ΠΕΡΙΕΧΟΜΕΝΑ:

ΜΑΘΗΜΑ 2.8: Μηχανισμός Events

1. Event Loop

Browser psounis

Event Loop

- Η JS τρέχει σε ένα μόνο νήμα
- Για να εκτελεστούν αποδοτικά τα events, χρησιμοποιείται το event loop για να συγκρατεί τις ενέργειες που πρέπει να κάνει.
- Σύνοψη του event loop:
  - **Call Stack:** Όπως είδαμε είναι ο μηχανισμός για την κλήση συναρτήσεων
  - **Browser:** Συναρτήσεις όπως η setTimeout(), events ή HTTP αιτήματα διαχειρίζονται από τον browser αυτόνομα
  - **Callback Queue:** Όταν τελειώνουν οι ασύγχρονες μέθοδοι, οι callback τους, προστίθενται σε αυτήν την ουρά
  - **Event Loop:** Ελέγχει συνεχώς αν η call stack είναι άδεια. Αν ναι, παίρνει την πρώτη callback από την ουρά και την εκτελεί.

Παράδειγμα 1: event-loop.html

```
console.log("Start");

setTimeout(() => {
  console.log("Timeout callback");
}, 2000);

console.log("End");
```

Τρόπος Λειτουργίας:

- Η 1η console.log μπαίνει στην call stack και εκτελείται.
- Η setTimeout κάνει register στον browser μια callback. Ο browser αναλαμβάνει να παρακολουθήσει τη χρονική διάρκεια 2 sec.
- Η εκτέλεση συνεχίζει αμέσως με την 2η console.log, η οποία μπαίνει στην call stack και εκτελείται.
- Αφού περάσουν τα δύο δευτερόλεπτα, η callback μπαίνει στην callback queue
- Το event loop βλέπει ότι η call stack είναι άδεια, οπότε παίρνει την callback από την callback queue και την θέτει στο call stack
- Η callback εκτελείται και έχουμε την 3η console.log να εκτυπώνει «Timeout callback»

```
Start
End
Timeout callback
```

Σύγχρονες vs Ασύγχρονες Μέθοδοι:

- **Σύγχρονες Μέθοδοι:**
  - Ο κώδικας τους τρέχει άμεσως (π.χ. με την διαδοχική εκτέλεση των εντολών στο script, κάθε επόμενη εντολή μπαίνει στο call stack και εκτελείται αμέσως)
  - Λέμε ότι λειτουργούν με **blocking τρόπο** (δηλαδή το πρόγραμμα δεν μπορεί να συνεχίσει αν δεν ολοκληρωθούν)
- **Ασύγχρονες Μέθοδοι:**
  - Όπως για παράδειγμα η setTimeout, αιτήματα για δεδομένα στο δίκτυο (π.χ. με την fetch())
  - Λέμε ότι λειτουργούν με **non-blocking τρόπο**
  - Με σύγχρονο τρόπο γίνεται register η callback στον browser και αναλαμβάνει να την τρέξει
  - Όταν ολοκληρωθεί η διαχείριση από τον browser, τότε η callback μπαίνει στο callback queue

Events:

- Το registration τους γίνεται με σύγχρονο τρόπο
- Ωστόσο η συμπεριφορά τους είναι ασύγχρονη (Η callback δεν θα εκτελεστεί αν δεν το ενεργοποιήσει ο χρήστης με κάποια ενέργεια)
- Όταν ενεργοποιηθεί, τότε η callback μπαίνει στην callback queue
- (Άρα η συμπεριφορά τους είναι αντίστοιχη με τις ασύγχρονες μεθόδους)

Παράδειγμα 2: events.html

```
console.log('Start');
const button = document.getElementById('myButton’);

button.addEventListener('click', () => {
  console.log('Button clicked!');
});

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

Τρόπος Λειτουργίας:

- Οι δύο πρώτες εντολές μπαίνουν στην call stack και εκτελείται.
- Η addEventListener() κάνει register στον browser μια callback. Ο browser αναλαμβάνει να παρακολουθεί για την ενεργοποίηση του event.
- Η εκτέλεση συνεχίζει αμέσως με την 2η console.log, η οποία μπαίνει στην call stack και εκτελείται.
- Όταν ο χρήστης κάνει κλικ, η callback μπαίνει στην callback queue
- Το event loop βλέπει ότι η call stack είναι άδεια, οπότε παίρνει την callback από την callback queue και την θέτει στο call stack
- Η callback εκτελείται και έχουμε την 3η console.log να εκτυπώνει «Button Clicked»

```
Start
End
Button clicked!
```

ΜΑΘΗΜΑ 2.8: Μηχανισμός Events

2.1. vs Event Spamming: Debouncing

- **Event Spamming ή Event Flooding**: Το πρόβλημα του να προκύπτει ένα event πολύ συχνά

Debouncing:

- Τεχνική για να ελεγχθεί πόσο συχνά μία συνάρτηση εκτελείται
- Πολύ χρήσιμο για events που εκτελούνται πολύ γρήγορα, όπως η πληκτρολόγηση, το resize, το scroll και το κλικ.
- **Η συνάρτηση θα καλείται μόνο μία φορά μετά από μία προκαθορισμένη καθυστέρηση.**
- Υλοποίηση (δεν υποστηρίζεται εγγενώς από την JS, πρέπει να γράψουμε δικό μας κώδικα)
  - Ο χειριστής του event:
    - καθυστερεί την εκτέλεση της διαχειρισης του event κατά ένα χρονικο διάστημα
    - Αν πραγματοποιηθεί ξανά το event, τότε κάνει reset την καθυστέρηση

Στο παράδειγμα:

- 1ο κλικ:
  - Η debounce (αφού το timerId είναι undefined), ορίζει ότι μετά από 1s θα εκτελεστεί η handleClick()
- 2ο κλικ:
  - (Εφόσον έγινε μέσα στο 1s)
  - Το timerId έχει μία συγκεκριμένη τιμή
  - Γίνεται clearTimeout(timerId) και δεν θα εκτελεστεί η handleClick()
- Εφόσον περάσει 1s και δεν γίνει κλικ, θα εκτελεστεί η handleClick().

Παράδειγμα 3: debouncing.html

```js
// The debounce function
function debounce(func, delay) {
  let timerId; // Timer identifier
  return function (...args) {
    // Clear any existing timer
    if (timerId) {
      clearTimeout(timerId);
    }
    // Set a new timer
    timerId = setTimeout(() => {
      func.apply(this, args); // Execute the function after the delay
    }, delay);
  };
}

// The function that handles the click event
function handleClick() {
  console.log('Button clicked!');
}

// Adding event listener with debounce to the button
const clickButton = document.getElementById('clickButton');
clickButton.addEventListener('click', debounce(handleClick, 1000)); // 1 second delay
```

Throttling:

- Εναλλακτική τεχνική για την αποφυγή του event spamming
- **Η συνάρτηση θα καλείται ανά κάποιο χρονικό διάστημα**
- Υλοποίηση (δεν υποστηρίζεται εγγενώς από την JS, πρέπει να γράψουμε δικό μας κώδικα)
  - Ο χειριστής του event:
    - Την πρώτη φορά που λαμβάνει χώρα το event μπαίνει σε κατάσταση throttle.
    - Αν πραγματοποιηθεί ξανά το event, και είναι σε κατάσταση throttle, δεν γίνεται χειρισμός του event
    - Όταν πραγματοποιηθεί ο χειρισμός του event, βγαίνει από την κατάσταση throttle

Στο παράδειγμα:

- 1ο κλικ:
  - Η throttle(): Δεν είναι σε κατάσταση throttle, οπότε γίνεται ο χειρισός του κλικ άμεσα και μπαίνει σε κατάσταση throttle (θα τελειώσει σε 1s)
- 2ο κλικ (σε χρόνο μικρότερο από 1s):
  - Είναι σε κατάσταση throttle, οπότε δεν θα γίνει διαχείριση του event
- Όταν περάσει το 1s, βγαίνει από την κατάσταση throttle.

Παράδειγμα 4: throttling.html

```
// The throttle function
function throttle(func, limit) {
  let inThrottle; // Flag to keep track of throttle state
  return function (...args) {
    if (!inThrottle) {
      func.apply(this, args); // Call the function immediately
      inThrottle = true; // Set throttle flag
      setTimeout(() => {
        inThrottle = false; // Reset the throttle flag after limit time
      }, limit);
    }
  };
}

// The function that handles the button click event
function handleClick() {
  console.log('Button clicked at', new Date().toLocaleTimeString());
}

// Adding event listener with throttle to the button click event
const clickButton = document.getElementById('clickButton');
clickButton.addEventListener('click', throttle(handleClick, 1000)); // 1
second throttle
```

Παράδειγμα: Scrolling Bar

- Χρησιμοποιείται το throttling, ώστε να γίνεται η ενημέρωση της scrolling bar κάθε 200ms.

Παράδειγμα 5: example.html

```html
<div id="progressContainer">
  <div id="progressBar"></div>
</div>
```

```css
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
    height: 3000px; /* To make the page scrollable */
}
#progressContainer {
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 8px;
    background-color: #ccc;
    z-index: 9999;
}
#progressBar {
    width: 0;
    height: 100%;
    background-color: #4caf50;
}
```

```javascript
// The throttle function
function throttle(func, limit) {
    let inThrottle; // Flag to track throttle state
    return function (...args) {
        if (!inThrottle) {
            func.apply(this, args); // Execute the function immediately
            inThrottle = true; // Set throttle flag
            setTimeout(() => {
                inThrottle = false; // Reset the throttle flag after limit time
            }, limit);
        }
    };
}

// The function to update the scroll progress bar
function updateProgressBar() {
    const scrollTop = window.scrollY; // Number of pixels scrolled from the top
    const windowHeight = document.documentElement.scrollHeight -
window.innerHeight;
    const scrollPercentage = (scrollTop / windowHeight) * 100;

    const progressBar = document.getElementById('progressBar');
    progressBar.style.width = `${scrollPercentage}%`; // Set the width of the
progress bar
}

// Adding event listener with throttle to the window scroll event
window.addEventListener('scroll', throttle(updateProgressBar, 200)); // Throttle to
update every 200ms
```