειες θέσεις. Ο πίνακας αυτός λέγεται αραιός (sparse).

Να σημειωθεί ότι σε άλλες γλώσσες προγραμματισμού αυτή η συμπεριφορά δεν θα ήταν επιτρεπτή, για παράδειγμα η εισαγωγή στοιχείου στη θέση 10 ενός πίνακα 3 στοιχείων στην Python θα έδινε IndexError.

3.2.4 Αρχικοποίηση πίνακα

Αν επιθυμούμε να αρχικοποιήσουμε ένα πίνακα μήκους length, μπορούμε, εκτός από τον κλασσικό τρόπο με χρήση μιας δομής επανάληψης, να χρησιμοποιήσουμε τη μέθοδο fill(). Για παράδειγμα αν θέλουμε να δώσουμε ως αρχική τιμή σε όλα τα στοιχεία ενός πίνακα την τιμή 5:

```
const ar = new Array(10);
ar.fill(5);
console.log(ar)
> [
  5, 5, 5, 5, 5,
  5, 5, 5, 5, 5
]
```

3.2.5 Τροποποίηση στοιχείων πίνακα

Οι πίνακες είναι τροποποιήσημες δομές δεδομένων.

Με χρήση του συμβολισμού δείκτη `πίνακας[δείκτης]` μπορούμε να ορίσουμε ένα νέο στοιχείο ή να τροποποιήσουμε ένα στοιχείο του πίνακα που ήδη υπάρχει.

Επίσης μπορούμε να διαγράψουμε ένα στοιχείο με τον τελεστή `delete` όπως μπορούμε να διαγράψουμε τις ιδιότητες κάθε αντικειμένου της JavaScript.

Θα πρέπει να σημειωθεί ότι οι πράξεις αυτές δεν επηρεάζουν το μήκος του πίνακα που παραμένει ίσο με την τιμή του μέγιστου δείκτη + 1.

Επίσης το περιεχόμενο ενός πίνακα μπορεί να τροποποιηθεί με χρήση μεθόδων της κλάσης Array, όπως οι `push()`, `pop()`, `shift()`, `unshift()`, όπως θα περιγραφεί στην επόμενη ενότητα.

3.2.6 Μέθοδοι πινάκων

Ο τύπος δεδομένων Array διαθέτει πολύ ισχυρές μεθόδους για διαχείριση και τροποποίηση των στοιχείων ενός πίνακα. Στην ενότητα αυτή θα εστιάσουμε στις μεθόδους τροποποίησης των στοιχείων, και ταξινόμησης τους, ενώ σε επόμενο κεφάλαιο, όταν εισάγουμε τις έννοιες του συναρτησιακού προγραμματισμού θα δούμε μεθόδους για επαναληπτική εφαρμογή συναρτήσεων σε πίνακες.

3.2.6.1 Μέθοδοι στοίβας (stuck)

Οι μέθοδοι `push(στοιχείο)` και `pop()` επιτρέπουν την εισαγωγή και διαγραφή στοιχείων στο τέλος ενός πίνακα, ώστε να δώσουν στον πίνακα συμπεριφορά στοίβας LIFO (last in first out).

Παρόμοια λειτουργία έχουν οι μέθοδοι `shift()` και `unshift(στοιχείο)` που αντίστοιχα διαγράφουν ή εισάγουν στοιχείο στον πίνακα, μόνο που αυτές το κάνουν στην αρχή του πίνακα και όχι στο τέλος, όπως οι push,pop.

Θα πρέπει να σημειωθεί ότι η push() επιστρέφει το νέο μήκος του πίνακα, ενώ η pop() επιστρέφει το στοιχείο που διαγράφτηκε. Ας δούμε μερικά παραδείγματα.

```
const myArray = ["Athens", "Patras", "Thessaloniki", "Volos"]
myArray.push('Larissa'); // επιστρέφει 5
myArray.push('Katerini', 'Kavala'); // επιστρέφει 7
myArray.pop(); // επιστρέφει "Kavala"
```

Αντίστοιχη λειτουργία έχουμε με τις μεθόδους shift(), unshift():

```
const myArray = ["Athens", "Patras", "Thessaloniki", "Volos"]
myArray.unshift('Larissa'); // επιστρέφει 5
myArray.unshift('Katerini', 'Kavala'); // επιστρέφει 7
myArray.shift(); // επιστρέφει "Katerini"
```

3.2.7 Μέθοδοι υπό-πινάκων

Η μέθοδος `slice()`, που την έχουμε δει και στις συμβολοσειρές, μάς επιτρέπει να πάρουμε μια ``φέτα'' του πίνακα, ένα υποσύνολο των στοιχείων, και να δημιουργήσουμε ένα νέο πίνακα.

Ένα παράδειγμα:

```
const year = ["Ιανουάριος", "Φεβρουάριος", "Μάρτιος",
    "Απρίλιος", "Μάιος", "Ιούνιος", "Ιούλιος", "Αύγουστος",
    "Σεπτέμβριος", "Οκτώβριος", "Νοέμβριος", "Δεκέμβριος"];
const spring = year.slice(2,5);
console.log(spring);
> ["Μάρτιος", "Απρίλιος", "Μάιος"]
```

Η μέθοδος `slice()`, επιστρέφει ένα νέο πίνακα. Παίρνει δύο ορίσματα, τον δείκτη του πρώτου στοιχείου του υποπίνακα και τον δείκτη του τελευταίου στοιχείου, το οποίο όμως δεν περιλαμβάνεται. Αν παραληφθεί το δεύτερο όρισμα, τότε επιστρέφει τα στοιχεία του πίνακα μέχρι το τέλος.

Μια μέθοδος που τροποποιεί τα στοιχεία ενός πίνακα, εισάγοντας νέα στοιχεία και διαγράφοντας στοιχεία που υπάρχουν, είναι η `splice()`. Θα πρέπει να σημειωθεί ότι η μέθοδος αυτή επιστρέφει τα στοιχεία που διαγράφτηκαν, ωστόσο τροποποιεί τον αρχικό πίνακα.

Η μέθοδος αυτή συντάσσεται ως εξής:

```
myArray.splice(θέση, πλήθος-διαγραφή, στοιχεία-για-εισαγωγή);
```

όπου η `θέση` ορίζει το δείκτη του στοιχείου στο οποίο θα γίνει η τροποποίηση, η `πλήθος-διαγραφή` ορίζει το πλήθος στοιχείων που θα διαγραφούν από τη θέση αυτή και μετά, και τα `στοιχεία-για-εισαγωγή` είναι τα στοιχεία που θα εισαχθούν στο σημείο αυτό.

Να σημειωθεί ότι αν παραληφθούν το δεύτερο και τρίτο όρισμα, τότε θεωρούμε ότι τα στοιχεία για διαγραφή εκτείνονται μέχρι το τέλος του πίνακα, χωρίς στοιχεία για εισαγωγή.

Ένα πρώτο παράδειγμα είναι:

```
const ar = [1,2,3,4,5,6]
let x = ar.splice(3);
console.log(ar);
[1,2,3]
console.log(x);
[4,5,6]
```

Ένα ακόμη παράδειγμα με διαγραφή ορισμένου πλήθους στοιχείων:

```
const ar = [1,2,3,4,5,6]
let x = ar.splice(3,2); //διαγράφονται 2 στοιχεία
console.log(ar);
[1,2,3,6]
console.log(x);
[4,5]
```

Ακολουθεί ένα παράδειγμα με διαγραφή και ταυτόχρονη εισαγωγή στοιχείων:

```
const ar = [1,2,3,4,5,6]
let x = ar.splice(3,2,30,40); //διαγράφονται 2 στοιχεία και εισάγονται 2.
console.log(ar);
[1,2,3,30,40,6]
console.log(x);
[4,5]
```

3.2.7.1 Άσκηση

Έστω ο παρακάτω κώδικας, ποιο το αποτέλεσμα;

```
constidfruits = ["Μπανάνα", "Αχλάδι", "Μήλο", "Μάνγκο"];
fruits.splice(2, 1, "Λεμόνι", "Πεπόνι");
```

Απάντηση: θα διαγραφεί ένα φρούτο, στη θέση 2 (το `"Μήλο"`) και θα εισαχθούν δύο στην ίδια θέση, άρα τελικά θα έχουμε:

```
["Μπανάνα", "Αχλάδι", "Λεμόνι", "Πεπόνι", "Μάνγκο"]
```

3.2.8 Μέθοδοι αναζήτησης

Οι μέθοδοι `indexOf()` και `lastIndexOf()` μάς επιτρέπουν να αναζητήσουμε ένα στοιχείο σε πίνακα. Οι μέθοδοι αυτές εφαρμόζονται και στις συμβολοσειρές.

Μας επιστρέφουν το δείκτη του στοιχείου που έχει τιμή ίση με το όρισμα, η μεν indexOf() αρχίζοντας την αναζήτηση από την αρχή του πίνακα, και η lastIndexOf() από το τέλος του πίνακα. Αν το στοιχείο δεν βρεθεί, επιστρέφουν την τιμή -1.

Ας δούμε ένα παράδειγμα:

```
const ar = [5,10,15,20,10,5];
ar.indexOf(15);
> 2
ar.lastIndexOf(10);
> 4
```

Θα πρέπει να προσέξουμε ότι η αναζήτηση εφαρμόζει τον τελεστή σύγκρισης ``===" δηλαδή αναζητάει στοιχεία που είναι ίσα ως προς τον τύπο και την τιμή με το όρισμα της συνάρτησης.

Να σημειωθεί ότι οι μέθοδοι αυτές παίρνουν ως δεύτερο προαιρετικό όρισμα ένα δείκτη από πού να αρχίσει η αναζήτηση.

3.2.9 Μέθοδοι ταξινόμησης

Η μέθοδος `sort()` ταξινομεί αλφαβητικά τα στοιχεία του πίνακα τροποποιώντας τον ίδιο τον πίνακα στη θέση του.

Ένα πρώτο παράδειγμα:

```
const students = ['Μαρία', 'Κώστας', 'Ανδρέας'];
students.sort();
> [ 'Ανδρέας', 'Κώστας', 'Μαρία' ]
```

Θα πρέπει να προσέξουμε ότι αν δεν δώσουμε ως όρισμα μια συνάρτηση ταξινόμησης, η μέθοδος αυτή ταξινομεί αλφαβητικά τα στοιχεία, και αν αυτά δεν είναι συμβολοσειρές τα μετατρέπει σε συμβολοσειρές πριν την ταξινόμηση, για παράδειγμα:

```
const ar = [5, 10, 15];
ar.sort();
ar;
> [ 10, 15, 5 ]
```

Ο λόγος για τον οποίο αυτός ο πίνακας ακεραίων ταξινομείται με αυτόν τον τρόπο είναι γιατί τα στοιχεία του μετατρέπονται σε `[``5", ``10", ``15"] τα οποία ταξινομούνται ως συμβολοσειρές με τον τρόπο που φαίνεται στο παράδειγμα.

Για να αντιμετωπίσουμε το πρόβλημα αυτό θα πρέπει να περάσουμε ως όρισμα μια συνάρτηση ταξινόμησης η οποία να δέχεται ως όρισμα δύο διαδοχικά στοιχεία και να μας επιστρέφει ένα θετικό αριθμό ή αρνητικό αντίστοιχα για να οριστεί η διάταξή τους. Το θέμα αυτό θα το δούμε πιο αναλυτικά όταν συζητήσουμε τις συναρτήσεις σε επόμενο κεφάλαιο, όμως για πληρότητα, παρατίθεται εδώ η λύση του προβλήματος ταξινόμησης ακεραίων.

```
const ar = [5, 10, 15];
ar.sort((a, b) => a - b); //αύξουσα σειρά
ar;
> [ 5, 10, 15 ]
```

ενώ αντίστοιχα για φθίνουσα σειρά ταξινόμησης:

```
const ar = [5, 10, 15];
ar.sort((a, b) => b - a); // φθίνουσα σειρά
ar;
> [ 15, 10, 5 ]
```

Τέλος θα πρέπει να αναφερθεί ότι η μέθοδος `reverse()` αντιστρέφει τη σειρά των στοιχείων τροποποιώντας το ίδιο το αρχείο.

3.2.10 Μέθοδοι μετατροπής πίνακα σε συμβολοσειρά

Όταν συζητήσαμε τις συμβολοσειρές είδαμε την μέθοδο `split(διαχωριστικό)` η οποία μετατρέπει μια συμβολοσειρά σε πίνακα στοιχείων με βάση ένα διαχωριστικό. Αν μάλιστα το διαχωριστικό είναι η κενή συμβολοσειρά, `""` τότε μετατρέπει τη συμβολοσειρά σε ένα πίνακα που περιέχει ως στοιχεία τους χαρακτήρες της συμβολοσειράς.

Την αντίστροφη δουλειά κάνει η μέθοδος `join()` του τύπου Array. Η μέθοδος αυτή μετατρέπει τα στοιχεία σε συμβολοσειρές και στη συνέχεια τις συνενώνει σε μια συμβολοσειρά, θέτωντας ως διαχωριστικό τον χαρακτήρα `","`. Όπως και στην split() μπορούμε και στην join() να ορίσουμε ένα διαφορετικό διαχωριστικό μεταξύ των στοιχείων.

Παράδειγμα:

```
const ar = [5,10,15];
ar.join()
> "5,10,15"
```

Αν επιθυμούμε να συνενώσουμε τα στοιχεία χωρίς διαχωριστικό, τότε θα πρέπει να περάσουμε ως όρισμα στη μέθοδο `join()` την κενή συμβολοσειρά `""`.

Συνοψίζοντας, στην ενότητα αυτή είδαμε τον τύπο δεδομένων Array, που είναι ο πρώτος τύπος δεδομένων αναφοράς. Θα ακολουθήσει ο τύπος δεδομένων Object που είναι ο πιο βασικός τύπος δεδομένων αναφοράς. Υπάρχουν επίσης άλλες δομές που μοιάζουν με Array, όπως η λίστα κόμβων NodeList του DOM που επιστρέφει η συνάρτηση document.querySelectorAll(). Πολλές από τις μεθόδους που είδαμε στην ενότητα αυτή έχουν εφαρμογή και σε αυτή την περίπτωση.

Θα πρέπει να σημειωθεί οτι ο τύπος Array συνδέεται άμεσα με τις επαναληπτικές δομές `for/in` `for/of`, αλλά και με τις μεθόδους του συναρτησιακού προγραμματισμού που θα δούμε στην επόμενη ενότητα.

3.3 Συναρτήσεις

3.3.1 Εισαγωγή

Συνάρτηση είναι ένα επαναχρησιμοποιούμενο μπλοκ κώδικα, το οποίο ορίζεται μια φορά και μπορεί να κληθεί και να εκτελεστεί πολλές φορές. Η έννοια της συνάρτησης υπάρχει στις περισσότερες γλώσσες προγραμματισμού. Η JavaScript είναι ισχυρά συναρτησιακή γλώσσα, ορίζει τις συναρτήσεις ως αντικείμενα (αναφέρονται ως *callable objects*). Οι συναρτήσεις που ορίζονται ως ιδιότητες αντικειμένων (objects) ονομάζονται *μέθοδοι* (methods).

Το όνομα μιας συνάρτησης θα πρέπει απαραίτητα να ξεκινάει από ένα αλφαβητικό χαρακτήρα (κεφαλαία ή πεζά) ή το χαρακτήρα της κάτω παύλας (underscore) ή $. Μετά τη δήλωση συνάρτησης, αυτή μπορεί να κληθεί με χρήση του ονόματός της. Εξ ορισμού, οι τιμές των ορισμάτων εισόδου αποδίδονται στον κώδικα της συνάρτησης με αντιγραφή τους σε τοπικές μεταβλητές (κλήση μέσω τιμής -- by value). Αν όμως το όρισμα εισόδου είναι κάποιος πίνακας ή άλλο αντικείμενο, αυτό αποδίδεται στην εσωτερική τοπική μεταβλητή μέσω αναφοράς (by reference). Τέλος, υπάρχει η δυνατότητα ορισμού ανώνυμων συναρτήσεων (αντίστοιχες των συναρτήσεων lambda της Python), ως ορίσματα άλλων συναρτήσεων.

Η σύνταξη του ορισμού της συνάρτησης έχει επεκταθεί στην ES06, με την εισαγωγή της σημειολογίας της συνάρτησης βέλους ``=>" όπως θα δούμε στη συνέχεια.

3.3.2 Ορισμός συνάρτησης με χρήση της λέξης `function`

Υπάρχουν διάφοροι τρόποι ορισμού μιας συνάρτησης

Ο πρώτος τρόπος που θα δούμε είναι με χρήση της λέξης-κλειδί `function`

```
function onomaSynarthshs(orismata) {
  // σώμα συνάρτησης
}
```

Η εντολή `return` ορίζει τι επιστρέφει μια συνάρτηση. Αν μια συνάρτηση δεν περιέχει εντολή `return` στο τέλος εκτέλεσής της, τότε επιστρέφει την τιμή `undefined`.

Ένα παράδειγμα είναι η παρακάτω συνάρτηση που επιστρέφει ένα τυχαίο ακέραιο μεταξύ 1 και number.

```
function random(number) {
   // τυχαίος αριθμός μεταξύ 1 και number
   return Math.floor(Math.random()*number+1);
}

let x = random(5);  //κλήση συνάρτησης
```

Θα πρέπει να σημειωθεί ότι οι συναρτήσεις που δηλώνονται με αυτόν τον τρόπο μπορούν να χρησιμοποιηθούν σε οποιοδήποτε τμήμα του κώδικα, ακόμη και πριν τον ορισμό τους.

Παραλλαγή αυτής της δήλωσης είναι η εκχώρηση της συνάρτησης σε μεταβλητή (literal notation).

```
const onomaSynarthshs  = function(orismata) {
    //κώδικας
    return δεδομένα;
}
```

Αυτός ο τρόπος ορισμού συνάρτησης οφείλει την ύπαρξη του στο γεγονός ότι οι συναρτήσεις είναι αντικείμενα.

Θα πρέπει να προσέξουμε ότι σε αυτή την περίπτωση ισχύουν όλα όσα ισχύουν για ορισμό μεταβλητών, για παράδειγμα δεν μπορούν να χρησιμοποιηθούν πριν τον ορισμό τους.

Ξαναγράφουμε τη δήλωση της `random(number)`:

```
const random = function(number) {
   // τυχαίος αριθμός μεταξύ 1 και number
   return Math.floor(Math.random()*number+1);
}
```

Μια πιο σπάνια περίπτωση είναι η συνάρτηση που ορίζεται μέσω εκχώρησης της σε μεταβλητή να έχει και αυτή όνομα, ώστε για παράδειγμα να χρησιμοποιηθεί σε περίπτωση αναδρομής.

Ένα παράδειγμα αναδρομικού υπολογισμού παραγοντικού είναι:

```
const f = function fact(x) {
  if (x <= 1) return 1;
  else return x * fact(x - 1);
};
f(5);
>120
```

3.3.3 Ορισμός συνάρτησης με χρήση βέλους =>

Από την έκδοση ES6 της JavaScript έχει εισαχθεί ένας ακόμη τρόπος ορισμού συναρτήσεων, με χρήση του συμβόλου =>, οι συναρτήσεις που ορίζονται με αυτόν τον τρόπο ονομάζονται **συναρτήσεις βέλους** (arrow functions).

```
const f = (parametroi) => {
  // σώμα συνάρτησης
}
```

Ο τρόπος αυτός ορισμού είναι πιο σύντομος, παραλείπει τη χρήση της λέξης function και όπως θα δούμε στη συνέχεια μπορεί σε όρισμένες περιπτώσεις να απλοποιηθεί περαιτέρω.

Όταν η συνάρτηση περιέχει στο σώμα της μόνο μια εντολή και αυτή είναι η εντολή `return`, τότε μπορεί να παραληφθεί η λέξη return και τα άγκιστρα ως εξής:

```
// πρώτος τρόπος
const mult =  (x,y) => {
  return x * y;
};

// απλοποιημένη μορφή
const mult =  (x,y) => x * y;
```

Θα πρέπει να δοθεί ιδιαίτερη προσοχή στην περίπτωση εκείνη που η συνάρτηση επιστρέφει την τιμή ενός αντικειμένου (οι τιμές των αντικειμένων, όπως θα δούμε στη συνέχεια ορίζονται ως `{key: value}`). Στην περίπτωση αυτή είμαστε υποχρεωμένοι να βάλουμε το αντικείμενο σε παρενθέσεις, γιατί αλλοιώς υπάρχει σύγχυση στη χρήση των άγκιστρων ως τερματικών χαρακτήρων της συνάρτησης αλλά και των αντικειμένων.

Ένα παράδειγμα, αν μια συνάρτηση λαμβάνει ως ορίσματα το όνομα και ηλικία και επιστρέφει το αντικείμενο όπως `{name:"Κώστας", age:20}`, η συνάρτηση θα πρέπει να γραφεί ως εξής:

```
const f = (n, a) => ({name:n, age:a})
```

Αντίθετα η έκφραση :

```
const f = (n, a) => {name:n, age:a}
```

θα δώσει συντακτικό λάθος.

Οι συναρτήσεις βέλη χρησιμοποιούνται κύρια ως ανώνυμες συναρτήσεις, για παράδειγμα στις περιπτώσεις που περνάμε μια συνάρτηση ως όρισμα μιας άλλης συνάρτησης (callback functions), ή για τις περιπτώσεις που ορίζουμε τη συνάρτηση ως χειριστή ενός συμβάντος.

3.3.4 Εμφώλευση συναρτήσεων

Επιτρέπεται να οριστεί μια συνάρτηση μέσα σε μια άλλη συνάρτηση. Η εμφωλευμένη συνάρτηση μπορεί να κληθεί μόνο μέσα στην συνάρτηση που έχει οριστεί, και έχει πρόσβαση στις μεταβλητές της συνάρτησης αυτής.

Για παράδειγμα για τον υπολογισμό της υποτείνουσας μπορούμε να χρησιμοποιήσουμε την εξής συνάρτηση, μέσα στην οποία ορίζουμε μια συνάρτηση υπολογισμού του τετραγώνου:

```
function ypotinousa(a, b) {
  const sq = (x) => x*x;
  return Math.sqrt(sq(a) + sq(b));
}

ypotinousa(3,4)
> 5
```

Οι κανόνες εμβέλειας μεταβλητών συναρτήσεων είναι αντικείμενο της επόμενης ενότητας.

3.3.5 Εμβέλεια μεταβλητών

- Μεταβλητές που ορίζονται έξω από όλες τις συναρτήσεις, ανήκουν στην περιοχή καθολικής εμβέλειας global scope.
- Μεταβλητές που ορίζονται στο επίπεδο αυτό είναι προσβάσιμες από όλον τον κώδικα.
- Κάθε συνάρτηση ορίζει τοπική εμβέλεια (local scope) και οι μεταβλητές που ορίζονται στο επίπεδο αυτό είναι προσβάσιμες μόνο μέσα στη συνάρτηση

Έστω το παράδειγμα:

```
let x = 1;

function a() {
  let y = 2;
  b(x);
  b(y);
}

function b(value) {
  console.log( `Τιμή: ${value}`);
}
a();
> 'Τιμή: 1'
> 'Τιμή: 2'
```

Η συνάρτηση `b()` όταν καλείται μέσα από τη συνάρτηση `a()` έχει πρόσβαση και στις δύο μεταβλητές x,y, η πρώτη από τις οποίες έχει οριστεί στην περιοχή καθολικής εμβέλειας, ενώ η δεύτερη είναι τοπική μεταβλητή της `a()`.

Ας δούμε ένα παράδειγμα κώδικα που παράγει ReferenceError λόγω εμβέλειας μεταβλητών.

```
function myFunction() {
  let userName = 'Nikos';
  console.log(userName); // ok
}
console.log(userName); // ReferenceError
```

Αν στον κώδικα αυτόν αντικαθιστούσαμε τη δήλωση της τοπικής μεταβλητής με τη λέξη κλειδί `var` και πάλι δεν θα είχαμε κάποια αλλαγή, αφού η μεταβλητή userName παραμένει τοπική μεταβλητή στη συνάρτηση `myFunction()`.

Ένα ακόμη παράδειγμα:

Η μεταβλητή `color` στο παρακάτω παράδειγμα είναι καθολική μεταβλητή, αφού ορίζεται εκτός των συναρτήσεων. Αντίθετα, η `anotherColor` είναι τοπική στη συνάρτηση `changeColor` και είναι προσβάσιμη και στη συνάρτηση `swapColors` που ορίζεται εντός της `changeColor`.

```
let color = 'μπλε';
function changeColor() {
    let anotherColor = 'κόκκινο';
    function swapColors() {
        let tempColor = anotherColor;
        anotherColor = color;
        color = tempColor;
        // color, anotherColor, tempColor είναι προσβάσιμες εδώ
    }
    // μόνο color,anotherColor είναι προσβάσιμες
    swapColors();
}
// μόνο η color είναι προσβάσιμη εδώ
changeColor();
console.log(color); // 'κόκκινο'
```

3.3.5.1 Άσκηση

Έστω ο παρακάτω κώδικας:

```
function myBigFunction() {
  let myValue = 1;
  subFunction1();
  subFunction2();
}
function subFunction1() {
  console.log(myValue);
}
function subFunction2() {
  console.log(myValue);
}
myBigFunction();
```

Ποιο το αποτέλεσμα;

Απάντηση:

Ο κώδικας αυτός θα προκαλέσει σφάλμα ReferenceError: myValue is not defined Αυτό γιατί η μεταβλητή myValue ορίζεται ως τοπική μεταβλητή στο πλαίσιο της συνάρτησης `myBigFunction()`, άρα δεν είναι προσβάσιμη από τις άλλες δύο συναρτήσεις `subFunction1()` και `subFunction2()` οι οποίες επίσης είναι ορισμένες στην καθολική περιοχή. Αν επιθυμούσαμε να έχουν πρόσβαση στη μεταβλητή αυτή θα έπρεπε να ορίσουμε τις συναρτήσεις μέσα στην `myBigFunction()` ως εξής:

```
function myBigFunction() {
  let myValue = 1;
  function subFunction1() {
    console.log(myValue);
  }
  function subFunction2() {
    console.log(myValue);
  }
  subFunction1();
  subFunction2();
}

myBigFunction();
```

Στην περίπτωση αυτή το αποτέλεσμα του κώδικα θα ήταν η εκτύπωση της τιμής 1 δύο φορές.

3.3.6 Προαιρετικά ορίσματα συναρτήσεων

Η JavaScript πριν την έκδοση ES6 δεν υποστήριζε άμεσα προαιρετικά ορίσματα συναρτήσεων.

Η κλήση μιας συνάρτηση απαιτεί τον ορισμό τιμών σε όλα τα ορίσματα. Μάλιστα είναι αξιοσημείωτο ότι δεν προκαλείται εξαίρεση σε περίπτωση που δώσουμε περισσότερες τιμές από ότι τα ορίσματα.

```
function f(x,y,z){
    return x+y+z
}

f(1,2)
NaN
f(1,2,3,4,5)
6
```

Πριν την έκδοση ES6 αν επιθυμούσαμε σε κάποια ορίσματα να δώσουμε προκαθορισμένες τιμές αν ο χρήστης δεν τους δώσει τιμή κατά την κλήση της συνάρτησης, αυτό έπρεπε να γίνει με εσωτερικό έλεγχο που εντοπίζει τη μη χρήση του ορίσματος (που σε αυτή την περίπτωση έχει την τιμή `undefined`) και του αναθέτει μια προκαθορισμένη τιμή.

Να ένα παράδειγμα αυτής της προσέγγισης:

```
function f(optional) {
  if (typeof optional === 'undefined') optional = 1;
  return optional + 1;
}
```

```
f(3); // Επιστρέφεται 4
f(); // Επιστρέφεται 2
```

Επίσης ένας εναλλακτικός τρόπος να κάνουμε τον έλεγχο τιμής της μεταβλητής είναι να αντικαταστήσουμε την δεύτερη εντολή με την εξής ιδιωματική έκφραση:

```
  optional = optional || 1;
```

Ο τελεστής || επιστρέφει το πρώτο όρισμα αν είναι αληθές, αλλοιώς το δεύτερο, συνεπώς αν η παράμετρος `optional` δεν έχει πάρει τιμή, θα πάρει την τιμή 1.

Στην έκδοση ES6 επιτρέπονται πλέον προαιρετικά ορίσματα με προκαθορισμένες τιμές, κάτι αντίστοιχο με τα keyword arguments στη γλώσσα Python.

Το παραπάνω παράδειγμα στην νεότερη έκδοση της γλώσσας θα μπορούσε να γραφτεί απλούστερα ως εξής:

```
function f(optional = 1) {
  return optional + 1;
}
f(3); // Επιστρέφεται 4
f(); // Επιστρέφεται 2
```

Να σημειωθεί επίσης ότι τα προαιρετικά ορίσματα πρέπει να ακολουθούν τα υποχρεωτικά.

3.3.7 Ο τελεστής ... στα ορίσματα συνάρτησης

Ο τελεστής `...` (τελεστής ανάπτυξης, spread), είναι ένας τελεστής που χρησιμοποιείται για την ανάπτυξη ενός πίνακα η μιας ακολουθιακής δομής όπως μια συμβολοσειρά, στα επί μέρους στοιχεία από τα οποία απαρτίζεται.

Ο τελεστής αυτός είναι χρήσιμος σε διάφορες περιπτώσεις:

Αντίγραφο πίνακα Στον ορισμό ενός πίνακα, μπορούμε να μεταφέρουμε τα στοιχεία ενός πίνακα σε ένα άλλο, δημιουργώντας ένα αντίγραφό του.

Για παράδειγμα, το αποτέλεσμα του παρακάτω κώδικα:

```
const a = [1,2,3];
const b = [ ...a];
```

είναι ότι ο πίνακας b είναι ένας νέος πίνακας, πανομοιότυπο αντίγραφο του a. (να σημειωθεί ότι αν η δεύτερη εντολή αντικατασταθεί από την εντολή `const b = a;` δεν έχουμε το ίδιο αποτέλεσμα, αφού ο b είναι μια μεταβλητή που αναφέρεται στον ίδιο πίνακα όπως και η μεταβλητή a).

Πέρασμα των στοιχείων πίνακα ως ορίσματα συνάρτησης Μια δεύτερη χρήση του *τελεστή ανάπτυξης* είναι στα ορίσματα συναρτήσεων, όπου θέτει τα στοιχεία ενός πίνακα ως ξεχωριστά γνωρίσματα, όταν κάτι τέτοιο απαιτεί ο ορισμός της συνάρτησης.

Ας δούμε ένα παράδειγμα.

```
function f(x,y,z){
  return x+y+z;
}
const a = [5,6,7];
console.log(f(...a));
>18
```

Στην περίπτωση αυτή ``σπάμε'' τον πίνακα a στα επί μέρους στοιχεία του ώστε να τα περάσουμε ως ορίσματα στην συνάρτηση f().

Ορισμός συνάρτησης με απροσδιόριστο πλήθος στοιχείων

Μια τρίτη περίπτωση χρήσης του τελεστή ανάπτυξης είναι κατά τον ορισμό συνάρτησης με μεταβλητό πλήθος στοιχείων.

Στην περίπτωση αυτή ο τελεστής εφαρμόζεται σε μια μεταβλητή, την οποία μπορούμε στο σώμα της συνάρτησης να χειριστούμε ως πίνακα τιμών.

Ακολουθεί ένα παράδειγμα.

```
function max(...args) {
  let maxValue = args[0];
  for (let n of args) {
    if (n > maxValue) maxValue = n;
  }
  return maxValue;
}

max(10, 50, 60, 5, 8);
> 60
```

Στο παράδειγμα αυτό η μεταβλητή args εκπροσωπεί ένα πίνακα από τιμές μεταβλητού πλήθους.

3.3.8 Στατικές μεταβλητές συνάρτησης

Υπάρχουν περιπτώσεις που θα θέλαμε μια συνάρτηση να ``θυμάται'' τις προηγούμενες φορές που κλήθηκε. Αυτό μπορεί να έχει ενδιαφέρον σε κάποιες περιπτώσεις. Σε κάποιες γλώσσες προγραμματισμού αυτό επιτυγχάνεται με τις λεγόμενες στατικές μεταβλητές.

Ας υποθέσουμε ότι έχουμε μια συνάρτηση counter() η οποία κάθε φορά που την καλούμε επιστρέφει μια τιμή αυξημένη κατά 1 από την τελευταία φορά που την καλέσαμε.

Ένας απλός τρόπος να το πετύχουμε είναι να ορίσουμε μια μεταβλητή του *αντικειμένου* που εκπροσωπεί τη συνάρτηση.

Δεν θα πρέπει να ξεχνάμε ότι μια συνάρτηση είναι στην ουσία ένα object. Άρα μπορεί να έχει ιδιότητες, όπως όλα τα αντικείμενα.

```
function counter() {
 return ++counter.count;
}
counter.count = 0;

counter();  // επιστρέφει 1
counter(); // επιστρέφει 2
```

3.3.9 Οι συναρτήσεις ορίζουν μοναδικό χώρο ονομάτων

Υπάρχουν περιπτώσεις που θα θέλαμε ο κώδικας μας να μην έχει συγκρούσεις ως προς τα ονόματα μεταβλητών με άλλα τμήματα κώδικα που πιθανόν φορτωθούν μαζί, για παράδειγμα στο πλαίσιο μιας ιστοσελίδας.

Για να πετύχουμε αυτή την απομόνωση του χώρου ονομάτων μεταβλητών (namespace) του κώδικά μας, μπορούμε να ενσωματώσουμε τον κώδικα σε μια συνάρτηση, την οποία να καλέσουμε μόλις το DOM φορτωθεί, και μέσα στη συνάρτηση αυτή να ορίσουμε τις μεταβλητές, συναρτήσεις, αντικείμενα κλπ.

```
function pageScript(){
  // εδώ ορίζουμε μεταβλητές, συναρτήσεις,
  // χειριστές γεγονότων, κλπ
}

document.addEventListener("DOMContentLoaded", pageScript);
```

3.3.10 Μέθοδοι επεξεργασίας πινάκων

Η επεξεργασία των στοιχείων ενός πίνακα είναι συχνή στην JavaScript. Είτε με σκοπό τον μετασχηματισμό τους, είτε να τα χρησιμοποιήσει σε μια ακολουθία πράξεων που τα εμπλέκουν, πχ να βρεθεί το άθροισμά τους.

Ο πιο κλασικός τρόπος επεξεργασίας των στοιχείων ενός πίνακα ή γενικότερα μιας ακολουθιακής δομής, είναι η χρήση δομών επανάληψης, όπως η `for`.

Όμως η JavaScript, αναδεικνύοντας τον συναρτησιακό χαρακτήρα της, διαθέτει μεθόδους του αντικειμένου `Array` που επιτρέπουν πιο αποδοτικά και συνοπτικά να κάνουμε αυτή την επεξεργασία.

Οι μέθοδοι αυτές έχουν το χαρακτηριστικό ότι παίρνουν ως όρισμα συναρτήσεις οι οποίες εφαρμόζονται στα στοιχεία του πίνακα.

Παραδείγματα αυτών των μεθόδων είναι:

- myArray.**forEach**(myFunction) εφαρμόζει τη συνάρτηση σε κάθε στοιχείο του πίνακα

- myArray.**map**(myFunction) δημιουργεί νέο πίνακα με εφαρμογή της συνάρτησης σε κάθε στοιχείο του
- myArray.**filter**(myFunction) δημιουργεί νέο πίνακα με τα στοιχεία που ικανοποιούν τη συνάρτηση φίλτρο
- myArray.**reduce**(acumFun, αρχικήΤιμή) παράγει μια τιμή μετά από διαδοχική εφαρμογή της συνάρτησης acumFun στα στοιχεία του πίνακα.

Ας δούμε στη συνέχεια τυπικά παραδείγματα χρήσης των μεθόδων αυτών.

3.3.10.1 Η μέθοδος forEach

Η μέθοδος αυτή εφαρμόζει το όρισμα της σε ένα προς ένα τα στοιχεία του πίνακα, δεν επιστρέφει κάποια τιμή.

```
const myAr = ["Χίος", "Μυτιλήνη", "Σάμος"]
myAr.forEach((el, i) => console.log(i, 'Η ' + el));
> 0 'Η Χίος'
> 1 'Η Μυτιλήνη'
> 2 'Η Σάμος'
```

Η συνάρτηση που περνάμε ως όρισμα στην `forEach()` είναι η συνάρτηση που εφαρμόζεται στο αντίστοιχο στοιχείο, αν θέσουμε δεύτερο όρισμα, αυτό είναι ο δείκτης του στοιχείου, ενώ ένα τρίτο όρισμα αντιπροσωπεύει τον ίδιο τον πίνακα.

Να σημειωθεί ότι η μέθοδος αυτή δεν επηρεάζει τον ίδιο τον πίνακα στον οποίο εφαρμόζεται, ενώ δεν επιστρέφει τιμή.

Αν επιθυμούμε να τροποποιήσουμε τον ίδιο τον αρχικό πίνακα, μπορούμε να εκμεταλλευτούμε το γεγονός ότι το τρίτο όρισμα της συνάρτησης που περνάμε στην `forEach()` είναι ο ίδιος ο πίνακας.

Έτσι για παράδειγμα για να μετατρέψουμε τα ονόματα των νησιών σε κεφαλαία:

```
const myAr = ['Χίος', 'Μυτιλήνη', 'Σάμος'];
myAr.forEach((el, i, a) => {
  a[i] = a[i].toUpperCase();
});
console.log(myAr);
> [ 'ΧΙΟΣ', 'ΜΥΤΙΛΉΝΗ', 'ΣΆΜΟΣ' ]
```

3.3.10.2 Η μέθοδος map

Η μέθοδος αυτή επιστρέφει ένα νέο πίνακα που προκύπτει από την εφαρμογή της συνάρτησης που θέτουμε ως όρισμά της, σε κάθε στοιχείο του αρχικού πίνακα. Η μέθοδος `map()` είναι συνεπώς μια συνάρτηση μετασχηματισμού ενός πίνακα σε ένα νέο πίνακα.

```
const myAr = ['Χίος', 'Μυτιλήνη', 'Σάμος'];
const newAr = myAr.map((el, i) => i + ': η νήσος ' + el);
console.log(newAr);
> [ '0: η νήσος Χίος', '1: η νήσος Μυτιλήνη', '2: η νήσος Σάμος' ]
```

Όπως και στην προηγούμενη περίπτωση, η μέθοδος `map()` δεν επηρεάζει τον αρχικό πίνακα, όμως επιστρέφει ένα νέο πίνακα που προκύπτει από τον μετασχηματισμό.

Η συνάρτηση που περνάμε ως όρισμα στην map δέχεται τρία ορίσματα, το πρώτο είναι το εκάστοτε στοιχείο του πίνακα, το δεύτερο ο δείκτης του στοιχείου και το τρίτο ο ίδιος ο πίνακας.

3.3.10.3 Η μέθοδος filter

Η μέθοδος αυτή επιστρέφει επίσης ένα νέο πίνακα που προκύπτει από την εφαρμογή της συνάρτησης που θέτουμε ως όρισμά της, η οποία δρα ως φίλτρο, σε κάθε στοιχείο του αρχικού πίνακα. Τα στοιχεία εκείνα για τα οποία η συνάρτηση αυτή επιστρέφει την τιμή true, περιέχονται στον νέο πίνακα, ενώ εκείνα που επιστρέφουν false όχι. Η μέθοδος `filter()` είναι συνεπώς μια συνάρτηση μετασχηματισμού ενός πίνακα σε ένα νέο πίνακα.

```
const myAr = ['Χίος', 'Μυτιλήνη', 'Σάμος'];
const newAr = myAr.filter((el) => el.length > 5);
console.log(newAr);
> [ 'Μυτιλήνη' ]
```

Όπως και στην προηγούμενη περίπτωση, η μέθοδος `filter()` δεν επηρεάζει τον αρχικό πίνακα, όμως επιστρέφει ένα νέο πίνακα που προκύπτει από τον μετασχηματισμό.

Η συνάρτηση που περνάμε ως όρισμα στην filter() δέχεται ως όρισμα το εκάστοτε στοιχείο του πίνακα, και πρέπει να επιστρέφει τιμή true/false.

3.3.10.4 Η μέθοδος reduce

Η μέθοδος αυτή διαφέρει από τις προηγούμενες, αφού επιστρέφει μια τιμή και όχι ένα πίνακα. Η τιμή αυτή προκύπτει από την εφαρμογή της συνάρτησης (πρώτο όρισμά της), διαδοχικά σε κάθε στοιχείο του αρχικού πίνακα.

Η συνάρτηση πρώτο όρισμα της `reduce()` παίρνει τα εξής ορίσματα: * ως πρώτο όρισμα ένα συσσωρευτή που διαδοχικά συσσωρεύει το αποτέλεσμα των προηγούμενων πράξεων, * ως δεύτερο όρισμα το εκάστοτε στοιχείο του πίνακα, * προεραιτικά μπορεί να πάρει ακόμη τον δείκτη στο εκάστοτε στοιχείο και τον ίδιο τον πίνακα.

Η μέθοδος αυτή παίρνει ως δεύτερο όρισμα την αρχική τιμή του συσσωρευτή.

Ακολουθεί ένα παράδειγμα.

```
const myAr = ['Χίος', 'Μυτιλήνη', 'Σάμος'];
const result = myAr.reduce((islands, el) => {
  return (islands += ' ' + el);
}, '');
console.log(result);
> "Χίος Μυτιλήνη Σάμος"
```

Όπως και στην προηγούμενη περίπτωση, η μέθοδος `filter()` δεν επηρεάζει τον αρχικό πίνακα, όμως επιστρέφει την τελική τιμή σου συσσωρευτή.

3.3.11 Παραδείγματα

Έστω πίνακας που περιέχει ένα σύνολο αριθμητικών τιμών. Ως παράδειγμα ας υποθέσουμε ότι έχουμε τον παρακάτω πίνακα:

```
const a = [5, 10, 18, 32, 20, 44];
```

3.3.11.1 Παράδειγμα 1: Υπολογισμός αθροίσματος στοιχείων πίνακα

Για τον υπολογισμό του αθροίσματος, θα εφαρμόσουμε την `reduce()` στον πίνακα, χρησιμοποιώντας έναν αθροιστή που αρχικά έχει την τιμή μηδέν, και στον οποίο διαδοχικά αθροίζουμε τα στοιχεία.

```
const result = a.reduce((s, el) => s + el, 0);
console.log(result);
> 129
```

3.3.11.2 Παράδειγμα 2: Εύρεση ελάχιστης τιμής

Και στην περίπτωση αυτή θα εφαρμόσουμε την `reduce()` στον πίνακα, χρησιμοποιώντας έναν συσσωρευτή που αρχικά έχει την τιμή `Infinity`, στη συνέχεια για κάθε στοιχείο, ελέγχουμε αν το είναι μικρότερο από τον συσσωρευτή, αν ναι το στοιχείο παίρνει τη θέση του συσσωρευτή.

```
const result = a.reduce((s, el) => (el < s ? el : s), Infinity);
console.log(result);
> 5
```

με αντίστοιχο τρόπο βρίσκουμε και την μέγιστη τιμή του πίνακα.

3.3.11.3 Παράδειγμα 3: υπολογισμός τυπικής απόκλισης συνόλου τιμών

Έστω πίνακας με ένα σύνολο αριθμητικών τιμών. Ζητείται να ορισθεί συνάρτηση που υπολογίζει την τυπική απόκλιση.

Υπενθυμίζεται ότι ο τύπος της τυπικής απόκλισης είναι:

όπου x είναι μια τιμή, μ η μέση τιμή και N το πλήθος των τιμών.

Κατ' αρχάς η *μέση τιμή* μ των τιμών του a μπορεί να βρεθεί με επαναληπτική διαδικασία:

```
let mean = 0
for (let i of a) mean += i;
mean = mean/a.length
```

Ένας εναλλακτικός τρόπος υπολογισμού είναι με χρήση της μεθόδου `reduce()`:

```
mean = a.reduce((s, i) => s + i, 0) / a.length;
```

Για να υπολογίσουμε τον όρο Σ(|x-μ|)2 , μπορούμε επίσης να εφαρμόσουμε συναρτησιακή προσέγγιση με χρήση της `map()` για παραγωγή μιας ακολουθίας τετραγώνων:

```
console.log(a.map((x) => Math.pow(x - mean, 2)));
> [ 272.25, 132.25, 12.25, 110.25, 2.25, 506.25 ]
```

στη συνέχεια δε με χρήση της `reduce()` να υπολογίσουμε το άθροισμα των τετραγώνων, και την τετραγωνική ρίζα του αθροίσματος δια του πλήθους.

```
const s1 = a.map((x) => Math.pow(x - mean, 2));
const sd = Math.sqrt(s1.reduce( (a,b)=> a+b, 0)/a.length)
> 13.13709759929237
```

Συνοψίζοντας μπορούμε να δημιουργήσουμε μια συνάρτηση ως εξής:

```
function standardDeviation(a) {
  const mean = a.reduce((s, i) => s + i, 0) / a.length;
  const s1 = a.map((x) => Math.pow(x - mean, 2));
  return Math.sqrt(s1.reduce((a, b) => a + b, 0) / a.length);
}
```

3.3.11.4 Παράδειγμα 4. Τυχαία αναδιάταξη στοιχείων πίνακα

Έστω ότι ζητείται να αναδιατάξουμε με τυχαίο τρόπο τα στοιχεία ενός πίνακα. Αναζητήσετε διαφορετικούς τρόπους και ελέγξετε την τυχαιότητα της λύσης.

Υποθέτουμε για λόγους απλότητας ότι έχουμε τον πίνακα:

```
const a = [1,2,3]
```

Ως πρώτη προσέγγιση στο πρόβλημα της αναδιάταξης των τιμών με τυχαίο τρόπο, υποθέτουμε τη χρήση της συνάρτησης:

```
function shuffle1(array) {
  array.sort(() => Math.random() - 0.5);
}
```

θεωρώντας ότι η Math.random() επιστρέφει τιμές ομοιόμορφα κατανεμημένες στο διάστημα [0..1], η Math.random() - 0.5 θα έχει κατά τυχαίο τρόπο θετικό ή αρνητικό πρόσημο, άρα θα ταξινομεί δύο τυχαία στοιχεία σε αύξουσα ή φθίνουσα σειρά.

Εφαρμόζουμε τη συνάρτηση αυτή στον πίνακα a διαδοχικά για μεγάλο πλήθος επαναλήψεων, και στη συνέχεια ελέγχουμε αν οι διαφορετικές διατάξεις έχουν παρόμοιες συχνότητες.

```
const count = {};
for (let _ = 0; _ < 10000; _++) {
  const a = [1, 2, 3];
  const result = shuffle1(a).join('');
  count[result] = result in count ? count[result] + 1 : 1;
}
console.log(count);
console.log(`SD=${standardDeviation(Object.values(count))}`);
```

Το αποτέλεσμα που προκύπτει είναι:

```
{'123': 3809, '132': 642, '213': 1183
 '231': 633, '312': 634, '321': 3099
}
'SD=1294.856062356825'
```

Είναι φανερό ότι κάποιες τιμές έχουν πολύ υψηλότερη συχνότητα εμφάνισης, άρα αποκαλύπτεται μια αδυναμία του συγκεκριμένου αλγορίθμου.

Μια δεύτερη προσπάθεια γίνεται με τον αλγόριθμο Fisher-Yates:

```
function shuffle2(a) {
  //  αλγόριθμος Fisher-Yates
  for (let i = a.length - 1; i > 0; i--) {
    let j = Math.floor(Math.random() * (i + 1)); // random index from 0 to i
    [a[i], a[j]] = [a[j], a[i]];
  }
  return a;
}
```

Ο επαναληπτικό αυτός αλγόριθμος, σε κάθε βήμα για i = από ν-1 μέχρι 1, κάνει αντιμετάθεση ενός τυχαίου στοιχείου που βρίσκεται σε τυχαία θέση από 0 μέχρι i με το στοιχείο i.

Σε αυτή την περίπτωση, ο ίδιος έλεγχος δίνει τα εξής αποτελέσματα:

```
{ '123': 1669, '132': 1643, '213': 1644,
  '231': 1623, '312': 1701, '321': 1720
}
'SD=34.179265969622904'
```

Παρατηρείται μια σαφής βελτίωση έναντι της προηγούμενης περίπτωσης.

3.4 Αντικείμενα και κλάσεις

Τα *αντικείμενα* είναι ο πιο βασικός τύπος δεδομένων της JavaScript.

Ένα αντικείμενο στην JavaScript είναι ένας σύνθετος τύπος δεδομένων που περιέχει ιδιότητες που έχουν ως τιμή είτε πρωτογενή δεδομένα ή άλλα αντικείμενα. Οι ιδιότητες ενός αντικειμένου δεν είναι ταξινομημένες, και έχουν τη μορφή `ιδιότητα: τιμή`. Βεβαίως η τιμή κάποιων ιδιοτήτων μπορεί να είναι συναρτήσεις (που σε αυτή την περίπτωση λέγονται *μέθοδοι*). Αρχικά τα αντικείμενα της JavaScript μοιάζουν με τα λεξικά της Python, ή άλλες αντίστοιχες δομές, όπως πίνακες κατακερματισμού σε άλλες γλώσσες προγραμματισμού.

```
const car = {
  make: "volvo",
  speed: 140,
  engine: {
    size:1800,
    fuel: "diesel",
    pistons: ["piston1", "piston2"]},
  drive: function() {
    return `οδηγώ ${this.make}...`}}
}
```

Στο παράδειγμα αυτό το αντικείμενο `car` έχει 4 ιδιότητες, από αυτές η μια έχει ως τιμή ένα άλλο αντικείμενο, και η άλλη μια μέθοδο.

Στις ιδιότητες ενός αντικειμένου μπορούμε να αναφερθούμε με δύο τρόπους, με σημειολογία τελείας:

```
console.log(car.make);
>'volvo'
```

Εναλλακτικά μπορούμε να αναφερθούμε στην ιδιότητα με τετράγωνες αγκύλες ως εξής:

```
console.log(car["make"]);
>'volvo'
```

Όμως τα αντικείμενα της JavaScript δεν είναι απλά πίνακες αντιστοίχισης κλειδιών-τιμών. Κάθε αντικείμενο συνοδεύεται από μια πρόσθετη ιδιότητα που έχει την τιμή `prototype` που είναι ένα αντικείμενο, του οποίου

κληρονομεί τις ιδιότητες. Αυτή είναι μια ιδιαιτερότητα της JavaScript που δεν συναντάται σε άλλες γλώσσες προγραμματισμού. Ο μηχανισμός αυτός λέγεται ``*κληρονομικότητα μέσω προτοτύπου*''.

Για τα ονόματα των ιδιοτήτων των αντικειμένων ισχύουν όσα ισχύουν για τις μεταβλητές της JavaScript.

Κάθε ιδιότητα ενός αντικειμένου, εκτός από *όνομα* και *τιμή* έχει ακόμη τα εξής γνωρίσματα: *writable* (αν μπορεί να αλλάξει η τιμή), *enumerable* (αν αφορά ιδιότητα που θα εμφανίζεται σε βρόχο for), *configurable* (αν μπορεί να διαγραφεί). Να σημειωθεί ότι τα αντικείμενα της γλώσσας (Array, Number, Function, κλπ), δεν επιτρέπουν τη διαγραφή των ιδιοτήτων ή την τροποποίησή τους, κάτι που όμως επιτρέπεται για τα αντικείμενα των χρηστών.

Έτσι στα αντικείμενα των χρηστών, των οποίων οι ιδιότητες είναι enumerable, μπορούμε να εφαρμόσουμε ένα βρόχο **for/in** ως εξής:

```
for (property in car) {
  console.log(property, car[property]);
}
> 'make' 'volvo'
> 'speed' 140
> 'engine' { size: 1800, fuel: 'diesel', pistons: [ 'piston1', 'piston2' ] }
> 'drive' ƒ drive()'make'
```

Θα πρέπει να σημειώσουμε επίσης ότι τις ιδιότητες ενός αντικειμένου (μόνο τις enumerable) μπορούμε να τις ανακτήσουμε μέσω της μεθόδου `Object.keys()`

```
Objec.keys(car);
> [ 'make', 'speed', 'engine', 'drive' ]
```

Επίσης θα πρέπει να αναφερθεί ότι και ένας πίνακας έχει ως ιδιότητες τους δείκτες 0,1,2 .. άρα:

```
const ar = [10, 20, 30];
Object.keys(ar);
> [ '0', '1', '2' ]
```

3.4.1 Μετατροπή αντικειμένων σε JSON

Η μετατροπή ενός αντικειμένου JS σε μια συμβολοσειρά από την οποία εν συνεχεία μπορεί να ανακτηθεί λέγεται σειριοποίηση (serialization). Η διαδικασία αυτή είναι χρήσιμη γιατί η μεττροπή του αντικειμένου σε ακολουθία χαρακτήρων μάς επιτρέπει να το μεταβιβάσουμε σε ένα παραλήπτη ή να το αποθηκεύσουμε. Έχει οριστεί η JSON (JavaScript Object Notation), στην οποία μπορούμε να μετατρέψουμε τα αντικείμενα της JavaScript κατά τη σειριοποίησή τους. Η JSON είναι ένα πρότυπο ανταλλαγής δεδομένων με ευρεία χρήση, πέραν της JavaScript.

Η μετατροπή ενός αντικειμένου σε μορφή JSON γίνεται με τη μέθοδο `JSON.stringify(obj)`, ενώ η αντίθετη μετατροπή γίνεται με τη μέθοδο `JSON.parse(st)`.

Η μετατροπή αντικειμένων σε συμβολοσειρές δεν καλύπτει όμως όλες τις περιπτώσεις αντικειμένων της JavaScript, ώστε να μπορέσουμε να ανακτήσουμε την αρχική μορφή του αντικειμένου.

Τα αντικείμενα, πίνακες, συμβολοσειρές, αριθμοί, οι λογικές τιμές true/false, και null μπορούν να μετατραπούν σε JSON και να ανακτηθούν.

Όμως τιμές NaN, Infinity, και -Infinity μετατρέπονται όλα σε null και δεν μπορεί έτσι να ανακτηθεί η αρχική τους τιμή.

Αντικείμενα τύπου Date μετατρέπονται σε συμβολοσειρές για ημερομηνία κατά το πρότυπο ISO (σύμφωνα με την Date.toJSON()), όμως η JSON.parse() δεν επαναφέρει την ημερομηνία από τη συμβολοσειρά αυτή. Τέλος συναρτήσεις, κανονικές εκφράσεις, και αντικείμενα τύπου Error objects, καθώς και τιμές undefined δεν μπορούν να μετατραπούν σε JSON, ούτε βεβαίως να ανακτηθούν από JSON.

3.4.2 Δημιουργία κλάσεων και αντικειμένων

Ο πιο απλός τρόπος δημιουργίας αντικειμένων είναι με απευθείας ορισμό τους μέσα σε άγκιστρα που περιλαμβάνουν ακολουθία από ζευγάρια `ιδιότητα: τιμή`. Με τον τρόπο αυτό ορίστηκε το αντικείμενο `car` στην προηγούμενη ενότητα.

Για παράδειγμα το πιο απλό αντικείμενο ορίζεται ως εξής:

```
const ob = {};
```

Το αντικείμενο αυτό φαίνεται να μην έχει ιδιότητες, όμως όπως ήδη αναφέρθηκε περιλαμβάνει την έξτρα ιδιότητα του πρωτοτύπου. Έτσι αν στην κονσόλα ζητήσουμε να δούμε το περιεχόμενο αυτού του αντικειμένου, παρουσιάζεται η εξής εικόνα:

```
> ob
{}
__proto__: Object
```

Ένας δεύτερος τρόπος για να δημιουργήσουμε ένα αντικείμενο είναι με χρήση του τελεστή `new` που ακολουθείται από μια συνάρτηση. Ο τελεστής αυτός δημιουργεί ένα καινούργιο αντικείμενο. Η συνάρτηση που ακολουθεί λέγεται δημιουργός (constructor) και χρησιμεύει για να αρχικοποιήσει το καινούργιο αντικείμενο.

Υπάρχουν δημιουργοί για τα εγγενή αντικείμενα της γλώσσας, όπως οι δημιουργοί `Object(), Array(), Date()`, κλπ. Όπως θα δούμε στη συνέχεια, μπορούμε και εμείς να ορίσουμε συναρτήσεις δημιουργούς αντικειμένων.

Για να ορίσουμε μια δική μας κλάση αντικειμένων, που έχουν παρόμοια δομή, πρέπει να ορίσουμε μια συνάρτηση η οποία θα δημιουργεί τα αντικείμενα αυτά.

Έστω ότι επιθυμούμε να δημιουργήσουμε μια κλάση αυτοκινήτων που αφορά αντικείμενα της μορφής:

```
const myCar = {
  make: "VW",
  speed: 140,
  drive: function(){
    return `οδηγώ ${this.make}...`}
}
```

Επιθυμούμε να ορίσουμε μια συνάρτηση δημιουργό `Car()` η οποία θα παράγει αυτοκίνητα, στην οποία να περνάμε τις τιμές των ιδιοτήτων ενός αντικειμένου (στιγμιότυπου της κλάσης) ως εξής:

```
const myCar = new Car("VW", 140);
const myOtherCar = new Car("Porshe", 350);
```

Η συνάρτηση αυτή ορίζεται ως εξής:

```
function Car(make, speed){
  this.make = make;
  this.speed = speed;
}
```

Παρατηρούμε τη χρήση της λέξης `this` για αναφορά στο εκάστοτε στιγμιότυπο της κλάσης. Ας χρησιμοποιήσουμε τώρα τη συνάρτηση αυτή για να κατασκευάσουμε ένα αντικείμενο `myCar` και ας εξερευνήσουμε τη δομή του νέου αντικειμένου :

```
const myCar = new Car("VW", 200);
> myCar
Car {make: "VW", speed: 200}
  make: "VW"
  speed: 200
  __proto__:
    constructor: ƒ Car(make, speed)
    __proto__: Object
```

Για να ελέγξουμε μάλιστα αν ένα αντικείμενο ανήκει σε ορισμένη κλάση, χρησιμοποιούμε τον τελεστή `instarnceof`:

```
myCar instanceof Car;
> true
```

Αυτό είναι ένα πρώτο παράδειγμα χρήσης της συνάρτησης δημιουργού. Η συνάρτηση Car(), όπως παρατηρούμε δεν έχει την ίδια συμπεριφορά με τις συναρτήσεις που έχουμε δει ως τώρα. Αν και δεν περιλαμβάνει εντολή return, μάς επιστρέφει ένα νέο αντικείμενο όταν τη χρησιμοποιούμε με τον τελεστή `new`. Είναι ο τελεστής `new` που δίνει σε αυτή τη συνάρτηση ειδική χρήση, κάνει τη συνάρτηση Car() δημιουργό συνάρτηση αντικειμένων.

Επίσης παρατηρούμε ότι το αντικείμενο που δημιουργήσαμε εκτός από τις δύο ιδιότητες που ορίσαμε στον δημιουργό του, έχει μια ακόμη ιδιότητα, την `__proto__` που δηλώνει το *πρωτότυπο* του αντικειμένου,

κληρονομεί από το αντικείμενο Object και έχει ως δημιουργό του τη συνάρτηση Car() δηλαδή τη δημιουργό συνάρτηση της κλάσης μας. Το ``πρωτότυπο'' αυτό δημιουργήθηκε λοιπόν από τη συνάρτηση δημιουργό μας. Θα πρέπει να σημειωθεί βεβαίως ότι η __proto__ δεν είναι πραγματικά ιδιότητα αλλά αναφορά στον πρωτότυπο του δημιουργού, δεν μπορούμε να την τροποποιήσουμε και το αντικείμενο αν ερωτηθεί σχετικά δεν την αναγνωρίζει:

```
myCar.hasOwnProperty("make");
> true
myCar.hasOwnProperty("__proto__");
> false
```

3.4.3 Ορισμός μεθόδων

Ένα ακόμη θέμα που πρέπει να δούμε είναι πώς ορίζουμε τις μεθόδους μιας κλάσης αντικειμένων. Έστω ότι θέλουμε τα αντικείμενά μας να έχουν την μέθοδο drive().

Αυτήν μπορούμε να την ορίσουμε στο πρωτότυπο της κλάσης ώστε να την κληρονομήσουν όλα τα στιγμιότυπά της:

```
Car.prototype.drive: function() {
    return `οδηγώ ${this.make}...`;
  }

myCar.drive()
> 'οδηγώ VW...'
```

Αν στην κονσόλα ζητήσουμε τη δομή του αντικειμένου myCar, αυτή τώρα είναι:

```
> myCar
Car {make: "VW", speed: 200}
  make: "VW"
  speed: 200
  __proto__:
    drive: ƒ ()
    __proto__: Object
      constructor: ƒ Car(make, speed)
```

Όπως βλέπουμε, έχει προστεθεί η ιδιότητα drive στο πρωτότυπο, η τιμή της οποίας είναι η μέθοδος που ορίσαμε.

Στο μέλλον μπορούμε να προσθέτουμε ιδιότητες ή μεθόδους στο πρωτότυπο της κλάσης Car, και αυτές θα κληρονομούνται από όλα τα στιγμιότυπα.

Ας δούμε ένα εναλλακτικό τρόπο ορισμού της μεθόδου drive() μέσα στη δημιουργό συνάρτηση της κλάσης Car :

```
function Car(make, speed) {
  this.make = make;
  this.speed = speed;
  this.drive = function () {
    return `οδηγώ ${this.make}...`;
  };
}
```

Σε αυτή την περίπτωση το αντικείμενο έχει την εξής δομή:

```
myCar
> Car {make: "VW", speed: 200, drive: ƒ}
  drive: ƒ ()
  make: "VW"
  speed: 200
  __proto__:
    constructor: ƒ Car(make, speed)
    __proto__: Object
```

Υπάρχει μια μικρή διαφορά μεταξύ των δύο τρόπων ορισμού μιας μεθόδου, που είδαμε ως τώρα. Με το δεύτερο τρόπο ορισμού της μεθόδου `drive()`, η μέθοδος είναι η τιμή της ιδιότητας drive του αντικειμένου. Πρακτικά αυτό σημαίνει ότι κάθε φορά που δημιουργούμε ένα νέο αντικείμενο με τον δημιουργό `Car()` δημιουργούμε ένα νέο αντίγραφο της συνάρτησης drive. Άσκοπη δαπάνη μνήμης. Ενώ στην πρώτη προσέγγιση είχαμε μόνο ένα αντίγραφο της μεθόδου στο αντικείμενο `Car.prototype` από το οποίο κληρονομούν τη μέθοδο όλα τα στιγμιότυπα τύπου Car. Εκεί όμως έχουμε καθυστέρηση κάθε φορά που καλούμε τη μέθοδο, την αναζητάμε πρώτα στο στιγμιότυπο και στη συνέχεια στο πρωτότυπο. Ότι χάνουμε σε χώρο το κερδίζουμε σε χρόνο.

3.4.4 Όρισμός κλάσεων με τη λέξη κλειδί class

Μια από τις αλλαγές που έφερε η ES6 είναι η εισαγωγή μιας νέας σύνταξης για ορισμό κλάσεων που περιλαμβάνει τη λέξη κλειδί `class`. Η σύνταξη αυτή δεν αλλάζει την ουσία του μηχανισμού δημιουργίας αντικειμένων που αναφέρθηκε, που στηρίζεται στα πρωτότυπα, όμως κάνει πιο απλή τη σύνταξη και κάνει την JavaScript να μοιάζει πιο πολύ με άλλες γλώσσες προγραμματισμού.

```
class Car{
  constructor(make, speed){
    this.make = make;
    this.speed = speed;
  }
  drive(){
    return `οδηγώ ${this.make}...`;
  }
```

```
}

const myCar = new Car("VW", 140);

myCar.drive();
> "οδηγώ VW..."
```

Παρατηρούμε ότι η σύνταξη της `class` έχει κάποιες ιδιαιτερότητες: Περιλαμβάνει χρήση της λέξης class ακολουθούμενης από το όνομα της κλάσης και τον ορισμό της μέσα σε άγκιστρα. Μέσα στο σώμα περιλαμβάνουμε τη δημιουργό συνάρτηση με τη λέξη `constructor()` και στη συνέχει ορισμό των μεθόδων χωρίς τη χρήση της λέξης κλειδί `function`. Δεν βάζουμε κόμμα ανάμεσα στις μεθόδους ή στον constructor και τις μεθόδους.

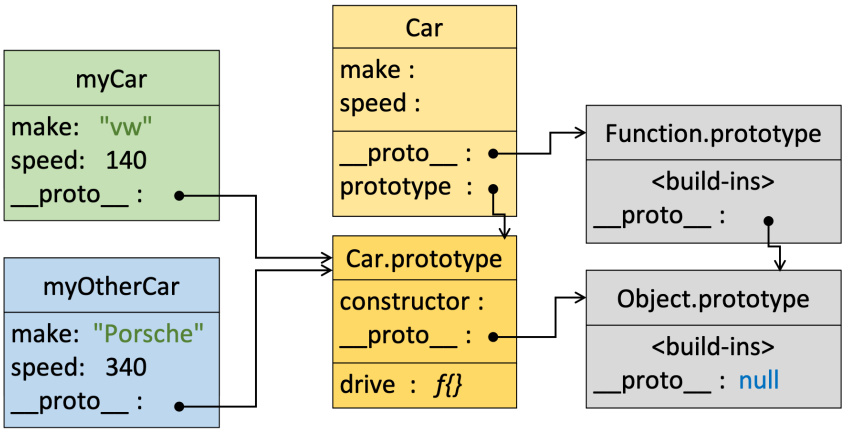
Να σημειωθεί εδώ ότι αυτός ο ορισμός παράγει αντικείμενα με τη μέθοδο στο πρωτότυπο (πρώτη μέθοδος της προηγούμενης ενότητας).

Έτσι αν δημιουργήσουμε ένα αντικείμενο με χρήση αυτής της κλάσης το περιεχόμενο του είναι:

```
const myCar = new Car("VW", 140);
> myCar
Car {make: "VW", speed: 140}
  make: "VW"
  speed: 140
  __proto__:
    constructor: class Car
    drive: ƒ drive()
    __proto__: Object
```

Παρατηρούμε ότι η μέθοδος drive() ανήκει στο αντικείμενο της ιδιότητας `__proto__`, δηλαδή στο πρωτότυπο αντικείμενο της κλάσης.

Αν θέλαμε να αποδώσουμε σχηματικά τη σχέση μεταξύ του δημιουργού και του πρωτοτύπου αυτή φαίνεται στην παρακάτω εικόνα για την περίπτωση της κλάσης `Car`.



Σχήμα 3.1: Δημιουργός, πρωτότυπο για την κλάση Car.

Ο δημιουργός της κλάσης, δηλαδή η συνάρτηση `Car`, έχει την ιδιότητα `prototype` η οποία έχει τιμή το πρωτότυπο της συγκεκριμένης κλάσης. Να σημειωθεί ότι όλες οι συναρτήσεις έχουν ιδιότητα `prototype`. Το αντικείμενο αυτό έχει ως δημιουργό του την κλάση Car(). Κάθε στιγμιότυπο που δημιουργούμε με την κλάση Car, κληρονομεί το πρωτότυπο αυτό, οι ιδιότητες του οποίου μπορεί να είναι νέες μέθοδοι που δημιουργούνται ως εξής:

```
Car.prototype.newMethod = function() { ... }
```

Το ίδιο αποτέλεσμα μπορούμε να έχουμε με έμεση αναφορά στο πρωτότυπο αυτό, για παράδειγμα αν δημιουργήσουμε την μέθοδο ως

```
myCar.__proto__.newMethod = function() { ... }
```

κάνουμε έμμεση αναφορά στο πρωτότυπο Car.prototype.

Τέλος να αναφερθεί ότι η συνάρτηση δημιουργός Car() όπως και όλες οι συναρτήσεις, κληρονομούν το πρωτότυπο `Function.prototype`, ενώ το αντικείμενο `Car.prototype` κληρονομεί το πρωτότυπο `Object.prototype`, το οποίο ως πρωτότυπο του αρχέγονου αντικειμένου δεν κληρονομεί από κανένα αντικείμενο, και για αυτό η ιδιότητά του `__proto__` έχει την τιμή null.

3.4.5 Κληρονομικότητα κλάσεων

Αν επιθυμούμε να ορίσουμε μια υποκλάση μιας κλάσης, αυτό γίνεται με χρήση του τελεστή `extends`.

Ας δούμε ένα παράδειγμα, έστω μια κλάση `Person` η οποία ορίζεται ως ακολούθως:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
```

```
  }
}
```

Ας υποθέσουμε ότι θέλουμε να δημιουργήσουμε μια υποκλάση της `Person` που την εξειδικεύει για την περίπτωση δασκάλων. Έστω λοιπόν η κλάση `Teacher` που ορίζεται ως εξής:

```
class Teacher extends Person {
  constructor(name, age, school) {
    super(name, age);
    this.school = school;
  }
}
```

Παρατηρούμε ότι η κλάση `Teacher` ορίζεται ως επέκταση της κλάσης `Person`. Ο δημιουργός αντικειμένων της νέας αυτής κλάσης κάνει αναφορά στην αρχική κλάση για τις ιδιότητες name, age που είναι ιδιότητες όλων των αντικειμένων τύπου `Person`, αυτό γίνεται με κλήση της μεθόδου `super()` που εκπροσωπεί τη δημιουργό της υπερκλάσης. Αν θέλαμε να αναφερθούμε σε επί μέρους μεθόδους της υπερκλάσης θα μπορούσαμε να το κάνουμε με σημειογραφία τελείας: `super.μέθοδος()`.

Για να δημιουργήσουμε ένα αντικείμενο της κλάσης `Teacher`, αυτό γίνεται ως εξής:

```
const t = new Teacher('Κώστας', 40, '1ο Λύκειο');

>t
Teacher {name: "Κώστας", age: 40, school: "1ο Λύκειο"}
  age: 40
  name: "Κώστας"
  school: "1ο Λύκειο"
  __proto__: Person
    constructor: class Teacher
    __proto__: Object
```

Στο παραπάνω απόσπασμα φαίνεται το περιεχόμενό του, όπως προκύπτει οι ιδιότητες `name, age` έχουν δημιουργηθεί από τον δημιουργό `super()` της υπερκλάσης, ενώ η ιδιότητα `school` προστέθηκε από τη δημιουργό της `Teacher`.

3.4.6 Αναφορές

- Διάκριση μεταξύ `__proto__` και της ιδιότητας `prototype` στην JavaScript, σχετική συζήτηση στο stackoverflow
- JavaScript. The Core: 2nd Edition Dmitry Soshnikov

3.5 Σύνοψη

Στο κεφάλαιο αυτό είδαμε διάφορους τύπους δεδομένων αναφοράς της JavaScript. Αρχικά είδαμε τον τύπο Array (πίνακας), ως πρώτο τύπο δεδομένων αναφοράς. Έγινε εκτενής αναφορά στις μεθόδους του τύπου Array, και τους τρόπους ορισμού πινάκων. Οι πίνακες είναι ειδική κατηγορία αντικειμένων (objects). Στη συνέχεια εστιάσαμε στη συναρτισιακή λειτουργία της γλώσσας, που ορίζει τις συναρτήσεις ως αντικείμενα. Επίσης είδαμε τους τρόπους που διαθέτουμε στη JS για δημιουργία αντικείμενων και προγραμματισμό με το αντικειμενοστραφές μοντέλο. Σε αυτό το σημείο έχουμε καλύψει σε μεγάλο βαθμό τη γλώσσα. Στο επόμενο κεφάλαιο θα εμβαθύνουμε στον ασύγχρονο χαρακτήρα της, που την κάνει μια ενδιαφέρουσα τεχνολογία για την πλευρά του εξυπηρετητή, που αποτελεί το αντικείμενων των τελευταίων κεφαλαίων του βιβλίου.

4 Ασύγχρονη JavaScript - διαχείριση συμβάντων

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 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 JavaScript έχει δύο μηχανισμούς ασύγχρονης εκτέλεσης:

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

4.1 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, ή μετά από ένα συμβάν, το οποίο θα προκληθεί από κάποιον έξω προς το περιβάλλον της γλώσσας, πχ. από τον χρήστη, από το δίκτυο, ή από άλλες διεργασίες του λειτουργικού συστήματος.

Δεν θα πρέπει να ξεχνάμε ότι η τυπική λειτουργία ενός κώδικα JavaScript στον φυλλομετρητή, όπως ήδη περιγράψαμε, είναι ότι αφού περάσει αρχικά από τη φάση φορτώματος του κώδικα και 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 αντικειμένου window: **`setTimeout(f, t)`**, η οποία επιτρέπει την εκτέλεση μιας συνάρτησης f μετά παρέλευση ορισμένου χρόνου t, ή η **`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).

Ο πιο απλός μηχανισμός ασύγχρονης εκτέλεσης κώδικα γίνεται με κλήση των μεθόδων του global object `setTimeout()` και `setInterval()` που περιγράφονται στη συνέχεια.

4.1.1 Συναρτήσεις `setTimeout()` και `setInterval()`

4.1.1.1 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 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 αυτής 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
```

4.1.1.2 Επαναληπτική κλήση συνάρτησης με κάποιο διάλειμμα

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

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 το λόγο αυτό χρησιμοποιείται σε κίνηση αντικειμένων (animations).

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

```
const id = setIterval(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 '
```

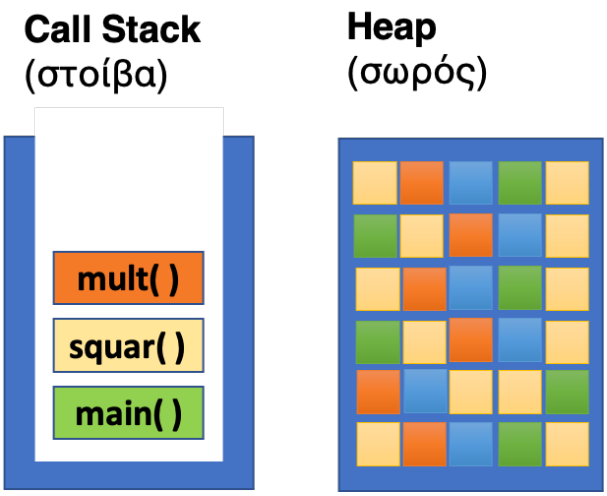
Πριν προχωρήσουμε σε πιο λεπτομερή περιγραφή των μηχανισμών ασύγχρονης λειτουργίας που διαθέτει η γλώσσα, θα πρέπει να περιγράψουμε τον μηχανισμό που ονομάζεται βρόχος ελέγχου συμβάντων (event loop).

4.1.2 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 μία 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 στο παρακάτω σχήμα:



Σχήμα 4.1: Δομές μνήμης κατά την εκτέλεση ενός προγράμματος JavaScript: στοίβα κλήσεων και σωρός. Η `main()` έχει καλέσει την `squar` και αυτή με τη σειρά της την `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 στην `squar()`.

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

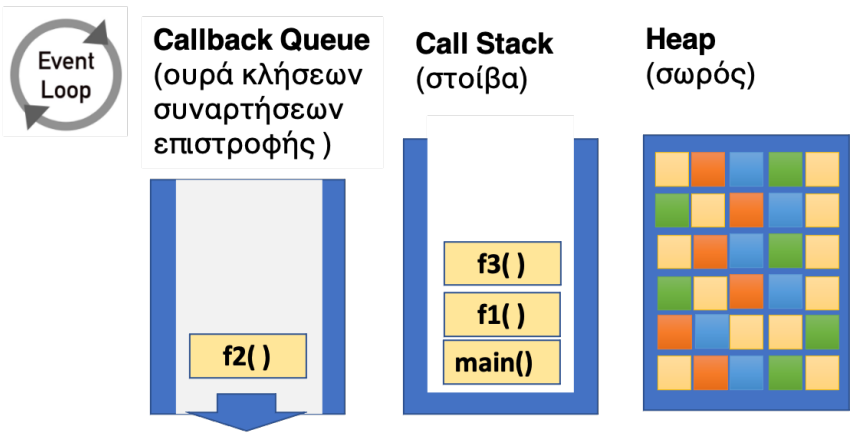
```
function mult(a,b) {
    return a*b
};

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

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

Στο σχήμα 4.1 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()`, η οποία 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.



Σχήμα 4.2: Δομές μνήμης κατά την εκτέλεση ενός προγράμματος JavaScript 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. Στο σχήμα αυτό φαίνεται ότι η συνάρτηση `f2()` τη στιγμή αυτή είναι σε αναμονή στην ουρά αυτή. 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 την 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 στην 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.

4.2 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, για παράδειγμα ως αποτέλεσμα ενεργειών του χρήστη. Το πρόγραμμά μας έχει καταχωρήσει ``χειριστές'', συναρτήσεις δηλαδή που περιμένουν αυτά τα συμβάντα. Οι χειριστές καταχωρούνται σε ένα **πίνακα συμβάντων**. Όταν προκύψει το συμβάν, η JavaScript ανατρέχει στο 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 των 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 ένα 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.

4.2.1 Κατηγορίες συμβάντων

Τα πιο σημαντικά συμβάντα προκύπτουν από ενέργειες του χρήστη, χρήση του ποντικιού ή άλλης δεικτικής συσκευής ή χειρισμών σε οθόνη αφής, πληκτρολογήσεις, κλπ. Όμως υπάρχουν πολλές κατηγορίες συμβάντων, ενώ οι διάφορες εκδόσεις του DOM level 2.0, 3.0 κλπ. 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`, κλπ.

2. **Συμβάντα ανεξάρτητα από συσκευή**. Το πιο κλασσικό 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 ακόμη επικόληση κειμένου. Επίσης τα συμβάντα `pointerdown`, `pointermove` αντιστοιχούν στα αντίστοιχα συμβάντα με ποντίκι, 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 αντικειμένου `Window`, καθώς και το συμβάν `DOMContentLoaded` του αντικείμενου `Document`, τα οποία είδαμε στην ενότητα περιγραφής του φορτώματος της ιστοσελίδας στον φυλλομετρητή (1.6). 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, κλπ.

4.2.2 Καταχώρηση χειριστών συμβάντων

Υπάρχουν δύο διαφορετικοί τρόποι για να οριστεί ένας χειριστής συμβάντων, είτε ως τιμή στην αντίστοιχη ιδιότητα του σχετικού αντικειμένου (πχ του στοιχείου του DOM), ή η πιο πρόσφατη προσέγγιση, μέσω της μεθόδου `addEventListener()` του αντίστοιχου αντικειμένου, στην οποία να περνάμε τον χειριστή και τον τύπο του συμβάντος ως ορίσματά της.

4.2.2.1 Χειριστής συμβάντων ως ιδιότητα του αντικειμένου

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

Ένα παράδειγμα είναι, αν επιθυμούμε να φορτώνεται ένα τμήμα του κώδικα όταν ολοκληρωθεί το φόρτωμα της ιστοσελίδας (όταν δηλαδή προκύψει το συμβάν `load` στο αντικείμενο window):

```
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>
```

4.2.2.2 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 αντικειμένου 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.

4.2.2.3 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 εταιρίας. 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 `Courrier`:

```
class Courrier{
    // 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 Courrier(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 χρόνοου, 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.

4.3 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 τη ακολουθιακή (σύγχρονη) κλήση μιας συνάρτησης.

Ας δούμε ένα παράδειγμα. Έστω η συνάρτηση `double()`, η οποία μετά την παρέλευση κάποιου χρόνου (2 δευτερόλεπτα) καλεί την συνάρτηση `f()` η οποία εκτελεί ένα 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 μάς επιστρέψει ο παραπάνω κώδικας είναι `undefined` αφού η `double()` κατά την κλήση της δεν έχει επιστρέψει το αποτέλεσμα 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 ειναι διαθέσιμο.

Έστω ότι αυτή είναι η συνάρτηση με όνομα `callback()`. Ξαναγράφουμε συνεπώς την 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 sec 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 προσδοκόμενο αποτέλεσμα γιατί για παράδειγμα, ο υπολογισμός απέτυχε (πχ ένας εξωτερικός πόρος, όπως ο εξυπηρετητής δεν είναι διαθέσιμος). Για να καλύψουμε και αυτό το πρόβλημα, θα πρέπει να προβλέψουμε στην `double()` να περάσουμε 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 επέστεψε 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, κλπ. 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.

4.4 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()`, η οποία 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 οχι αντίστοιχα.

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

4.4.1 Καταστάσεις του αντικειμένου 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 αντικειμένου, και παίρνει τις τιμές:

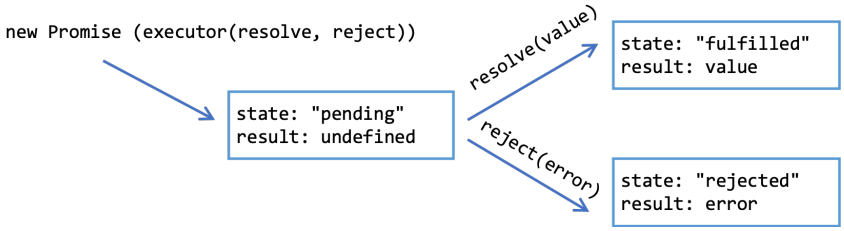
- `"pending"` ένδειξη ότι η ασύγχρονη εργασία είναι σε εξέλιξη
- `"fullfilled"` ένδειξη ότι η ασύγχρονη εργασία έχει ολοκληρωθεί επιτυχώς
- `"rejected"` ένδειξη ότι η ασύγχρονη εργασία απέτυχε να ολοκληρωθεί.

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 `"fullfilled"`, 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, εκτος από τη συνάρτηση `executor()`. Αυτό για να αποφεύγουμε την 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 αντικειμένου 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 ``fullfilled'', ενώ όταν κληθεί η `reject()` θα αλλάξει η κατάσταση σε ``rejected''.



Σχήμα 4.3: 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 αυτήν περνάμε το αντικείμενο `Error` ή το μήνυμα σφάλματος, το οποίο στην περίπτωση αυτή αποτελεί την τιμή της ιδιότητας `result` της υπόσχεσης, όπως φαίνεται στο διάγραμμα.

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 σε σφάλματα κα σύγχρονη εκτέλεση κώδικα της JavaScript.

Ένα παράδειγμα:

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

Στον κώδικα αυτό, 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 ασύγχρονο κώδικα γίνεται με τις μεθόδους καταναλωτές του αντικειμένου Promise, που περιγράφονται στη συνέχεια.

4.4.2 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 αντικειμένου `Promise`.

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

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

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

Ακολουθεί τέλος ένα παράδειγμα που 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 μια 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, και 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.

4.4.2.1 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()` η οποία 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'
```

4.4.3 Παράδειγμα αλυσίδας καταναλωτών υποσχέσεων

Όπως είδαμε, η κατανάλωση μιας υπόσχεσης γίνεται κύρια με τη 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 αντικειμένου, η οποία δεν μπορεί να ξανα-αλλάξει 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 υλοποιηθει 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"));
```

4.4.4 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 **στοίβαςκλήσεων συναρτήσεων**.

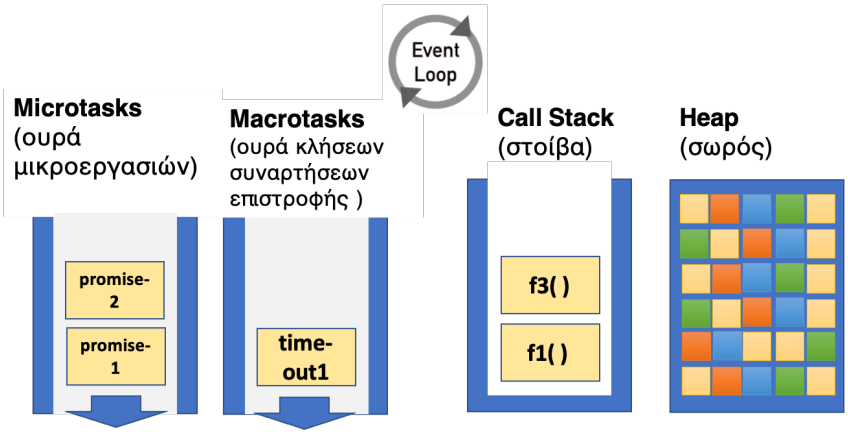
Το ερώτημα που προκύπτει είναι πώς η JavaScript χειρίζεται την ασύγχρονη κλήση συναρτήσεων που γίνεται μέσω του μηχανισμού των Promises.

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.



Σχήμα 4.4: Δομές μνήμης κατά την εκτέλεση κώδικα JavaScipt 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 την 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 κλπ), ενώ εξαντλούνται τα microtasks (promises), ενώ σε διαφορετικούς φυλλομετρητές η σειρά εκτέλεσης των εργασιών αυτών πιθανόν να μεταβάλλεται.

4.5 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 αντικειμένου `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 διεπαφή Fetch, που περιλαμβάνει κλήση της μεθόδου `fetch(url)` η οποία έχει ενσωματωθεί στο αντικείμενο global window, για παράδειγμα: `fetch("/file.json")`, και η οποία 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.

4.5.1 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
```

Στο παράδειγμα στον καταναλωτή `then()` ελέγχουμε την κατάταση της απόκρισης μέσω του γνωρίσματος `status` της απόκρισης που έχουμε λάβει. Αν αυτή είναι 200 (`οκ') τότε καλούμε τη μέθοδο `json()` του αντικειμένου `response`. Αυτή η μέθοδος εκτελεί αντίστοιχη λειτουργία με αυτή της μεθόδου `JSON.parse()` που είδαμε σε προηγούμενη ενότητα (3.4.1), 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 (πχ να εκτυπωθεί στην κονσόλα) από την επόμενη μέθοδο `then()` της αλυσίδας καταναλωτών.

4.5.1.1 Ιδιότητες του αντικειμένου που επιστρέφει η fetch

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

- **headers**. Μέσω του αντικειμένου αυτού έχουμε πρόσβαση στην κεφαλίδα του 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 ένα ακέραιο αριθμό 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. 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** Ιδίοτητα 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, κλπ.

4.5.1.2 Ανάκτηση διαθέσιμων δεδομένων από διεπαφές REST

Η `fetch()` μπορεί να έχει πολλές χρήσεις, μεταξύ των 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)

4.5.2 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, κλπ.) είτε δίνοντας το όνομα της χώρας, είτε δίνοντας ένα κωδικό δύο γραμμάτων της χώρας (πχ ``GR'' για την Ελλάδα), είτε κωδικό τριών γραμμάτων κλπ.

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 την 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 κάθε μια από τις χώρες αυτές θα πρέπει να βρούμε τα στοιχεία της με βάση τον κώδικά της, από αυτά να διαλέξουμε το όνομά της. Άρα να ανακτήσουμε τα στοιχεία κάθε μιας 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 κλήσεις σε διεπαφή δεδομένων.

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

4.5.2.1 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 ένα πίνακα υποσχέσεων τον οποίο να περάσουμε ως όρισμα στην 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 (οκ), κάνουμε συντακτική ανάλυση του αποτελέσματος στον πρώτο καταναλωτή της υπόσχεσης `then()` και εξάγουμε το αποτέλεσμα με τη μορφή αντικειμένου 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 (πχ 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 αυτόν με ασύγχρονες υποσχέσεις ανάκτησης των στοιχείων όλων των χωρών του πίνακα borders.

Τέλος περνάμε τον πίνακα αυτόν 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.

4.6 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, εξαρτώμενα από συσκευές, ανεξάρτητα συσκευής, συμβάντα της διεπαφής, ή συμβάντα που προκύπουν από APIs.

Επίσης έχουμε συζητήσει τρόπους να ορίσουμε ένα 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 στο οποίο έχει οριστεί χειριστής του, η συνάρτηση-χειριστής καλείται.

4.6.1 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.
- `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 τη δεικτική συσκευή έχουν ως ιδιότητες τα `event.clientX` και `event.clientY` με τις συντεταγμένες του σημείου στο παράθυρο. Αν το συμβαν αφορά το πληκτρολόγιο, το συμβάν έχει ως ιδιότητα τον κωδικό του χαρακτήρα που πληκτρολογήθηκε, ως `event.keyCode`, κλπ.

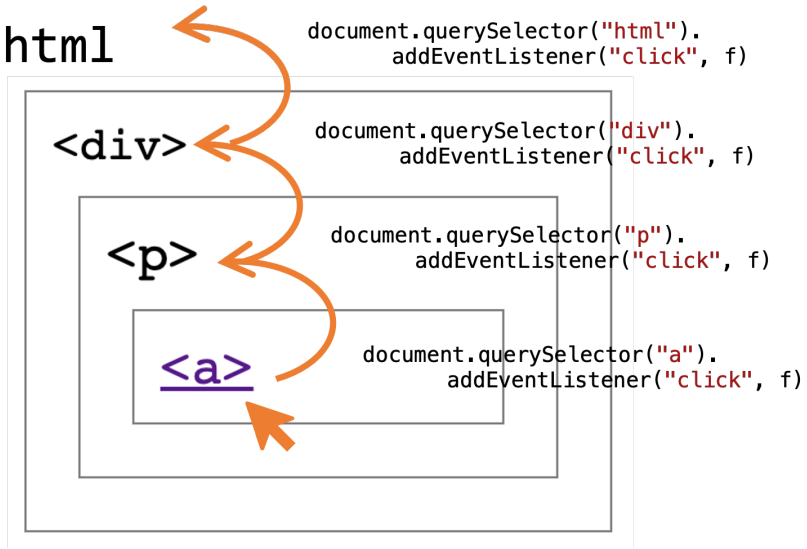
Θα πρέπει να λάβουμε υπόψη μας το γεγονός ότι ένας χειριστής ορίζεται ως τιμή στην ιδιότητα ενός αντικειμένου, συνεπώς ο τελεστής this σε ένα χειριστή, αναφέρεται στο αντικείμενο στο οποίο αυτός έχει οριστεί.

4.6.2 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, κλπ. 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 ένα σύνθετο μηχανισμό διάδοσης συμβάντων (ο **μηχανισμός φυσαλίδας**, event bubbling) που περιγράφεται στη συνέχεια, για να αντιμετωπίσει το θέμα αυτό.

Ας υποθέσουμε το παράδειγμα που φαίνεται στην παρακάτω εικόνα:



Σχήμα 4.5: Παράδειγμα ιεραρχίας στοιχείων και αντίστοιχων χειριστών συμβάντων

Ας υποθέσουμε ότι σε ένα έγγραφο HTML έχουμε την εξής ιεραρχία, που φαίνεται και στο σχήμα 4.5 :

```
<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 μάς 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 τεσσερις 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 ** ** 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 ένα χειριστή για ένα 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 επίσης να σημειώσουμε ότι κάποια συμβάντα δεν διαδίδονται με τον μηχανισμό φυσαλίδας, αυτά περιλαμβάνουν τα συμβάντα ``focus,'' ``blur,'' και ``scroll''.

Θα πρέπει επίσης να αναφερθεί ότι το φαινόμενο διάδοσης ενός συμβάντος είναι ακόμη πιο σύνθετο από τον μηχανισμό φυσαλίδας που ήδη περιγράφτηκε. Η πλήρης περιγραφή του μηχανισμού περιγράφεται στη συνέχεια.

4.6.2.1 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). Αυτή η φάση προηγείται της κλήσης του χειριστή συμβάντος και δεν προκαλεί κλήση των χειριστών που θα ευρεθούν 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 οταν 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 ένα χειριστή του συμβάντος κατά τη φάση της διάδοσης με τον μηχανισμό φυσαλίδας.

4.6.2.2 Αποτροπή προκαθορισμένης συμπεριφοράς

Τα συμβάντα προκαλούν προκαθορισμένη συμπεριφορά σε μια ιστοσελίδα.

Για παράδειγμα 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 κύληση 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 ένα χειριστή του συμβάντος ``submit'' με σκοπό να κάνουμε έλεγχο εγκυρότητας των δεδομένων που έχει εισάγει 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.

4.7 Σύνοψη

Στο κεφάλαιο αυτό, με το οποίο ολοκληρώσαμε την επισκόπηση της JavaScript, είδαμε τους μηχανισμούς που μάς προσφέρει η γλώσσα για ασύγχρονη εκτέλεση τμημάτων του κώδικα, ενώ εξετάσαμε τον μηχανισμό διαχείρισης συμβάντων (events). Όλα αυτά αποτελούν υπόβαθρο για τα επόμενα κεφάλαια, στα οποία θα δούμε πώς με την JavaScript μπορούμε να εξυπηρετήσουμε αιτήματα στην πλευρά του εξυπηρετητή.