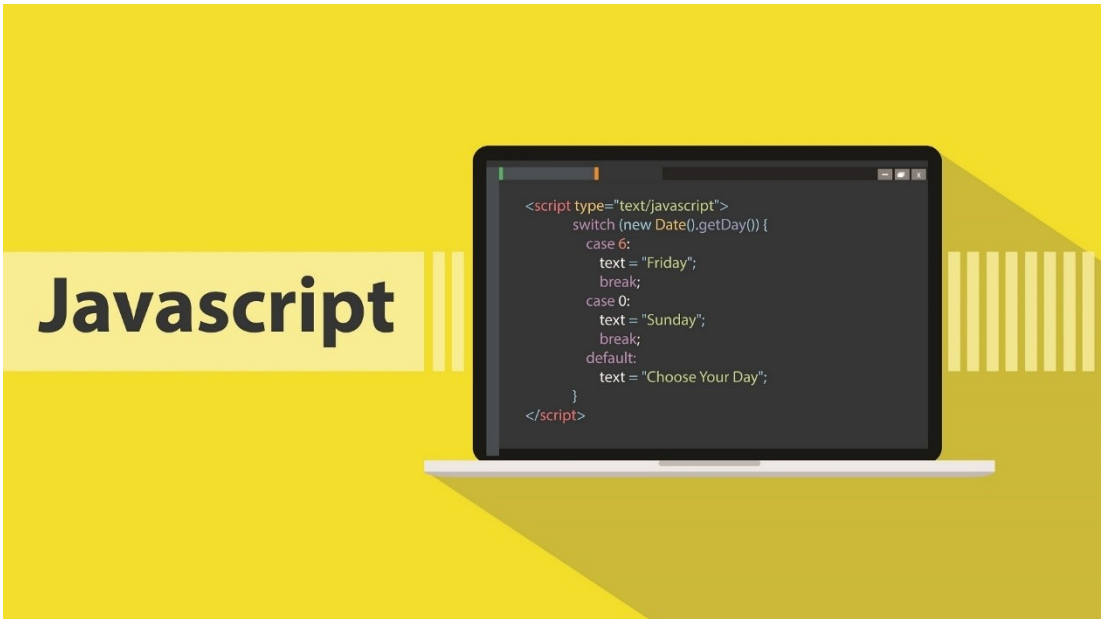
MINISTERUL EDUCAȚIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA

IP COLEGIUL „IULIA HASDEU” DIN CAHUL

CATEDRA TIC

JAVASCRIPT



CAHUL 2022

IP Colegiul „Iulia Hasdeu" Din Cahul

Autor

Pădure Gheorghe, IP Colegiul „Iulia Hasdeu" din Cahul

Aprobat de:

Consiliul metodico-științific al IP Colegiul „Iulia Hasdeu din Cahul", Proces Verbal Nr: 7 din 23 mai 2022.

Recenzenți:

Pîrvan Evghenii, profesor informatică, grad didactic superior, IP Colegiul „Iulia Hasdeu" din Cahul

Pădure Maria profesoară informatică, grad didactic II, IP Colegiul „Iulia Hasdeu" din Cahul

Cuprins

Capitol I. Limbaj de scriptare pentru web

UNITĂȚI DE CONȚINUT

1. Specificațiile limbajului de scriptare.
2. Depănarea programelor:
    a. instrucțiunile alert, console, log;
    b. instrumente folosite la depănarea programelor.
3. Vocabularul și sintaxa limbajului de scriptare.
4. Tipuri de date.
5. Variabile și valori.
6. Operatori:
    c. de atribuire.
    d. aritmetici.
    e. relaționali.
    f. logici.
    g. pentru șiruri.
    h. pentru funcții.
    i. pentru obiecte.
    j. ternari;
    k. de tip typeof și instanceof.
7. Expresii.
8. Domeniul de vizibilitate a variabilelor.
9. Conversiunea tipurilor de date

ABILITĂȚI

- Utilizarea dicționarelor specializate pentru căutarea informației.
- Utilizarea instrumentelor integrate în aplicația client (browser) la depănarea scripturilor.
- Integrarea scriptului în documentele web.
- Respectarea regulilor de sintaxă în scrierea scripturilor.
- Inserarea de comentarii în componența scriptului.
- Definirea tipurilor de date din componența scriptului.
- Descrierea constantelor și variabilelor din componența scriptului.
- Evaluarea expresiilor din componența scriptului.
- Efectuarea conversiunii tipurilor de date.
- Inserarea în script a ferestrelor predefinite de dialog.

Să vedem ce este atât de special la JavaScript, ce putem realiza cu el și ce alte tehnologii lucrează bine cu el.

1. Specificațiile limbajului de scriptare

JavaScript a fost inițial creat pentru a „face paginile web vii”. Programele din acest limbaj se numesc scripturi. Acestea pot fi scrise chiar în HTML-ul unei pagini web și rulate automat pe măsură ce pagina se încarcă. Scripturile sunt furnizate și executate ca text simplu. Nu au nevoie de pregătire specială sau compilare pentru a rula. În acest aspect, JavaScript este foarte diferit de un alt limbaj numit Java.

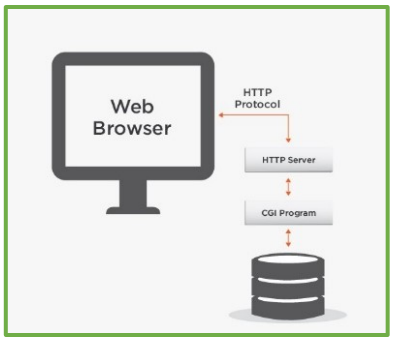
De ce se numește Java Script?

Când a fost creat JavaScript, inițial avea un alt nume: „LiveScript”. Dar Java era foarte popular la acea vreme, așa că s-a decis că poziționarea unui nou limbaj ca „frate mai mic” al Java ar ajuta.

Dar pe măsură ce a evoluat, JavaScript a devenit un limbaj complet independent, cu propria sa specificație numită ECMAScript , iar acum nu are deloc nicio legătură cu Java.

Noțiuni!
Client-side - Partea de client ce se referă la operațiunile efectuate de client într-o relație client-server dintr-o rețea de calculatoare.

Server-side - Latura serverului se referă la operațiunile efectuate de server într-o relație client-server dintr-o rețea de calculatoare.



Limbajele Client Side
Aplicatiile de tip Client Side Script se refera în general la acele programe Web care ruleaza pe client (browser), dupa sau in timpul incarcarii pagini Web de catrea acesta. Limbajele de programare Client Side Script sunt acele limbaje care permit scrierea unor astfel de aplicatii. Aceste limbaje fie sunt integrate in limbajul HTML si suportate direct de catre browser.

JAVASCRIPT

Limbajul JavaScipt (sau ECMAScript) este un limbaj de tip script suportat de browserele Web care poate fi integrat direct intr-o pagina HTML. JavaScript a fost inventat de Brendan Eich în 1995. A fost dezvoltat pentru Netscape 2 și a devenit standardul ECMA-262 în 1997. JavaScript este un limbaj de programare orientat obiect bazat pe conceptul prototipurilor. Este folosit mai ales pentru introducerea unor funcționalități în paginile web, codul JavaScript din aceste pagini fiind rulat de către browser.

ECMASCRIPT

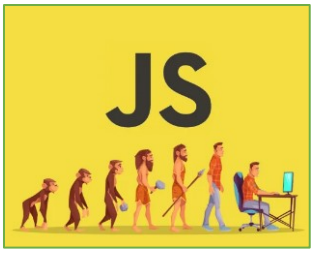
ECMAScript (European Computer Manufacturers Association) este un limbaj de programare de uz general, standardizat de Ecma International conform documentului ECMA-262. Este un standard JavaScript menit să asigure interoperabilitatea paginilor web în diferite browsere web. În anul 1996 ECMA International a elaborat specificații standard numite ECMAScript (ES) pe care toți furnizorii de browsere le-ar putea implementa. Iar Javascript este cea mai cunoscută implementare a ES, în timp ce ActionScript (de la Macromedia / Adobe Systems) și JScript (de la Microsoft) sunt alte implementări ale ES.

TypeScript - este un limbaj de programare open source dezvoltat și menținut de Microsoft. Este un superset sintactic al limbajului JavaScript și asigură un sistem de tipuri opțional. TypeScript este proiectat pentru dezvoltarea de aplicații de mari dimensiuni și se compilează în JavaScript.

ECMAScript (ES):
Până în prezent, ES a publicat 12 versiuni, iar cea mai recentă a fost publicată în anul 2021.

Cele mai noi versiuni ES

1. 8th Edition – ECMAScript 2017
2. 9th Edition – ECMAScript 2018
3. 10th Edition – ECMAScript 2019
4. 11th Edition – ECMAScript 2020
5. 12th Edition – ECMAScript 2021
6. 13th Edition – ECMAScript 2022



Javascript Frameworks

Bibliotecă JavaScript (JavaScript library) - este o bibliotecă de cod JavaScript pre-scris care permite dezvoltarea mai ușoară a aplicațiilor bazate pe JavaScript, în special pentru AJAX și alte tehnologii centrate pe web.

Framework-urile JavaScript sunt un tip de instrument care fac lucrul cu JavaScript mai ușor. Aceste framework-urile permit, de asemenea, programatorului să codeze aplicația ca dispozitiv receptiv. Această reacție este încă un alt motiv pentru care framework-urile JavaScript sunt destul de populare atunci când vine vorba de utilizarea unui limbaj mașină la nivel înalt. Să aruncăm o privire la cele mai bune cadre JS în 2021:

➔ Angular; Vue JS; Next JS; React JS; Ember JS; Node JS; Svelte JS; Gatsby JS; Nuxt JS; Bootstrap etc.

Top 5 cadre web JavaScript după popularitate și utilizare

1. **Vue JS** - este un framework JavaScript de tip front-source model-vizualizare-vizualizare front-end pentru construirea de interfețe utilizator și aplicații cu o singură pagină. A fost creat de Evan You și este întreținut de el și de restul membrilor activi ai echipei de bază.
2. **Angular** - este o platformă de dezvoltare web cu sursă deschisă bazată pe limbajul TypeScript. Proiectul este dezvoltat de Echipa Angular de la Google și de o comunitate de utilizatori individuali și companii. Angular este o rescriere completă, de către aceeași echipă, a frameworkului AngularJS.
3. **React -** este o bibliotecă JavaScript open-source pentru construirea de interfețe de utilizator. Este întreținută de Facebook și de o comunitate de dezvoltatori și companii individuale. React poate fi folosită ca bază pentru dezvoltarea aplicațiilor mobile cu o singură pagină sau mobile.
4. **Node -** este un mediu de execuție JavaScript de tip back-end open-source, cross-platform, care rulează pe motorul V8 și execută cod JavaScript în afara unui browser web.
5. **Ember -** este un cadru web JavaScript open-source, care utilizează un model de servicii componente. Permite dezvoltatorilor să creeze aplicații web scalabile cu o singură pagină, încorporând expresii comune, cele mai bune practici și modele din alte modele de ecosistem cu o singură pagină.

Limbajele Server Side

Ce este Node.js?

Node.js este un mediu runtime JavaScript open source care permite dezvoltatorilor să ruleze cod JavaScript pe server.

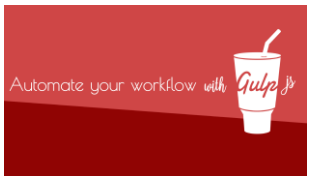
Instrumente de dezvoltare

Visual Studio Code este un mediu de dezvoltare integrat realizat de Microsoft pentru Windows, Linux și macOS. Funcțiile includ suport pentru depanare, evidențierea sintaxei, completarea inteligentă a codului, fragmente, refactorizarea codului și Git încorporat. Utilizatorii pot modifica tema, comenzile rapide de la tastatură, preferințele și pot instala extensii care adaugă funcționalități suplimentare.

Babel este un transcompilator JavaScript gratuit și open-source care este utilizat în principal pentru a converti codul ECMAScript 2015+ într-o versiune de JavaScript compatibilă cu versiunile anterioare care poate fi rulată de motoare JavaScript mai vechi.

Webpack este un pachet de module JavaScript open source. Este creat în principal pentru JavaScript, dar poate transforma active front-end, cum ar fi HTML, CSS și imagini, dacă sunt incluse încărcătoarele corespunzătoare. webpack preia module cu dependențe și generează active statice reprezentând aceste module.

NPM este un manager de pachete pentru limbajul de programare JavaScript menținut de npm, Inc. npm este managerul de pachete implicit pentru mediul de rulare JavaScript Node.js.



Gulp este un set de instrumente JavaScript open-source creat de Eric Schoffstall, utilizat ca un sistem de streaming streaming în dezvoltarea web front-end.

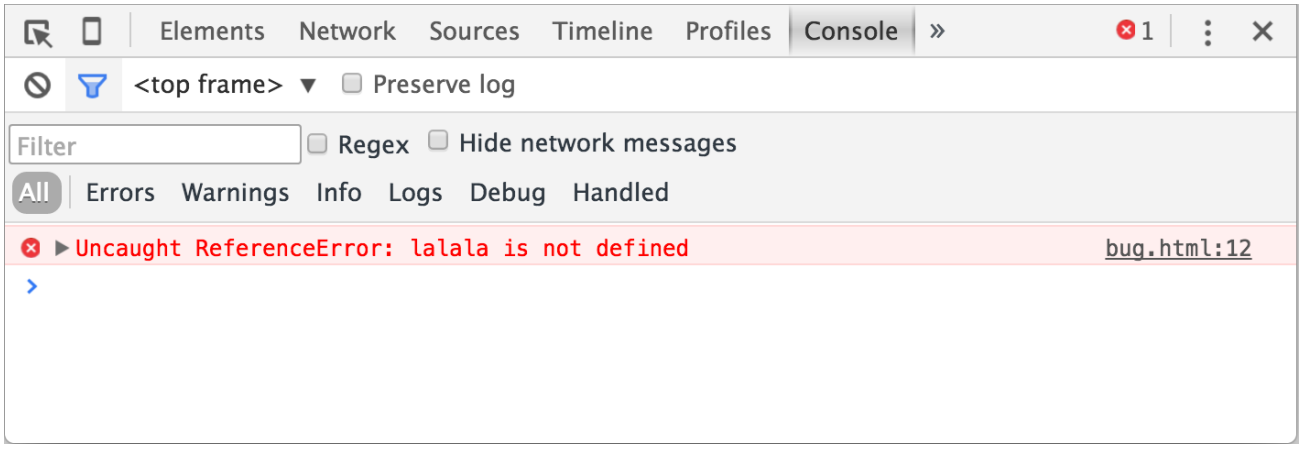
2. Depănarea programelor

Instrumente pentru dezvoltatorie?

Codul este predispus la erori. Probabil că veți face erori. Oh, despre ce vorbesc? Veți face absolut erori, cel puțin dacă sunteți un om, nu un robot. Dar în browser, utilizatorii nu văd erori în mod prestabilit. Deci, dacă ceva nu merge în scenariu, nu vom vedea ce este stricat și nu îl putem remedia. Pentru a vedea erori și pentru a obține o mulțime de alte informații utile despre scripturi, *„Instrumente pentru dezvoltatori”* au fost încorporate în browsere.
Majoritatea dezvoltatorilor se orientează spre Chrome sau Firefox pentru dezvoltare, deoarece aceste browsere dispun de cele mai bune instrumente pentru dezvoltatori. Alte browsere oferă, de asemenea, instrumente pentru dezvoltatori, uneori cu funcții speciale, dar, de obicei, joacă pentru Chrome sau Firefox. Așadar, majoritatea dezvoltatorilor au un browser „preferat” și trec la altele dacă o problemă este specifică browserului.

Google Chrome

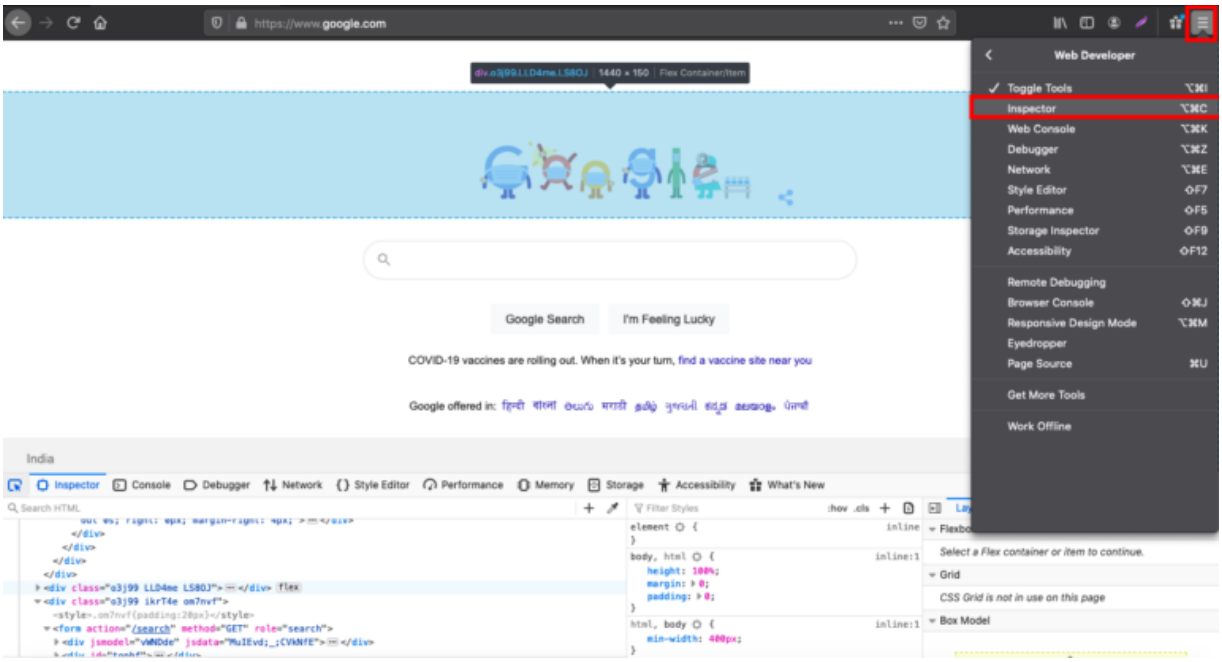
Una dintre cele mai simple modalități de a inspecta un anumit element web din Chrome este să faceți clic dreapta pe acel element și să selectați opțiunea Inspectare. Dacă faceți clic pe opțiunea Inspectare (F12) din meniul cu clic dreapta, se vor deschide direct instrumentele pentru dezvoltatori, inclusiv editorul, consola, sursele și alte instrumente.
Inspectarea elementelor web într-un browser permite dezvoltatorilor, designerilor sau specialiștilor în marketing digital să manipuleze aspectul unei pagini web. Este mai probabil ca dezvoltatorii sau testerii să utilizeze această caracteristică pentru a depana un anumit element, pentru a efectua teste de aspect sau pentru a efectua editare CSS live.
Funcția de inspectare a elementelor este o funcție ușor de utilizat, dar puternică, pentru dezvoltatorii web.
Este extrem de util deoarece permite utilizatorilor să manipuleze aspectul anumitor pagini web. Permite designerilor web să modifice instantaneu proprietățile CSS, cum ar fi fonturile, dimensiunile, culorile etc., pentru a avea o idee despre cum ar arăta o pagină web dacă se fac anumite modificări.



Mozilla Firefox

Mai jos sunt pașii pentru inspectarea elementelor din browserul Firefox:

1. Lansați Firefox și navigați la pagina web dorită care trebuie inspectată. (Să luăm în considerare pagina de pornire Google în acest caz)
2. În colțul din dreapta sus, faceți clic pe trei linii orizontale (deschideți meniul)
3. Din meniul derulant, faceți clic pe Dezvoltator web -> Inspector



Ferestre predefinite de dialog alert, prompt, confirm

Deoarece ca mediu de demonstrație este utilizat browserul, vom analiza funcții de interacțiune cu utilizatorul:

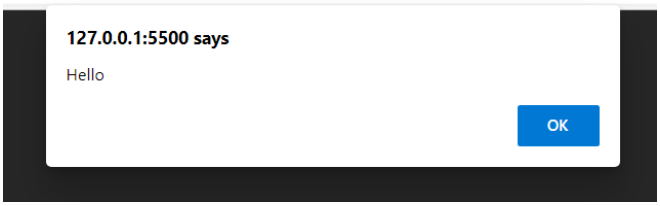
Alert

Funcția **alert** instruiește browserul să afișeze un dialog cu un mesaj opțional și să aștepte până când utilizatorul închide dialogul. Dialogul de alertă trebuie utilizat pentru mesajele care nu necesită niciun răspuns din partea utilizatorului, altul decât confirmarea mesajului.

Casetele de dialog sunt ferestre modale - împiedică utilizatorul să acceseze restul interfeței programului până când caseta de dialog este închisă. Din acest motiv, nu trebuie să folosiți în exces nicio funcție care creează o casetă de dialog (sau o fereastră modală).

De exemplu:

1	`alert("Hello");`
2	`window.alert("Hello");`

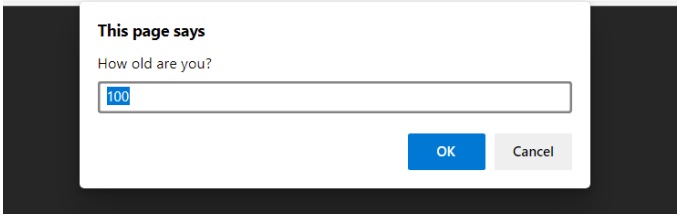


Prompt - Funcția window.prompt() instruiește browserul să afișeze un dialog cu un mesaj opțional care solicită utilizatorului să introducă ceva text și să aștepte până când utilizatorul fie trimite textul, fie anulează dialogul.

În anumite condiții – de exemplu, când utilizatorul schimbă filele – browserul poate să nu afișeze de fapt un dialog sau să nu aștepte ca utilizatorul să trimită text sau să anuleze dialogul.

De exemplu:

1	`Let result = prompt(title, [default]);` **title - Titlul afișat utilizatorului** **default - Valoarea implicită atribuită elementului**
2	`let age = prompt('How old are you?', 100);`



Confirm - window.confirm() indică browserului să afișeze un dialog cu un mesaj opțional și să aștepte până când utilizatorul fie confirmă, fie anulează dialogul.
Valoarea este un boolean care indică dacă a fost selectat OK (true) sau Anulare (false). Dacă un browser ignoră dialogurile din pagină, atunci valoarea returnată este întotdeauna false.

De exemplu:

1	`if (window.confirm("Do you really want to leave?")) {`
	`  window.open("exit.html", "Thanks for Visiting!");`
	`}`

2	`let isBoss = confirm("Are you the boss?");` `alert( isBoss ); // true if OK is pressed`

3. Variabile și constante.

În JavaScript, variabilele sunt utilizate pentru a stoca valori care pot fi modificate în timpul execuției programului. Constante, pe de altă parte, sunt utilizate pentru a stoca valori care nu pot fi modificate după ce au fost definite.

Există trei cuvinte cheie utilizate pentru a declara variabile în JavaScript: var, **let** și **const**.

1. **Var**: Variabilele declarate cu var pot fi declarate și redeclarate oriunde în codul JavaScript. De asemenea, var nu are o zonă de acoperire specifică, aceasta poate fi accesată în orice parte a codului.

De exemplu:

```
var x = 5;
console.log(x); // 5
var x = 10;
console.log(x); // 10
```

2. **Let**: Variabilele declarate cu **let** pot fi declarate și redeclarate doar în interiorul blocului în care au fost definite.

De exemplu:

```
let y = 10;
console.log(y); // 10
let y = 20; // va arunca o eroare, deoarece y a fost deja declarata
```

3. **Const**: Variabilele declarate cu **const** nu pot fi redeclarate sau modificate după ce au fost definite.

De exemplu:

```
const z = 20;
console.log(z); // 20
z = 30; // va arunca o eroare, deoarece z este o constanta
```

Există câteva convenții comune de declarare a variabilelor în JavaScript pentru a face codul mai ușor de citit și întreținut.

1. **Naming convention**: Variabilele ar trebui să aibă nume semnificative și descrivă scopul lor. De exemplu, dacă o variabilă stochează numele unei persoane, ar trebui să-i numiți "name" sau "personName" în loc de "n" sau "x".
2. **CamelCase**: Variabilele ar trebui să fie scrise în CamelCase, în care prima literă a fiecărei cuvinte este scrisă cu majuscule, cu excepția primului cuvânt. De exemplu, "firstName" sau "personAddress"
3. **Constante**: Constantele ar trebui să fie scrise cu litere mari și separate cu underscore, ex: "PI" sau "MAX_AGE"
4. **Prefix**: Variabilele care sunt de tip global ar trebui să aibă un prefix "g", iar cele de tip local un prefix "l". Ex: "gName" sau "lAge"

Consistență: Este important să se mențină o consistență în modul în care sunt declarate variabilele în tot codul, astfel încât să fie ușor de citit și înțeles.

Acestea sunt doar câteva dintre convențiile comune de declarare a variabilelor în JavaScript, dar pot varia în funcție de proiect și echipa de dezvoltare. Cel mai important aspect este ca acestea sa fie clare si usor de inteles de catre ceilalti dezvoltatori care vor lucra cu codul.

Câteva diferențe importante între let și var în JavaScript:

Scop: Variabilele declarate cu var au un scop global sau de funcție, în timp ce variabilele declarate cu let au un scop de bloc. Asta înseamnă că variabilele declarate cu var pot fi accesate oriunde în codul JavaScript, în timp ce variabilele declarate cu let pot fi accesate doar în interiorul blocului în care au fost definite.

Redeclararea: Variabilele declarate cu var pot fi redeclarate în același scop, în timp ce variabilele declarate cu let nu pot fi redeclarate în același scop.

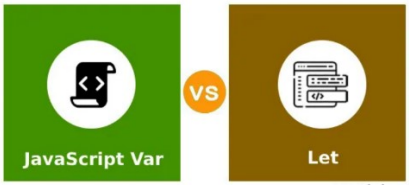
```
var x = 5;
var x = 10; // este permis, x va fi redecalrat cu 10

let y = 10;
let y = 20; // va arunca o eroare, deoarece y a fost deja declarata
```

Hoisting: Variabilele declarate cu var sunt ridicate în partea superioară a scopului în care au fost declarate, în timp ce variabilele declarate cu let nu sunt. Asta înseamnă că variabilele declarate cu var pot fi accesate și utilizate înainte de a fi declarate în cod, în timp ce variabilele declarate cu let nu pot fi accesate sau utilizate până când nu au fost declarate.

```
console.log(x); // undefined
var x = 5;

console.log(y); // va arunca o eroare, deoarece y nu a fost declarata
let y = 10;
```



În general, este recomandat să se evite utilizarea var și se preferă let, deoarece let oferă un control mai bun asupra scopului și redeclarării variabilelor, precum și o mai bună înțelegere a codului.

4. Tipuri de date.

JavaScript suportă diferite tipuri de date, care sunt utilizate pentru a stoca și manipula informațiile în codul JavaScript.

1. **Numere**: Numerele pot fi întregi sau cu virgulă flotantă. Acestea pot fi utilizate pentru a realiza operații aritmetice sau pentru a reprezenta valori precum dimensiunea, timpul sau cantitatea.

```
let num1 = 5; // numar intreg
let num2 = 3.14; // numar cu virgula flotanta
let result = num1 + num2; // 8.14
```

2. **Șiruri de caractere**: Șirurile de caractere sunt utilizate pentru a stoca și afișa text. Acestea pot fi create folosind ghilimele simple sau duble (ex: 'text' sau "text").

```
let string1 = 'Salut';
let string2 = "Mondial";
let string3 = string1 + " " + string2; // "Salut Mondial"
```

3. **Boolean**: Boolean-urile sunt valori care pot fi adevărate sau false. Ele sunt utilizate pentru a verifica dacă o anumită condiție este adevărată sau falsă.

```
let isTrue = true;
let isFalse = false;
if (isTrue) {
    console.log("Afirmatia este adevarata");
} else {
    console.log("Afirmatia este falsa");
}
```

4. **Array-uri**: Array-urile sunt structuri de date care stochează o colecție de elemente. Ele pot conține diferite tipuri de date, cum ar fi numere, șiruri de caractere sau obiecte.

```
let fruits = ['mango', 'banana', 'orange'];
console.log(fruits[0]); // "mango"
fruits.push('apple'); // adauga un element la finalul array-ului
console.log(fruits); // ["mango", "banana", "orange", "apple"]
```

5. **Obiecte**: Obiectele sunt structuri de date care conțin proprietăți și metode. Proprietățile sunt valori asociate unui obiect, iar metodele sunt funcții care pot fi apelate pe un obiect.

```
let car = {
    make: 'Toyota',
    model: 'Camry',
    year: 2020,
    startEngine: function() {
        console.log('Engine is running');
    }
};
console.log(car.make); // "Toyota"
car.startEngine(); // "Engine is running"
```

6. **Undefined**: Undefined este un tip de dată care indică că o variabilă nu are o valoare atribuită.

```
let x;
console.log(x); // undefined
```

7. **Null**: Null este un tip de dată care indică că o variabilă este intenționat "goală" sau "nedefinită".

```
let y = null;
console.log(y); // null
```

Acestea sunt doar câteva exemple de utilizare a diferitelor tipuri de date în JavaScript. Există multe alte modalități de a utiliza și manipula aceste tipuri de date pentru a crea cod funcțional și performant.

QUIZ

1. *Care este diferența dintre variabilele declarate cu var, let și const?*

   a) Variabilele declarate cu var pot fi declarate oriunde în cod, în timp ce variabilele declarate cu let și const pot fi declarate doar în interiorul blocului în care au fost definite.

   b) Variabilele declarate cu var pot fi redeclarate, în timp ce variabilele declarate cu let și const nu pot fi redeclarate.

   c) Variabilele declarate cu const sunt constante, care nu pot fi modificate după ce au fost definite.

   d) Toate afirmațiile de mai sus sunt corecte.

2. *Care dintre următoarele este un exemplu de declarare a unei variabile cu let?*

   a) var x = 5;

   b) const x = 5;

   c) let x = 5;

3. *Care dintre următoarele este un exemplu de utilizare a unui array în JavaScript?*

   a) let fruits = ['mango', 'banana', 'orange'];

   b) let fruits = {'mango', 'banana', 'orange'};

   c) let fruits = ('mango', 'banana', 'orange');

4. *Care dintre următoarele este o convenție de numire a unei constante în JavaScript?*

   a) "PI" sau "MAX_AGE"

   b) "pi" sau "maxAge"

   c) "Pi" sau "Max_age"

5. *Care este un exemplu de utilizare a unui obiect în JavaScript?*

   a) let car = {make: 'Toyota', model: 'Camry', year: 2020};

   b) let car = ('Toyota', 'Camry', 2020);

   c) let car = 'Toyota Camry 2020';

Probleme practice

Sarcină practică 1. *Crearea unui joc ghicitorul numerelor.*

1. Utilizând prompt() cereți utilizatorului să introducă un număr între 1 și 10. Stocați numărul introdus într-o variabilă numită "userNumber"
2. Utilizând Math.random() generați un număr aleatoriu între 1 și 10 și stocați-l într-o variabilă numită "randomNumber"

3. Comparați numărul introdus de utilizator cu numărul generat aleatoriu utilizând un operator de egalitate (==) și stocați rezultatul într-o variabilă numită "isEqual"
4. Utilizând if-statement verificați dacă "isEqual" este adevărat sau fals și afișați un mesaj de felicitare sau un mesaj de încercare din nou utilizând confirm()
5. Utilizați un alt if-statement pentru a verifica dacă numărul introdus de utilizator este mai mare sau mai mic decât numărul generat aleatoriu și afișați un mesaj corespunzător utilizând alert()

***Sarcină practică 2**: Crearea unui program de conversie a valutei.*

1. Creați un fișier JavaScript separat și utilizați o variabilă declarată cu var sau let pentru a stoca o suma introdusa de utilizator prin intermediul metodei prompt().
2. Utilizați o variabila pentru a stoca cursul de schimb introdus de utilizator prin intermediul metodei prompt().
3. Utilizați metoda confirm() pentru a cere utilizatorului sa aleagă daca dorește sa convertească suma introdusa in dolari sau euro.
4. Utilizați o structură de control pentru a verifica dacă utilizatorul a ales sa convertească in dolari sau euro si afișați rezultatul în consolă utilizând metoda console.log().

***Sarcină practică 3**: Crearea unui program de afișare a temperaturii*

1. Creați un fișier JavaScript separat și utilizați o variabilă declarată cu var sau let pentru a stoca temperatura introdusa de utilizator prin intermediul metodei prompt().
2. Utilizați metoda confirm() pentru a cere utilizatorului sa aleagă daca dorește sa afișeze temperatura in grade Celsius sau Fahrenheit.
3. Utilizați o structură de control pentru a verifica dacă utilizatorul a ales sa afișeze in grade Celsius sau Fahrenheit si afișați rezultatul în consolă.

5. Operatorii

JavaScript are mai multe tipuri de operatori, inclusiv operatori aritmetici, operatori de atribuire, operatori de comparație, operatori logici și operatori terți.

Operatori aritmetici (exemplu: +, -, *, /, %): folosiți pentru efectuarea operațiunilor aritmetice, cum ar fi adunarea, scăderea, înmulțirea și împărțirea.

De exemplu:

```
let x = 5;
let y = 2;
let sum = x + y; // sum = 7
let difference = x - y; // difference = 3
let product = x * y; // product = 10
let quotient = x / y; // quotient = 2.5
let remainder = x % y; // remainder = 1
```

Operatori de atribuire (exemplu: =, +=, -=, *=, /=, %=): folosiți pentru a atribui valori variabilelor.

```
let x = 5;
let y = 2;
x += y; // x is now 7
x -= y; // x is now 5
x *= y; // x is now 10
x /= y; // x is now 5
x %= y; // x is now 1
```

Operatori de comparație (exemplu: ==, ===, !=, !==, >, <, >=, <=): folosiți pentru a compara valorile.

```
let x = 5;
let y = 2;
let isEqual = x == y; // isEqual is false
let isStrictlyEqual = x === y; // isStrictlyEqual is false
let isNotEqual = x != y; // isNotEqual is true
let isStrictlyNotEqual = x !== y; // isStrictlyNotEqual is true
let isGreaterThan = x > y; // isGreaterThan is true
let isLessThan = x < y; // isLessThan is false
let isGreaterThanOrEqualTo = x >= y; // isGreaterThanOrEqualTo is true
let isLessThanOrEqualTo = x <= y; // isLessThanOrEqualTo is false
```

Operatori logici (exemplu: &&, ||, !): folosiți pentru a efectua operații logice, cum ar fi și, sau și nu.

```
let x = true;
let y = false;
let and = x && y; // and is false
let or = x || y; // or is true
let not = !x; // not is false
```

Operatori terți (exemplu: typeof, instanceof): folosit pentru a obține informații despre un obiect sau o variabilă.

```
let x = 5;
let y = 'hello';
let typeOf
```

Operatorul ternar

Operatorul ternar (sau operatorul condițional) este un operator specific JavaScript care permite crearea unei expresii condiționale într-o singură linie. Acesta are trei componente: o expresie de test, o expresie care se evaluează dacă expresia de test este adevărată și o expresie care se evaluează dacă expresia de test este falsă.

Sintaxa operatorului ternar este:

```
expresie_test ? expresie_adevărată : expresie_falsă;
```

De exemplu:

```
let x = 5;
let y = 2;
let max = (x > y) ? x : y; // max = 5

let age = 18;
let isAdult = (age >= 18) ? true : false; // isAdult = true

let status = 'online';
let message = (status == 'online') ? 'Welcome, you are online' : 'Sorry, you are
offline'; // message = 'Welcome, you are online'
```

In primul exemplu, operatorul ternar verifica care dintre x si y este mai mare si atribuie valoarea maximei variabilei max.

In al doilea exemplu, operatorul ternar verifica daca varsta e mai mare sau egala cu 18 si atribuie valoarea true sau false variabilei isAdult.

In al treilea exemplu, operatorul ternar verifica statusul utilizatorului si atribuie un mesaj corespunzator variabilei message.

Operatorul ternar este util atunci când doriți să efectuați o operație condițională într-o singură linie, fără a utiliza un bloc if-else.

Exemple suplimentare:

```
let user = "admin";
let access = (user === "admin") ? "full access" : "limited access";
console.log(access); // "full access"

let hour = new Date().getHours();
let timeOfDay = (hour < 12) ? "morning" : (hour < 18) ? "afternoon" : "evening";
console.log(timeOfDay); // "morning" if it's before 12pm, "afternoon" if it's
between 12pm and 6pm, "evening" otherwise.
```

In primul exemplu, operatorul ternar verifica daca user este "admin" si atribuie valoarea "full access" sau "limited access" variabilei access.

In al doilea exemplu, operatorul ternar verifica ora si atribuie valoarea "morning" daca ora este mai mica de 12, "afternoon" daca ora este mai mica de 18 si "evening" in orice alt caz.

In acest fel putem utiliza mai multe operatori ternari în cascadă pentru a crea expresii condiționale complexe, într-o singură linie.

Este important de notat ca operatorul ternar poate fi utilizat oriunde este nevoie de o valoare, nu doar atunci cand se asigneaza o valoare unei variabile.

6. Converisa tipurilor de date

JavaScript este un limbaj de programare dinamic și tipat slab, ceea ce înseamnă că nu este necesar să se declare tipul de date al unei variabile înainte de a o utiliza. De asemenea, JavaScript poate converti automat tipurile de date în unele cazuri, ceea ce poate fi util, dar poate fi și sursa unor erori dacă nu se ține cont de acest lucru.

Există mai multe metode de conversie a tipurilor de date în JavaScript:

- Utilizarea funcției **Number**() pentru a converti un tip de date în număr:

```
let x = "5";
let y = Number(x); // y is now 5 (a number)
```

- Utilizarea funcției **String**() pentru a converti un tip de date în șir:

```
let x = 5;
let y = String(x); // y is now "5" (a string)
```

- Utilizarea funcției **Boolean**() pentru a converti un tip de date în Boolean:

```
let x = "hello";
let y = Boolean(x); // y is true, because x is a non-empty string
let z = "";
let a = Boolean(z); // a is false, because z is an empty string
```

- Utilizarea operatorului + pentru a concatena un șir cu un număr:

```
let x = 5;
let y = "hello";
let z = y + x; // z is "hello5"
```

- Utilizarea operatorului + pentru a aduna doua numere:

```
let x = "5";
let y = 2;
let z = x + y; // z is "52", not 7
```

Este important de menționat că, deși JavaScript poate converti automat tipurile de date în unele cazuri, acest lucru poate duce la erori, în special dacă se presupune că o valoare este de un anumit tip, dar de fapt este de alt tip. De exemplu, în exemplul de mai sus, dacă s-ar fi presupus că x este un număr, rezultatul ar fi fost o eroare, deoarece în realitate x este un șir.

Descoperă!

1. Ce este operatorul ternar și cum se folosește în JavaScript?
2. Care este diferența dintre operatorii == și === în JavaScript?
3. Cum se poate converti un tip de date în Boolean în JavaScript?
4. Care este rolul operatorilor logici (&&, ||, !) în JavaScript?

5. Care este diferența dintre o variabilă declarată cu let și o variabilă declarată cu var în JavaScript?

Probleme practice

Sarcină practică 1. Scrieți un script care cere utilizatorului să introducă numele său folosind **prompt**() și să afișeze un mesaj de bun venit folosind **alert**().

Sarcină practică 2. Scrieți un script care cere utilizatorului să introducă un număr folosind **prompt**() și să verifice dacă acesta este par sau impar folosind operatorul modulo(%) și **alert**() pentru a afișa rezultatul.

Sarcină practică 3. Scrieți un script care cere utilizatorului să confirme dacă dorește să continue folosind **confirm**() și să afișeze un mesaj diferit în funcție de răspunsul utilizatorului.

Sarcină practică 4. Scrieți un script care declară și inițializează două variabile, una cu un număr și cealaltă cu un șir, apoi concatenează cele două valori și afișează rezultatul folosind **console.log**()

Sarcină practică 5. Scrieți un script care declară o variabilă cu o valoare de tip șir și verifică dacă aceasta este goală folosind proprietatea **length** și **console.log**() pentru a afișa rezultatul.

QUIZ

1. Care este sintaxa operatorului ternar?
   a) x > y ? x : y
   b) x < y : x ? y
   c) x = y ? x : y
   d) x == y ? x : y
2. Care este diferența dintre operatorii == și === în JavaScript?
   a) Operatorul == compara doar valoarea, în timp ce operatorul === compara atât valoarea, cât și tipul de date
   b) Operatorul == compara doar tipul de date, în timp ce operatorul === compara doar valoarea
   c) Operatorul == și operatorul === sunt sinonime și au același efect
   d) Operatorul == compara doar valoarea sau tipul de date, in functie de context
3. Cum se poate converti un tip de date în Boolean în JavaScript?
   a) Utilizând funcția Number()
   b) Utilizând funcția String()
   c) Utilizând funcția Boolean()
   d) Utilizând operatorul +
4. Care este rolul operatorilor logici (&&, ||, !) în JavaScript?

a) Operatorul && este utilizat pentru a aduna două numere
b) Operatorul || este utilizat pentru a concatena două șiruri
c) Operatorul ! este utilizat pentru a nega o expresie logică
d) Operatorii &&, ||, ! sunt utilizați pentru a efectua operații logice, cum ar fi și, sau și nu.

5. Care este diferența dintre o variabilă declarată cu let și o variabilă declarată cu var în JavaScript?
   a) Variabilele declarate cu var pot fi accesate din afara blocului în care au fost declarate, în timp ce variabilele declarate cu let nu pot fi accesate din afara blocului
   b) Variabilele declarate cu let pot fi redefinite, în timp ce variabilele declarate cu var nu pot fi redefinite
   c) Variabilele declarate cu var sunt globale, în timp ce variabilele declarate cu let sunt locale
   d) Variabilele declarate cu var au valoarea implicită undefined, în timp ce variabilele declarate cu let nu au o valoare implicită
   e) Ambele tipuri de variabile pot fi accesate din afara blocului în care au fost declarate, dar var este accesibil la nivel global si let la nivelul blocului. Variabilele declarate cu let pot fi redeschise in acelasi bloc unde au fost declarate, in timp ce variabilele declarate cu var nu pot fi redeschise.

Capitol II. Instrucțiunile limbajului de scriptare

UNITĂȚI DE CONȚINUT

1) Instrucțiuni declarative.
2) Instrucțiuni decizionale.
   a) instrucțiunea if;
   b) instrucțiunea else if;
   c) instrucțiunea switch.
3) Instrucțiuni repetitive.
   a) instrucțiunea while;
   b) instrucțiunea do/while;
   c) instrucțiunea for;
   d) Instrucțiunea for/in;
4) Instrucțiuni de salt.
   a) etichete;
   b) instrucțiunea break;
   c) instrucțiunea continue;
   d) instrucțiunea return;
   e) instrucțiunea throw.

ABILITĂȚI

- Aplicarea instrucțiunilor decizionale pentru prelucrarea datelor din pagini web.
- Aplicarea instrucțiunilor repetitive pentru prelucrarea datelor din pagini web.
- A12. Utilizarea instrucțiunii de salt pentru prelucrarea datelor din pagini web.

Javascript este un limbaj de programare interpretat, utilizat pentru a adăuga funcționalități dinamice la paginile web. Unele dintre instrucțiunile comune din Javascript includ:

Variabile: utilizate pentru a stoca valori, cum ar fi numere sau șiruri de caractere.

Condiționale: utilizate pentru a verifica dacă o anumită condiție este adevărată sau falsă și pentru a executa anumite acțiuni în consecință.

Buclă: utilizate pentru a repeta anumite acțiuni de mai multe ori.

Funcții: utilizate pentru a agrupa anumite instrucțiuni care sunt utilizate împreună des și pentru a le putea apela cu ușurință.

Event-uri: utilizate pentru a răspunde la acțiunile utilizatorilor, cum ar fi clicurile sau modificările de text.

DOM(Document Object Model) : utilizat pentru a accesa și modifica elementele din pagină web.

AJAX (Asynchronous JavaScript and XML) : utilizat pentru a face cereri HTTP în fundal, fără a reîncărca întreaga pagină.

Acestea sunt doar câteva dintre instrucțiunile comune din Javascript. Există multe altele, precum și biblioteci și framework-uri care pot fi utilizate pentru a face dezvoltarea mai ușoară și mai eficientă.

1. Instrucțiuni declarative

Variabilele sunt utilizate pentru a stoca valori în Javascript. Ele pot fi declarate folosind cuvântul cheie "var", "let" sau "const" și pot fi utilizate pentru a stoca diferite tipuri de date, cum ar fi numere, șiruri de caractere, obiecte sau chiar funcții.

- "var" este cuvântul cheie utilizat pentru a declara o variabilă în mod tradițional în Javascript. Variabilele declarate cu "var" pot fi redeclarate și redefinite în același context.
- "let" este cuvântul cheie introdus în ECMAScript 6 pentru a declara variabile. Aceste variabile au o scopul de viață în blocul unde sunt declarate și pot fi redeclarate în același context dar nu pot fi redefinite.
- "const" este, de asemenea, un cuvânt cheie introdus în ECMAScript 6 pentru a declara variabile. Aceste variabile sunt constante și nu pot fi redeclarate sau redefinite după ce au fost inițializate.

De exemplu:

```
var nume = "John"; //declararea unei variabile și atribuirea valorii "John"
let age = 25; //declararea unei variabile și atribuirea valorii 25
const pi = 3.14; //declararea unei variabile constante și atribuirea valorii
3.14
```

Este important să se ia în considerare că variabilele declarate fără a fi inițializate au valoarea implicită ***undefined***.

Este recomandat să se evite utilizarea variabilelor globale, deoarece pot duce la conflicte de nume și la probleme de întreținere a codului. In schimb se recomanda utilizarea variabilelor locale sau modulelor.

2. Instrucțiuni decizionale

Instrucțiuni decizionale sunt utilizate pentru a verifica dacă o anumită condiție este adevărată sau falsă și pentru a executa anumite acțiuni în consecință. Cel mai comun operator de comparație utilizat în condiționale este operatorul "==" (egalitate) sau "===", dar există și alți operatori disponibili, cum ar fi "!=" (inegalitate), ">" (mai mare decât), "<" (mai mic decât), ">=" (mai mare sau egal cu), "<=" (mai mic sau egal cu) și "!" (negare).

Instrucțiunile condiționale cele mai comune utilizate în Javascript sunt:

- **"if-else"** - aceasta este structura de control condițională de bază care permite verificarea unei condiții și executarea acțiunilor corespunzătoare în funcție de rezultatul verificării.

- **"if-else if-else"** - aceasta este o extensie a structurii "if-else" care permite verificarea mai multor condiții și executarea acțiunilor corespunzătoare în funcție de care condiție este adevărată.
- **"switch"** - aceasta este o altă structură de control condițională care permite verificarea unei variabile împotriva mai multor valori posibile și executarea acțiunilor corespunzătoare în funcție de valoarea variabilei.
- **"ternary operator"** - operatorul ternar este o metodă scurtă de a scrie o structură "if-else" într-o singură linie. Sintaxa este: ***condition ? value1 : value2***

```
let age = 25;
let adult = (age >= 18) ? true : false;
```

In Javascript exista si alte instrucțiuni condiționale, cum ar fi operatoarele logice și cele de control al fluxului cum ar fi **while, do-while, for, for-in, for-of, break, continue, return, throw, try-catch, debugger etc**.

Instrucțiunea if

Instrucțiunea "if" în Javascript este utilizată pentru a verifica dacă o anumită condiție este adevărată sau falsă. Dacă condiția este adevărată, acțiunile specificate în interiorul blocului "if" sunt executate. Dacă condiția este falsă, acțiunile din interiorul blocului "if" sunt ignorate.

Sintaxa pentru o structură condițională este următoarea:

```
if (condiție) {
    // acțiuni care vor fi executate dacă condiția este adevărată
} else {
    // acțiuni care vor fi executate dacă condiția este falsă
}
```

De exemplu:

```
let age = 25;
if (age >= 18) {
    console.log("Este legal adult");
} else {
    console.log("Nu este legal adult");
}
```

De asemenea, se poate utiliza structura "if-else if-else" pentru a verifica mai multe condiții:

```
let age = 25;
if (age >= 21) {
    console.log("Poate cumpăra alcool");
} else if (age >= 18) {
    console.log("Poate vota");
} else {
    console.log("Nu poate face niciuna dintre aceste activități");
}
```

Este important să se țină cont că condiția trebuie să fie o expresie care va returna un rezultat boolean (true sau false). Și să se evite utilizarea unui număr mare de instrucțiuni "if" în același cod, deoarece poate face codul mai dificil de înțeles și de întreținut. Este recomandat să se utilizați alte structuri de control cum ar fi "if-else if-else" sau "switch" pentru a gestiona mai multe cazuri posibile.

Este de remarcat ca operatorul "==" compara doar valoarea expresiei, in timp ce operatorul "===" compara atât valoarea cat si tipul expresiei.

Exemple de utilizare a instrucțiunii

```
let age = 20;
if (age >= 18) {
    console.log("Este legal adult");
} else {
    console.log("Nu este legal adult");
}
```

In acest exemplu se verifica daca varsta este mai mare sau egala cu 18 și se afișează un mesaj corespunzător.

Expresiile logice în Javascript sunt utilizate pentru a combina mai multe expresii și pentru a returna un singur rezultat boolean (adevărat sau fals). Cele mai comune expresii logice sunt:

- **&& (and) - returnează adevărat dacă ambele expresii sunt adevărate**
- **|| (or) - returnează adevărat dacă cel puțin una dintre expresii este adevărată**
- **! (not) - returnează opusul valorii expresiei**

```
let grade = "A";
if (grade === "A") {
    console.log("Excelent");
} else if (grade === "B") {
    console.log("Bine");
} else if (grade === "C") {
    console.log("Acceptabil");
} else {
    console.log("Sub acceptabil");
}
```

In acest exemplu se verifica nota obținută și se afișează un mesaj corespunzător.

```
let num = 5;
if (num > 0) {
    console.log(num + " este pozitiv");
} else if (num < 0) {
    console.log(num + " este negativ");
} else {
    console.log(num + " este 0");
}
```

In acest exemplu se verifica daca numarul este pozitiv, negativ sau egal cu 0 și se afișează un mesaj corespunzător.

Instrucțiunea switch

Instrucțiunea "switch" în Javascript este utilizată pentru a verifica o variabilă împotriva mai multor valori posibile și pentru a executa acțiuni specifice în funcție de acea valoare. Sintaxa pentru o instrucțiune "switch" este următoarea:

```
switch (variabilă) {
    case valoare1:
        // acțiuni care vor fi executate dacă variabila este egală cu valoare1
        break;
    case valoare2:
        // acțiuni care vor fi executate dacă variabila este egală cu valoare2
        break;
    default:
        // acțiuni care vor fi executate dacă nicio valoare nu se potrivește
}
```

De exemplu:

```
let day = "Monday";
switch (day) {
    case "Monday":
        console.log("Este prima zi a săptămânii");
        break;
    case "Friday":
        console.log("Este a cincea zi a săptămânii");
        break;
    default:
        console.log("Nu este nici o zi specială");
}
```

In acest exemplu se verifica ziua din saptamana si se afiseaza un mesaj corespunzator:

```
let grade = "A";
switch (grade) {
    case "A":
        console.log("Excelent");
        break;
    case "B":
        console.log("Bine");
        break;
    case "C":
        console.log("Acceptabil");
        break;
    default:
        console.log("Sub acceptabil");
}
```

In acest exemplu se verifica nota obținută și se afișează un mesaj corespunzător.

Este important să se adauge "break" la finalul fiecărei case pentru a opri executarea ulterioară a codului. Dacă nu se adaugă "break", codul din toate cazurile care se potrivesc cu valoarea variabilei va fi executat.

De asemenea, este important să se țină cont că valorile comparate în instrucțiunea "switch" trebuie să fie de același tip cu variabila. Dacă se compara o variabilă de tip string cu valori de tip number sau boolean, acest lucru va duce la o eroare.

In general, instrucțiunea "switch" este mai clară și mai ușor de înțeles decât o cascadă de instrucțiuni "if-else" când se compară o variabilă împotriva unui număr mare de valori posibile. Este recomandat să se evite utilizarea unui număr mare de cazuri în aceeași instrucțiune "switch" pentru a evita confuzia și dificultatea în întreținerea codului.

3. Instrucțiuni repetitive

Instrucțiunile repetitive sau buclele în Javascript sunt utilizate pentru a repeta un anumit bloc de cod de mai multe ori. Cele mai comune instrucțiuni repetitive utilizate în Javascript sunt:

- "**for**" - Buclele "for" sunt utilizate atunci când se știe înainte de începerea buclării câte iterări trebuie efectuate. Aceasta este o buclă care se execută pentru un număr specific de ori. Sintaxa

este:

```
for (initializare; condiție; incrementare/decrementare) {
    // codul care se va repeta
}
```

"*initializare*" este o declarare de variabile sau un set de instrucțiuni care se execută o singură dată, la începutul buclei.

"*condiție*" este o expresie care se evaluează înainte de fiecare iterație. Dacă aceasta este adevărată, codul din interiorul buclei se va executa, dacă este fals, bucla se va opri.

"*incrementare/decrementare*" este un set de instrucțiuni care se execută după fiecare iterație.

De exemplu:

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

Această buclă va afișa numerele de la 0 la 9, prin iterarea variabilei "i" de la 0 la 9.

- Buclele "**while**" sunt utilizate atunci când nu se știe înainte de începerea buclării câte iterări trebuie efectuate, dar se știe când trebuie să se oprească. Aceasta este o buclă care se execută atât timp cât o anumită condiție este adevărată. Sintaxa este:

```
while (condiție) {
    // codul care se va repeta
}
```

Condiția este o expresie care se evaluează înainte de fiecare iterație. Dacă aceasta este adevărată, codul din interiorul buclei se va executa, dacă este fals, bucla se va opri.

De exemplu:

```
let i = 0;
while (i < 10) {
    console.log(i);
    i++;
}
```

Această buclă va afișa numerele de la 0 la 9.

- "**do-while**" - aceasta este o buclă similară cu "while", cu excepția faptului că codul din interiorul buclei se va executa cel puțin o dată, indiferent dacă condiția este adevărată sau nu. Sintaxa este:

```
do {
    // codul care se va repeta
} while (condiție);
```

De exemplu:

```
let i = 0;
do {
    console.log(i);
    i++;
} while (i < 10);
```

Această buclă va afișa numerele de la 0 la 9.

Este important să se evite utilizarea unui număr mare de bucle în același cod sau buclarea nesfârșita, deoarece poate face codul mai dificil de înțeles și de întreținut si poate cauza probleme de performanță.

QUIZ

1. *Care dintre următoarele este o instrucțiune conditională în Javascript?*
   a) while
   b) switch
   c) repeat
   d) if
2. *Care dintre următoarele este o instrucțiune conditională în Javascript?*
   a) @
   b) &&
   c) #
   d) $
3. *Care este scopul instrucțiunii "for" în Javascript?*
   a) pentru a repeta un bloc de cod pentru un număr specific de ori
   b) pentru a verifica o variabilă împotriva mai multor valori posibile
   c) pentru a repeta un bloc de cod atât timp cât o anumită condiție este adevărată

d) pentru a repeta un bloc de cod cel puțin o dată, indiferent dacă o anumită condiție este adevărată sau nu

4. *Care este scopul instrucțiunii "break" utilizată în interiorul unei instrucțiuni "switch"?*

   a) pentru a opri executarea ulterioară a codului

   b) pentru a continua executarea ulterioară a codului

   c) pentru a returna un rezultat boolean

   d) pentru a afișa un mesaj

5. *Care dintre următoarele este o recomandare pentru utilizarea buclelor în Javascript?*

a) utilizarea unui număr mare de bucle în același cod

b) utilizarea buclării infinite

c) evitarea utilizării unui număr mare de bucle în același cod pentru a evita confuzia și dificultatea în întreținerea codul

4. Instrucțiuni de salt

Instrucțiunile de salt din Javascript permit programatorului să controleze fluxul de execuție al programului, permițând sare peste anumite porțiuni de cod sau să revină la anumite puncte din cod.

- Instrucțiunea "**break**" permite programatorului să iasă din bucla sau din instrucțiunea switch în care se află și să continue execuția programului după acea buclă sau instrucțiune.
- Instrucțiunea "**continue**" permite programatorului să sari peste o iterație a unei bucle și să continue cu următoarea iterație.
- Instrucțiunea "**return**" permite programatorului să iasă din funcția curentă și să returneze o valoare către apelant sau să continue execuția programului după acea funcție.

De exemplu:

```
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break;
  }
  console.log(i);
}
```

Acest cod va afișa numerele de la 0 la 4, pentru că instrucțiunea "break" oprește execuția buclei atunci când i = 5.

Este important să utilizați aceste instrucțiuni cu atenție pentru a evita bucla infinită sau alte probleme de execuție în program.

De exemplu:

```
for (let i = 0; i < 10; i++) {
  if (i % 2 !== 0) {
    continue;
  }
  console.log(i);
}
```

Acest cod va afișa numerele pare de la 0 la 8, pentru că instrucțiunea "continue" sari peste iterația curentă atunci când i este impar și continua cu următoarea iterație.

De exemplu:

```
function findMax(numbers) {
  let max = numbers[0];
  for (let i = 1; i < numbers.length; i++) {
    if (numbers[i] > max) {
      max = numbers[i];
    }
  }
  return max;
}

let numbers = [3, 5, 7, 2, 8, 10];
let maxNum = findMax(numbers);
console.log(maxNum); // afisează 10
```

Această funcție parcurge un array de numere și returnează cel mai mare număr din array. Instrucțiunea "return" returnează valoarea maxima către apelantul funcției.

Probleme practice

Sarcină practică 1. Creați o variabilă numită "num" care să conțină numărul 1.

Sarcină practică 2. Utilizând o buclă "for", iterați de la num la 10 și afișați fiecare număr pe ecran.

Sarcină practică 3. Utilizând o buclă "while", iterați de la num la 10 și afișați doar numerele pare.

Sarcină practică 4. Utilizând o instrucțiune "if-else", verificați fiecare număr și afișați un mesaj diferit pentru numerele mai mari sau mai mici decât 5.

Sarcină practică 5. Adăugați o instrucțiune "switch" care să verifice ultimul număr din interva și să afișeze un mesaj diferit în funcție de valoare (de exemplu, "Ultimul număr este 10").

Sarcină practică 6. Adăugați o opțiune în instrucțiunea "switch" care să afișeze un mesaj pentru cazul în care numărul nu se potrivește cu niciunul dintre cazurile definite.

Sarcină practică 7.

1. Creați un program care să genereze un număr aleatoriu între 1 și 1000 și să returneze cel mai apropiat număr prim mai mare decât acesta.
2. Utilizând o buclă "while" iterați prin intervalul de la numărul generat la 1000 și verificați dacă fiecare număr este prim sau nu.
3. Utilizând o instrucțiune "if" verificați dacă numărul este prim și salvați-l într-o variabilă dacă este cel mai apropiat număr prim mai mare decât numărul generat.
4. Utilizând un "break" opriți bucla atunci când cel mai apropiat număr prim mai mare decât numărul generat este găsit
5. Returnați cel mai apropiat număr prim mai mare decât numărul generat sau un mesaj corespunzător dacă nu a fost găsit unul.
6. Adăugați o instrucțiune "return" în interiorul buclei "while" care să returneze cel mai apropiat număr prim mai mare decât numărul generat dacă diferența dintre acesta și numărul generat este mai mică decât o anumită valoare prestabilită (de exemplu, 5). Aceasta va permite oprirea buclei și returnarea rezultatului când este găsit un număr suficient de apropiat.
7. Utilizând o instrucțiune "switch" sau "if-else" afișați un mesaj corespunzător în funcție de rezultatul returnat (de ex: "Cel mai apropiat număr prim mai mare decât numărul generat este X" sau "Nu a fost găsit un număr prim suficient de apropiat.")
8. Testați programul cu diferite numere generate și verificați dacă returnează rezultatele așteptate.

Probleme individuale:

1. Creați un program care să returneze toți factorii primi ai unui număr introdus de utilizator.
2. Creați un program care să determine dacă un număr introdus de utilizator este prim sau nu.
3. Creați un program care să returneze cel mai mare divizor comun al a două numere introduse de utilizator.

4. Creați un program care să returneze cel mai mic multiplu comun al a două numere introduse de utilizator.
5. Creați un program care să primească ca input un șir de caractere și să returneze numărul de apariții ale fiecărui caracter în șirul introdus utilizand o structura de tip switch sau if-else. Utilizand o bucla for sau while, iterati prin șirul de caractere și verificați fiecare caracter cu ajutorul unei instrucțiuni switch sau if-else și incrementați o variabilă corespunzătoare pentru fiecare caracter.
6. Creați un program care să returneze toate numerele prime dintr-un interval dat de numere, folosind o structura de tip while sau for, și verificarea fiecarui număr daca este prim sau nu, folosind o metodă de verificare a numerelor prime. Utilizați o instrucțiune "break" pentru a opri bucla atunci când intervalul specificat a fost parcurs complet.
7. Creați un program care să returneze suma și numărul de elemente dintr-un array de numere, utilizand o structura de tip for sau while, și o instrucțiune if pentru a verifica dacă elementul este par sau impar.

QUIZ

1. *Care este scopul instrucțiunii "break" în Javascript?*

   a) să iasă din bucla sau din instrucțiunea switch în care se află și să continue execuția programului după acea buclă sau instrucțiune

   b) să continue cu următoarea iterație

   c) să iasă din funcția curentă și să returneze o valoare către apelant
2. *Care este scopul instrucțiunii "continue" în Javascript?*

   a) să iasă din bucla sau din instrucțiunea switch în care se află și să continue execuția programului după acea buclă sau instrucțiune

   b) să sari peste o iterație a unei bucle și să continue cu următoarea iterație

   c) să iasă din funcția curentă și să returneze o valoare către apelant
3. *Care este scopul instrucțiunii "return" în Javascript?*

   a) să iasă din bucla sau din instrucțiunea switch în care se află și să continue execuția programului după acea buclă sau instrucțiune

   b) să sari peste o iterație a unei bucle și să continue cu următoarea iterație

   c) să iasă din funcția curentă și să returneze o valoare către apelant
4. *Cum poți verifica dacă un număr este prim în Javascript?*

a) Prin utilizarea unei bucle care să parcurgă numerele de la 2 la numărul introdus și să verifice dacă numărul introdus are vreun divizor în afara de 1 și de el însuși

b) Prin utilizarea unei funcții predefinite numită "isPrime"

c) Prin utilizarea metodei "Math.prime"

5. *Care este diferența dintre o instrucțiune "if-else" și o instrucțiune "switch"?*

   a) Instrucțiunea "if-else" este utilizată pentru a verifica o singură condiție, în timp ce instrucțiunea "switch" este utilizată pentru a verifica mai multe condiții

   b) Instrucțiunea "if-else" este mai rapidă decât instrucțiunea "switch"

   c) Instrucțiunea "if-else" poate fi utilizată doar pentru verificarea valorilor numerice, în timp ce instrucțiunea "switch" poate fi utilizată pentru verificarea valorilor numerice sau a valorilor de tip string sau boolean.

6. *Care este diferența dintre o buclă "for" și o buclă "while" în Javascript?*

   a) Bucla "for" este utilizată pentru a itera prin un număr fix de iterații, în timp ce bucla "while" este utilizată pentru a itera până când o anumită condiție este îndeplinită

   b) Bucla "for" are un număr fix de parametri care trebuie specificați, în timp ce bucla "while" nu are nevoie de niciun parametru

   c) Bucla "for" poate fi utilizată doar pentru iterații cu numere, în timp ce bucla "while" poate fi utilizată pentru iterații cu orice tip de date.

7. *Ce este o expresie logică în Javascript?*

   a) O expresie logică este o afirmație care poate fi evaluată ca adevărată sau falsă

   b) O expresie logică este o funcție care returnează un rezultat

   c) O expresie logică este un tip de date

8. *Care este scopul instrucțiunii "switch" în Javascript?*

   a) Să verifice o singură condiție

   b) Să verifice mai multe condiții și să execute o acțiune specifică în funcție de care condiție este îndeplinită

   c) Să returneze o valoare

9. *Care este scopul instrucțiunii "return" în Javascript?*

   a) Să iasă din bucla sau din instrucțiunea switch în care se află și să continue execuția programului după acea buclă sau instrucțiune

   b) Să returneze o valoare către apelant

c) Să afișeze un mesaj

10. *Cum poți utiliza o instrucțiune "if-else" pentru a verifica dacă un număr este mai mare sau mai mic decât o valoare prestabilită?*

    a) Utilizând comparatorii "<" sau ">" și specificând valoarea prestabilită în condiția if sau else

    b) Utilizând funcția predefinită "compareNumber"

    c) Utilizând operatorii matematici "+" sau "-" și specificând valoarea prestabilită în condiția if sau else.

Capitol III. Funcții de scriptare

UNITĂȚI DE CONȚINUT

1) Declararea funcției.
2) Domeniul de vizibilitatea a variabilelor locale și globale.
3) Apelul funcției.
4) Parametrii funcției.

ABILITĂȚI

- Aplicarea funcțiilor predefinite.
- Definirea funcțiilor proprii.

O **funcție** în **JavaScript** este un bloc de cod care poate fi invocat în orice moment și poate returna un rezultat. Ele sunt definite cu cuvântul cheie "**function**" și pot accepta argumente ca parametri. Funcțiile sunt utilizate pentru a împacheta logica și a permite reutilizarea codului în aplicația ta.

În JavaScript, funcțiile sunt elemente importante ale limbajului și sunt utilizate pentru a împacheta logica de aplicație și pentru a permite reutilizarea codului. Unele detalii suplimentare despre funcțiile din JavaScript sunt:

1) **Sintaxa**: Funcțiile sunt definite cu cuvântul cheie "**function**", urmat de un nume pentru funcție și o listă de parametri între paranteze.
2) **Argumente**: Funcțiile pot accepta argumente ca parametri și pot fi apelate cu valori specifice pentru acestea.
3) **Returnare**: Funcțiile pot returna valori prin intermediul cuvântului cheie "**return**".
4) **Invocare**: Funcțiile pot fi apelate în orice moment prin intermediul numelui lor și a parantezelor.
5) **Scop**: Funcțiile au un scop propriu în care variabilele definite în interiorul lor sunt accesibile numai în interiorul funcției.

Funcțiile sunt utile pentru a împacheta logica complexă și pentru a permite reutilizarea codului în aplicația ta. Ele fac codul mai ușor de înțeles și de menținut.

În JavaScript există mai multe tipuri de funcții, inclusiv:

1) Funcții **declarative**: Sunt funcții care au un nume și pot fi invocate prin intermediul acestuia.

2) Funcții **anonyme**: Sunt funcții fără nume care sunt atribuite unei variabile sau sunt trimise ca argumente.
3) Funcții de **recursive**: Sunt funcții care se pot invoca pe ele însele.
4) Funcții **asociative**(***Arrow***): Sunt funcții scurte și concise care folosesc o sintaxă mai scurtă decât funcțiile obișnuite.
5) Funcții **constructor**: Sunt funcții care sunt folosite pentru a crea noi obiecte.
6) Funcții **expresii**: Funcțiile expresie sunt definite prin intermediul unei expresii anonime asignate unei variabile sau prin intermediul unei proprietăți a unui obiect.

1. Declararea funcției

Există două modalități de a declara o funcție în JavaScript: funcții declaration și funcții expresie.

Funcția declarativă:

```
function greet(name) {
  console.log("Hello, " + name);
}

greet("John"); // Output: Hello, John
```

Funcția expresie

```
const greet = function(name) {
  console.log("Hello, " + name);
};

greet("John"); // Output: Hello, John
```

În ambele exemple, funcția poate fi invocată prin intermediul numelui său și poate primi orice număr de argumente necesare.

Funcția arrow (sau funcția cu săgeată) este o formă scurtă de a defini funcții anonime în JavaScript. Este definită prin intermediul operatorului =>.

```
const greet = (name) => {
  console.log("Hello, " + name);
};

greet("John"); // Output: Hello, John
```

Funcțiile arrow au câteva diferențe importante față de funcțiile traditionale, cum ar fi:

- Nu au acces la **this** și **arguments** în mod implicit, ci le moștenesc din contextul în care sunt definite.
- Nu au acces la **new.target**.
- Nu pot fi utilizate ca constructor.

Acestea sunt câteva dintre avantajele funcțiilor arrow:

- Sunt mai concise decât funcțiile traditionale.
- Nu au un **this** propriu, ci moștenesc **this** din contextul în care sunt definite.
- Sunt utile în funcții callback sau în codul asincron.

2. Domeniul de vizibilitatea a variabilelor locale și globale

În JavaScript, există două tipuri de variabile: variabile globale și variabile locale.

Variabilele globale sunt definite în afara oricărei funcții și pot fi accesate de oriunde în cod, indiferent de contextul în care sunt invocate.

Variabilele locale sunt definite în interiorul unei funcții și sunt accesibile numai în interiorul acestei funcții. Domeniul de vizibilitate al unei variabile locale este limitat la interiorul funcției în care este definită.

De exemplu:

```
let globalVariable = "Acesta este un global";

function localScope() {
  let localVariable = "Acesta este un local";
  console.log(globalVariable); // Output: Acesta este un global
  console.log(localVariable); // Output: Acesta este un local
}

console.log(globalVariable); // Output: Acesta este un global
console.log(localVariable); // Output: ReferenceError: localVariable is not define
```

În acest exemplu, variabila **globalVariable** poate fi accesată oriunde în cod, în timp ce variabila **localVariable** poate fi accesată numai în interiorul funcției **localScope**. Dacă încercați să accesați variabila **localVariable** în afara funcției, veți obține o eroare *"ReferenceError: localVariable is not defined"*.

3. Apelul funcției

Apelul funcției se referă la procesul prin care se invocă sau se execute o funcție în cod. În JavaScript, pentru a apela o funcție, trebuie să specificați numele acesteia, urmat de paranteze care conțin eventualii parametri.

De exemplu:

```
function greeting(name) {
  console.log("Hello, " + name);
}

greeting("John"); // Output: Hello, John
```

În acest exemplu, funcția **greeting** este apelată cu parametrul "John". Aceasta va afișa mesajul "Hello, John" în consolă.

Este important de menționat că o funcție poate fi apelată oricând și de oriunde în cod, atâta timp cât este definită înaintea apelului său.

4. Parametrii funcției

Parametrii unei funcții sunt variabile care sunt declarate în definiția funcției și care primesc valorile când funcția este invocată. Parametrii pot fi folosiți pentru a primi valori sau date de la exterior și pentru a le utiliza în interiorul funcției.

De exemplu:

```
function greet(name) {
  console.log("Hello, " + name);
}

greet("John"); // Output: Hello, John
```

În acest exemplu, **name** este un parametru al funcției **greet**. Când funcția este invocată cu valoarea "**John**", parametrul **name** primește această valoare și o utilizează în interiorul funcției pentru a afișa mesajul *"Hello, John"*.

Parametrii opționali sunt parametri ai unei funcții care nu sunt necesari pentru a invoca funcția și care au valori implicite definite în cazul în care nu sunt furnizați. Dacă un parametru opțional nu este furnizat atunci valoarea implicită va fi utilizată.

De exemplu:

```
function greet(name, greeting = "Hello") {
  console.log(greeting + ", " + name);
}

greet("John"); // Output: Hello, John
```

În acest exemplu, parametrul **greeting** este un parametru opțional care are o valoare implicită de *"Hello"*. Dacă nu este furnizat, această valoare implicită va fi utilizată.

5. Funcții expresii

O funcție expresie este o expresie care poate fi evaluată ca o funcție în JavaScript. Funcțiile expresie sunt definite prin intermediul unei expresii anonime asignate unei variabile sau prin intermediul unei proprietăți a unui obiect. Acestea pot fi numite sau invocate la fel ca și funcțiile declaration.

De exemplu:

```
const greet = function(name) {
  console.log("Hello, " + name);
};

greet("John"); // Output: Hello, John
```

În acest exemplu, funcția **greet** este definită ca o funcție **expresie** și este asignată unei variabile cu același nume. Poate fi invocată ca orice altă funcție.

Funcțiile expresii sunt funcții care sunt definite în timpul execuției programului, în contrast cu funcțiile declarate, care sunt definite în mod sintactic înaintea execuției programului.

În JavaScript, funcțiile expresii pot fi definite prin atribuirea unei expresii funcției la o variabilă sau prin pasarea unei funcții ca argument către o altă funcție.

De exemplu:

```
let sum = function(a, b) {
  return a + b;
};

console.log(sum(2, 3)); // Output: 5

let add = (a, b) => a + b;

console.log(add(2, 3)); // Output: 5
```

În acest exemplu, funcția **sum** este definită prin atribuirea unei expresii funcției la variabila **sum**. Aceasta poate fi apelată prin apelul variabilei **sum** și va returna suma parametrilor a și b.

Funcția **add** este definită prin utilizarea unei funcții "**arrow**". Aceasta este un tip special de funcție expresie și oferă un mod scurt de a scrie funcții anonime.

Este important de menționat că, dacă o funcție expresie nu este atribuită unei variabile, aceasta nu poate fi apelată. De asemenea, numele funcției expresii nu este disponibil în afara contextului în care este definită.

6. Funcții asociative (Arrow)

Funcțiile Arrow (sau funcțiile cu sintaxa "fat arrow") sunt o formă compactă de funcții în JavaScript, introdusă în ECMAScript 6. Ele au sintaxa **(parametri) => expresie**. Funcțiile arrow pot fi folosite în locul funcțiilor tradiționale și au următoarele caracteristici:

- Nu au **this** propriu - **this** din interiorul unei funcții **arrow** este legat de **this** din contextul în care funcția a fost declarată.
- Nu au **arguments** propriu - în funcțiile arrow, arguments nu este definit.
- Nu au proprietatea **prototype** - funcțiile arrow nu pot fi utilizate ca constructori (nu pot fi apelate cu **new**).

De exemplu:

```
// Funcție tradițională
let add = function (a, b) {
  return a + b;
};

// Funcție arrow
let add = (a, b) => {
  return a + b;
};

// Funcție arrow scurtă
let add = (a, b) => a + b;
```

În acest exemplu, toate cele trei variante de funcții sunt echivalente și implementează aceeași logică de adăugare a două numere. Sintaxa funcției arrow scurte este cea mai compactă și se poate utiliza atunci când funcția are o singură expresie.

Este important de menționat că funcțiile arrow pot fi utilizate în locul funcțiilor tradiționale în majoritatea situațiilor, cu excepția cazurilor în care este necesară accesarea proprietăților proprii ale funcției sau utilizarea acestora ca constructori.

7. Callback functions

Callback-urile sunt funcții care sunt pase ca argumente în alte funcții. Acestea sunt apelate la un moment ulterior în timp, atunci când un anumit eveniment sau condiție se îndeplinește. Callback-urile sunt utilizate pentru a implementa funcționalități asincrone în JavaScript, cum ar fi citirea datelor de la server sau executarea unei acțiuni după încărcarea paginii.

De exemplu, următorul cod declară o funcție **loadData** care primește un **callback** ca argument și îl apelă după ce datele au fost citite de la server:

De exemplu:

```
function loadData(callback) {
  // Citirea datelor de la server
  // ...

  // Apelarea callback-ului
  callback();
}

// Apelul funcției loadData cu un callback
loadData(function() {
  console.log('Datele au fost încărcate');
});
```

În acest exemplu, funcția **loadData** poate fi utilizată în multiple locuri din aplicație, cu diferite callback-uri care sunt apelate după încărcarea datelor. Acest lucru permite reutilizarea codului și separarea responsabilităților în mod clar între diferite părți ale aplicației.

Un alt exemplu de utilizare a unui callback poate fi atunci când doriți să sortați un array de numere și să specificați modul în care sunt sortate acestea. În loc să definiți o funcție de sortare care să funcționeze doar într-un mod specific, puteți trece o funcție de comparare ca argument care poate fi personalizată în funcție de necesități.

De exemplu:

```
function sortNumbers(numbers, comparisonFunction) {
  return numbers.sort(comparisonFunction);
}

const numbers = [1, 3, 5, 2, 4];

// Sortare crescătoare
const ascendingOrder = (a, b) => a - b;
console.log(sortNumbers(numbers, ascendingOrder)); // [1, 2, 3, 4, 5]

// Sortare descrescătoare
const descendingOrder = (a, b) => b - a;
console.log(sortNumbers(numbers, descendingOrder)); // [5, 4, 3, 2, 1]
```

În acest exemplu, funcția **sortNumbers** poate fi utilizată pentru a sorta orice array de numere, cu orice funcție de comparare care poate fi definită pentru a personaliza modul în care sunt sortate acestea.

8. Funcții anonime

Funcțiile anonime sunt funcții fără nume care sunt adesea utilizate în JavaScript ca argumente pentru alte funcții sau pentru a fi atribuite unei variabile. Acestea sunt create în momentul în care sunt invocate și nu pot fi apelate în altă parte din cod.

De exemplu:

```
// Funcție anonimă ca argument
setTimeout(function() {
  console.log('Așteptat 3 secunde.');
}, 3000);

// Funcție anonimă atribuită unei variabile
const logHello = function() {
  console.log('Hello!');
};

logHello();
```

În exemplul de mai sus, funcția anonimă este folosită ca argument pentru funcția **setTimeout** și este atribuită variabilei **logHello**. Aceste funcții anonime pot fi, de asemenea, înlocuite cu funcții arrow, care sunt o sintaxă mai scurtă pentru funcțiile anonime.

9. Metoda bind()

Metoda **bind** din JavaScript este utilizată pentru a lega o anumită valoare la **this** într-o funcție. Acest lucru poate fi util atunci când se dorește ca o funcție să fie executată cu o anumită valoare a **this**, indiferent de contextul în care este apelată.

De exemplu:

```
const person = {
  name: 'John Doe',
  sayHello: function() {
    console.log(`Hello, my name is ${this.name}`);
  }
};

const sayHello = person.sayHello.bind(person);
sayHello(); // Output: Hello, my name is John Doe
```

În exemplul de mai sus, metoda **sayHello** este legată la obiectul **person** cu ajutorul metodei **bind**. Astfel, atunci când funcția **sayHello** este apelată, **this** se referă la obiectul **person**, permițând afișarea numelui **John Doe**.

Probleme practice

1. Scrieți o funcție **calculateSum** care primește 2 parametri și returnează suma lor.
2. Scrieți o funcție **convertToCelsius** care primește o temperatură în Fahrenheit și returnează conversia în Celsius.
3. Scrieți o funcție **greatest** care primește o listă de numere și returnează cel mai mare număr din listă.
4. Scrieți o funcție **longestString** care primește o listă de șiruri și returnează cel mai lung șir din listă.
5. Scrieți o funcție **countVowels** care primește un șir de caractere și returnează numărul de vocale din șir.
6. Scrieți o funcție **reverseString** care primește un șir de caractere și returnează șirul inversat.

7. Scrieți o funcție **fibonacci** care returnează un array cu primele n numere din secvența Fibonacci.
8. Scrieți o funcție **isPalindrome** care primește un șir de caractere și returnează true dacă șirul este un palindrom și false în caz contrar.
9. Scrieți o funcție **sumArray** care primește un array de numere și returnează suma lor.
10. Scrieți o funcție **average** care primește un array de numere și returnează media aritmetică a acestora.
11. Scrieți o funcție **filterArray** care primește un array de numere și un număr x și returnează un array cu numerele mai mari decât x.
12. Scrieți o funcție **multiplyBy** care primește un număr și returnează o altă funcție care primește un alt număr și îl înmulțește cu numărul primit în prima funcție.
13. Scrieți o funcție **counter** care returnează o funcție anonimă care, atunci când este apelată, incrementează și returnează un număr.
14. Scrieți o funcție **createAdder** care primește un număr și returnează o altă funcție care, atunci când este apelată, adaugă numărul primit în prima funcție la argumentul său.
15. Scrieți o funcție **makeArray** care returnează o funcție anonimă care poate adăuga sau returna valori într-un array.
16. Scrieți o funcție **sum** care utilizează apply pentru a adăuga toți parametrii primiți ca argumente.
17. Scrieți o funcție calculator care returnează o funcție care poate efectua operații matematice de bază (adunare, scădere, înmulțire, împărțire) utilizând valori legate la this.
18. Scrieți o funcție **createPerson** care utilizează bind pentru a lega valori la proprietățile unui obiect și returnează obiectul legat.
19. Scrieți o funcție **createObject** care returnează un obiect cu o proprietate greet care este o funcție anonimă care afișează Hello!. Utilizați call sau apply pentru a schimba valoarea this în interiorul funcției.

Probleme individuale:

1. Scrie o functie care sa primeasca doua numere ca argumente si sa returneze suma acestora. Foloseste metoda **bind**() pentru a lega primul argument la functie.
2. Scrie o functie care sa returneze un salut personalizat in functie de numele specificat ca argument. Foloseste metoda **bind**() pentru a lega un nume specific la functie.
3. Scrie o functie care sa returneze o propozitie cu numele unui fruct si culoarea acestuia. Foloseste metoda **bind**() pentru a lega un fruct specific si culoarea acestuia la functie.

4. Scrie o functie care sa primeasca doua numere si sa execute o functie de adunare, care sa returneze suma numerelor, folosind conceptul de callback function.
5. Scrie o functie care sa primeasca un sir de cuvinte si sa execute o functie care sa returneze numarul total de litere din acest sir, folosind conceptul de callback function.
6. Scrie o functie care sa primeasca un sir de numere si sa execute o functie care sa returneze suma numerelor din sir, folosind conceptul de callback function.
7. Scrie o functie care sa primeasca un sir de numere si sa execute o functie care sa returneze cel mai mare numar din sir, folosind conceptul de callback function.
8. Scrie o functie care sa primeasca un sir de numere si sa execute o functie care sa returneze cel mai mic numar din sir, folosind conceptul de callback function.

QUIZ

Capitol IV. Obiecte de scriptare.

UNITĂȚI DE CONȚINUT

1) Noțiunea de obiect.
2) Definirea obiectului.
3) Proprietățile și metodele obiectului.
4) Obiecte native (integrate):
   - Number;
   - Boolean;
   - String;
   - Math;
   - Date;
   - RegExp;
   - Array;
   - JSON.

ABILITĂȚI

- Crearea obiectelor.
- Definirea proprietăților și metodelor obiectelor.
- Aplicarea obiectelor la prelucrarea datelor din pagini web.

În JavaScript, un **obiect** este o colecție de proprietăți, care sunt perechi de cheie-valoare. Proprietățile pot fi variabile sau funcții și sunt grupate împreună într-un singur obiect.

Obiectele pot fi create folosind sintaxa literalei de obiect {} sau prin intermediul constructorilor de obiecte, cum ar fi **new Object()**, **new Array()**, **new Date()**, etc.

1. Noțiunea de obiect

Un obiect poate fi văzut ca o entitate autonomă care poate fi manipulată independent. Acest lucru permite programatorilor să organizeze codul lor în mod modular, să încapsuleze date și funcționalitate, să creeze abstracțiuni și să simuleze obiecte din lumea reală în codul lor. Obiectele sunt o parte fundamentală a programării orientate spre obiecte și sunt utilizate în mod extensiv în JavaScript și în alte limbaje de programare.

Unele dintre cele mai comune obiecte de scriptare în JavaScript includ:

- Obiectul "**Math**" - utilizat pentru a efectua operații matematice complexe.
- Obiectul "**Date**" - utilizat pentru a manipula date și ore.
- Obiectul "**String**" - utilizat pentru a manipula șiruri de caractere.
- Obiectul "**Array**" - utilizat pentru a manipula tablouri de date.
- Obiectul "**Function**" - utilizat pentru a defini funcții.
- Obiectul "**Object**" - utilizat pentru a crea și manipula obiecte.
- Obiectul "**RegExp**" - utilizat pentru a efectua căutări și înlocuiri în șiruri de caractere.

➢ Obiectul "**JSON**" - utilizat pentru a lucra cu date JSON.

Aceste obiecte pot fi utilizate în diferite moduri pentru a crea script-uri complexe și interactivități pe paginile web.

De exemplu:

```
const person = {
    name: 'John',
    age: 30,
    occupation: 'Developer',
    greet: function () {
        console.log(`Hello, my name is ${this.name}.
        I'm a ${this.occupation} and I'm ${this.age} years old.`);
    }
};
```

Acesta este un obiect numit "person" care are patru proprietăți: "name", "age", "occupation" și "greet". Prima trei proprietăți sunt variabile care stochează numele, vârsta și ocupația persoanei. Ultima proprietate este o metodă care afișează un mesaj de salut care conține aceste proprietăți.

Metoda "greet" folosește cuvântul cheie "this" pentru a accesa proprietățile din același obiect. În acest exemplu, proprietățile sunt utilizate pentru a genera un mesaj de salut personalizat.

Acesta este doar un exemplu simplu, dar obiectele pot fi mult mai complexe și pot fi folosite într-o varietate de moduri pentru a organiza și manipula date și funcționalități în JavaScript.

2. Definirea obiectului

Obiectele pot fi definite în JavaScript folosind literalul de obiect {} sau prin intermediul constructorilor de obiecte, cum ar fi **new Object(), new Array(), new Date()**, etc. Literalul de obiect este cel mai comun și cel mai simplu mod de a defini un obiect în JavaScript.

De exemplu:

```
const car = {
  make: 'Ford',
  model: 'Mustang',
  year: 2022,
  color: 'red',
  isConvertible: true,
  drive: function() {
    console.log(`The ${this.make} ${this.model} is driving.`);
  }
};
```

Acesta este un obiect numit "car" care are șase proprietăți: "make", "model", "year", "color", "isConvertible" și "drive". Primele cinci proprietăți sunt variabile care stochează informații despre mașină, cum ar fi marca, modelul, anul, culoarea și dacă este sau nu decapotabilă. Ultima proprietate este o metodă care afișează un mesaj care spune că mașina este în mișcare.

Metoda "drive" utilizează cuvântul cheie "this" pentru a accesa proprietățile din același obiect. În acest exemplu, proprietățile sunt utilizate pentru a genera un mesaj care indică marca și modelul mașinii și faptul că este în mișcare.

3. Proprietățile și metodele obiectului

Proprietățile și metodele sunt elemente esențiale ale obiectelor în JavaScript. Proprietățile sunt variabilele care stochează date sau valori într-un obiect, iar metodele sunt funcțiile care pot fi apelate pentru a efectua operații cu aceste proprietăți sau cu alte elemente din obiect.

Iată câteva exemple de proprietăți și metode pentru un obiect în JavaScript:

```
const person = {
  name: 'John',
  age: 30,
  occupation: 'Developer',
  greet: function() {
    console.log(`Hello, my name is ${this.name}. I'm a ${this.occupation}
  }
};
```

În exemplul de mai sus, "person" este un obiect care are patru proprietăți:

1. "**name**" este o proprietate care stochează numele persoanei.
2. "**age**" este o proprietate care stochează vârsta persoanei.
3. "**occupation**" este o proprietate care stochează ocupația persoanei.
4. "**greet**" este o proprietate care este o metodă și care afișează un mesaj de salut care conține proprietățile "name", "occupation" și "age" ale obiectului.

Putem accesa proprietățile unui obiect folosind operatorul punct, astfel:

```
console.log(person.name); // output: John
console.log(person.age); // output: 30
console.log(person.occupation); // output: Developer
```

Putem apela și metoda "**greet**" astfel:

```
person.greet(); // output: Hello, my name is John. I'm a Developer
```

Obiectele pot avea și alte metode și proprietăți în funcție de scopul lor. De exemplu, un obiect "**car**" ar putea avea proprietăți precum "**make**", "**model**", "**year**" și "**color**" și metode precum "**start**", "**stop**", "**accelerate**" și "**brake**".

4. Obiectele native

Obiectele native în JavaScript sunt obiecte predefinite în limbajul JavaScript și care sunt disponibile global în orice mediu JavaScript, cum ar fi într-un browser sau în Node.js. Acestea sunt definite de specificația JavaScript și sunt construite în limbajul însuși.

Iată câteva exemple de obiecte native în JavaScript:

- Obiectul **String** - acest obiect este utilizat pentru lucrul cu șiruri de caractere. El oferă o serie de metode utile pentru manipularea șirurilor de caractere, cum ar fi toUpperCase() pentru a converti un șir la majuscule și indexOf() pentru a căuta un șir într-un alt șir.
- Obiectul **Number** - acest obiect este utilizat pentru lucrul cu numere. El oferă o serie de metode utile pentru manipularea numerelor, cum ar fi toFixed() pentru a fixa numărul de zecimale și isNaN() pentru a verifica dacă valoarea este NaN.
- Obiectul **Boolean** - acest obiect este utilizat pentru lucrul cu valori booleane (true sau false). El oferă o singură metodă, valueOf(), care returnează valoarea booleană a obiectului.
- Obiectul **Array** - acest obiect este utilizat pentru lucrul cu liste de elemente. El oferă o serie de metode utile pentru manipularea și accesarea elementelor dintr-un array, cum ar fi push() pentru a adăuga un element la sfârșitul unui array și sort() pentru a sorta elementele unui array.
- Obiectul **Date** - acest obiect este utilizat pentru lucrul cu date și ore. El oferă o serie de metode utile pentru a obține și seta diferite componente de dată și oră, cum ar fi getMonth() pentru a obține luna curentă și setFullYear() pentru a seta anul.

Acestea sunt doar câteva exemple de obiecte native în JavaScript, dar există și alte obiecte native cum ar fi **Math, RegExp** sau **Error**. Aceste obiecte sunt utilizate frecvent în programarea JavaScript pentru a efectua diferite operații.

Obiectul Number

În JavaScript, obiectul **Number** este un obiect global care oferă o funcționalitate pentru lucrul cu numere. Obiectul Number poate fi utilizat pentru a crea noi instanțe de numere și pentru a accesa diferite proprietăți și metode utile pentru lucrul cu numere.

Iată câteva exemple de utilizare a obiectului Number în JavaScript:

1. Crearea unei instanțe de număr:

```
const number = new Number(42);
console.log(number); // output: 42
```

2. Accesarea proprietăților obiectului Number:

```
console.log(Number.MAX_VALUE); // output: 1.7976931348623157e+308
console.log(Number.MIN_VALUE); // output: 5e-324
```

3. Utilizarea metodelor obiectului Number:

```
const num = 3.14159;
console.log(num.toFixed(2)); // output: 3.14
console.log(num.toString()); // output: "3.14159"
console.log(Number.isInteger(num)); // output: false
```

Metodele obiectului Number sunt utile pentru a efectua diferite operații cu numere. În exemplul de mai sus, metoda toFixed() este utilizată pentru a fixa numărul de zecimale afișate, metoda toString() este utilizată pentru a converti numărul într-un șir de caractere, iar metoda isInteger() este utilizată pentru a verifica dacă numărul este un număr întreg.

Obiectul Boolean

În JavaScript, obiectul **Boolean** este un tip de date care poate fi utilizat pentru a reprezenta o valoare booleană, adică **true** sau **false**. Boolean este, de asemenea, un obiect global în JavaScript, cu un constructor și două proprietăți și metode predefinite.

Constructorul **Boolean**() poate fi utilizat pentru a crea un nou obiect Boolean. Acesta poate primi o singură valoare ca argument, care este convertită într-o valoare booleană. De asemenea, obiectul Boolean are două proprietăți predefinite: **Boolean.prototype** și **Boolean.length**. Proprietatea Boolean.prototype este utilizată pentru a adăuga proprietăți și metode la toate obiectele Boolean, în

timp ce proprietatea Boolean.length este setată la 1 și indică numărul de argumente așteptate de constructorul Boolean().

Obiectul Boolean are două metode predefinite: **Boolean.toString()** și **Boolean.valueOf()**. **Boolean.toString()** este utilizată pentru a converti o valoare booleană într-un șir de caractere, iar **Boolean.valueOf()** este utilizată pentru a obține valoarea booleană a unui obiect Boolean.

De exemplu:

```
// crearea unui obiect Boolean
const myBoolean = new Boolean(true);

// afișarea proprietăților și metodelor obiectului Boolean
console.log(myBoolean.constructor);
// afișează constructorul obiectului Boolean
console.log(myBoolean.valueOf());
// afișează valoarea booleană a obiectului Boolean
console.log(myBoolean.toString());
// afișează valoarea obiectului Boolean sub formă de șir de caractere
```

În acest exemplu, se crează un obiect Boolean numit **myBoolean** cu valoarea true. Apoi, se utilizează metoda constructor pentru a afișa constructorul obiectului Boolean, metoda valueOf pentru a obține valoarea booleană a obiectului și metoda toString pentru a afișa valoarea obiectului sub formă de șir de caractere.

Obiectul String

În JavaScript, obiectul String este un tip de date utilizat pentru a reprezenta șiruri de caractere. String este, de asemenea, un obiect global în JavaScript, cu un constructor și o serie de proprietăți și metode predefinite.

Constructorul String() poate fi utilizat pentru a crea un nou obiect String. Acesta poate primi una sau mai multe valori ca argumente, care sunt convertite într-un șir de caractere. De asemenea, obiectul String are o serie de proprietăți predefinite, cum ar fi String.prototype și String.length. Proprietatea String.prototype este utilizată pentru a adăuga proprietăți și metode la toate obiectele String, în timp ce proprietatea String.length este setată la 1 și indică numărul de argumente așteptate de constructorul String().

Obiectul **String** are o serie de metode predefinite care pot fi utilizate pentru a efectua diferite operații asupra șirurilor de caractere. De exemplu, metoda **charAt**() este utilizată pentru a obține caracterul de la un anumit index în șirul de caractere, iar metoda **substring**() este utilizată pentru a obține o parte din șirul de caractere.

De exemplu:

```
// crearea unui obiect String
const myString = new String('Hello, world!');

// afișarea proprietăților și metodelor obiectului String
console.log(myString.constructor);
// afișează constructorul obiectului String
console.log(myString.charAt(0));
// afișează primul caracter din șirul de caractere
console.log(myString.substring(0, 5));
// afișează primele 5 caractere din șirul de caractere
```

În acest exemplu, se crează un obiect **String** numit **myString** cu valoarea 'Hello, world!'. Apoi, se utilizează metoda constructor pentru a afișa constructorul obiectului **String**, metoda **charAt** pentru a obține primul caracter din șirul de caractere și metoda **substring** pentru a obține primele 5 caractere din șirul de caractere.

Obiectul Math

Obiectul **Math** este un obiect global în JavaScript, utilizat pentru a efectua operații matematice. Acesta conține o serie de proprietăți și metode predefinite, care permit efectuarea de operații matematice complexe.

Math nu este un constructor și, prin urmare, nu poate fi utilizat pentru a crea obiecte. În schimb, proprietățile și metodele obiectului Math pot fi accesate direct, fără a fi necesară crearea unui obiect. Iată câteva dintre metodele și proprietățile predefinite ale obiectului **Math**:

- **Math.PI** - returnează valoarea lui pi (3.141592653589793)
- **Math.sqrt(x)** - returnează rădăcina pătrată a unui număr
- **Math.pow(x, y)** - returnează x la puterea y
- **Math.round(x)** - returnează numărul întreg cel mai apropiat de x
- **Math.floor(x)** - returnează cel mai mare număr întreg mai mic sau egal cu x
- **Math.ceil(x)** - returnează cel mai mic număr întreg mai mare sau egal cu x
- **Math.random()** - returnează un număr pseudo-aleatoriu între 0 și 1

De exemplu:

```
// generarea unui număr aleatoriu între 1 și 10
const randomNum = Math.floor(Math.random() * 10) + 1;

// afișarea valorii lui pi
console.log(Math.PI);

// afișarea rădăcinii pătrate a unui număr
console.log(Math.sqrt(16));

// afișarea unui număr ridicat la o putere
console.log(Math.pow(2, 3));

// rotunjirea unui număr
console.log(Math.round(4.6));

// obținerea unui număr întreg mai mic sau egal cu un număr dat
console.log(Math.floor(4.6));

// obținerea unui număr întreg mai mare sau egal cu un număr dat
console.log(Math.ceil(4.6));
```

În acest exemplu, se generează un număr aleatoriu între 1 și 10 utilizând metoda **Math.random**(), care returnează un număr pseudo-aleatoriu între 0 și 1. Acest număr este apoi înmulțit cu 10, utilizând metoda **Math.floor**(), pentru a obține un număr întreg între 0 și 9, la care se adaugă 1 pentru a obține un număr întreg între 1 și 10.

Apoi, se afișează valoarea lui pi utilizând proprietatea **Math.PI**, se calculează rădăcina pătrată a unui număr utilizând metoda **Math.sqrt**(), se calculează un număr ridicat la o putere utilizând metoda **Math.pow**(), se rotunjește un număr la cel mai apropiat întreg utilizând metoda **Math.round**(), se obține cel mai mare număr întreg mai mic sau egal cu un număr dat utilizând metoda **Math.floor().**

Obiectul Date

Obiectul Date este utilizat pentru lucrul cu date și ore în JavaScript și este utilizat pentru a obține și seta informații despre data și ora curentă.

Un obiect Date poate fi creat utilizând constructorul Date() și oferă o serie de metode pentru a accesa și manipula informațiile despre dată și oră.

De exemplu, pentru a crea un obiect Date care reprezintă data și ora curentă, putem folosi următorul cod:

```
const currentDate = new Date();
```

Acest lucru va crea un obiect **Date** care va reprezenta data și ora curentă, care poate fi utilizat pentru a accesa informații precum anul, luna, ziua, ora, minutul, secunda și milisecunda.

De asemenea, obiectul **Date** oferă metode pentru a manipula și formata datele. De exemplu, metoda **getFullYear**() poate fi utilizată pentru a obține anul curent, în timp ce metoda **toLocaleDateString**() poate fi utilizată pentru a formata data într-un format localizat.

De exemplu:

```
const currentDate = new Date();

const currentYear = currentDate.getFullYear(); // Obține anul curent
const formattedDate = currentDate.toLocaleDateString(); // Formatează data
```

Obiectul **Date** are, de asemenea, capacitatea de a realiza operații matematice cu date și ore, ceea ce permite, de exemplu, adăugarea sau scăderea unui număr specific de zile sau ore dintr-o dată.

Metodele obiectului Date:

- **getDate**(): obține ziua din luna curentă (începând cu 1).
- **getDay**(): obține ziua săptămânii (începând cu Duminică = 0).
- **getFullYear**(): obține anul.
- **getHours**(): obține ora (0 - 23).
- **getMilliseconds**(): obține milisecundele (0 - 999).
- **getMinutes**(): obține minutele (0 - 59).
- **getMonth**(): obține luna (începând cu Ianuarie = 0).
- **getSeconds**(): obține secundele (0 - 59).
- **getTime**(): obține timpul în milisecunde din 1 ianuarie 1970.
- **getTimezoneOffset**(): obține diferența dintre fusul orar al sistemului și ora UTC, în minute.
- **setDate**(): setează ziua din luna curentă (începând cu 1).

- **setFullYear**(): setează anul.
- **setHours**(): setează ora (0 - 23).
- **setMilliseconds**(): setează milisecundele (0 - 999).
- **setMinutes**(): setează minutele (0 - 59).
- **setMonth**(): setează luna (începând cu Ianuarie = 0).
- **setSeconds**(): setează secundele (0 - 59).

Acestea sunt doar câteva exemple de metode. Obiectul **Date** are multe alte metode utile pentru a manipula datele și ora.

Obiectul RegExp

Obiectul **RegExp** este utilizat pentru a crea și manipula expresii regulate în JavaScript. Expresiile regulate sunt șiruri de caractere care definesc un șablon de căutare. Cu ajutorul obiectului RegExp, puteți căuta, înlocui și valida șiruri de caractere în funcție de un șablon.

Pentru a crea o expresie regulată, puteți utiliza sintaxa literalei **/pattern/modifiers**, unde **pattern** este șablonul de căutare și **modifiers** sunt modificatorii care pot fi utilizați pentru a face căutarea mai flexibilă.

De exemplu, expresia regulată /hello/ caută orice șir care conține cuvântul "hello". Modificatorul i poate fi adăugat pentru a face căutarea nesensibilă la majuscule și minuscule, astfel încât expresia regulată /hello/i va găsi șiruri care conțin "hello", "Hello" sau "HELLO".

Obiectul RegExp are câteva metode utile, printre care se numără:

- **test**(): verifică dacă un șir corespunde expresiei regulate și returnează true sau false.
- **exec**(): caută un șir care corespunde expresiei regulate și returnează o matrice cu informații despre primul șir găsit.
- **toString**(): returnează șirul de caractere care reprezintă expresia regulată.

Metoda "match" este o metodă a obiectului "RegExp" din JavaScript, care este folosită pentru a căuta un șir de caractere care se potrivește cu un model specificat de expresie regulată (RegExp). Metoda "match" poate fi apelată pe un șir de caractere și poate lua ca argument un obiect RegExp sau o expresie regulată definită ca șir de caractere.

Sintaxa metodei "match" este următoarea:

```
string.match(regexp)
```

unde "string" este șirul de caractere pe care dorim să îl căutăm, iar "regexp" este expresia regulată care specifică modelul pe care îl căutăm.

Metoda "match" returnează un array care conține informații despre prima potrivire găsită. Dacă nu este găsită nicio potrivire, metoda "match" returnează null. Array-ul returnat conține următoarele elemente:

- Elementul 0 - este întregul șir care a fost găsit ca potrivire
- Elementele 1, 2, 3 etc. - sunt subșirurile care corespund cu fiecare grup de captură definit în expresia regulată.

De exemplu, următorul cod JavaScript demonstrează cum se utilizează metoda "match" pentru a căuta un model specificat de expresie regulată într-un șir de caractere:

```
const str = "Acesta este un text de exemplu.";
const regex = /exemplu/;

const result = str.match(regex);
console.log(result);
```

În acest exemplu, metoda "match" este utilizată pentru a căuta șirul "exemplu" în șirul "str". Rezultatul afișat în consolă va fi un array cu un singur element ["exemplu"], deoarece nu s-a definit niciun grup de captură în expresia regulată.

Iată un exemplu simplu de utilizare a obiectului **RegExp** pentru a căuta cuvinte care conțin literele "a" și "b":

De exemplu:

```
const regex = /[ab]/;
const str = "abc123";
const result = regex.test(str); // true
```

În acest exemplu, metoda **test**() este utilizată pentru a verifica dacă șirul **str** conține literele "a" sau "b". Rezultatul va fi **true**, deoarece șirul conține litera "a".

- **source**: Această proprietate returnează textul sursei expresiei regulate sub forma unui șir.
- **flags**: Această proprietate returnează un șir conținând toate indicatoarele de tip (flags) utilizate în expresia regulată.

- **compile**(): Această metodă compila o expresie regulată într-un obiect de tip RegExp. Aceasta este o metodă învechită și nu este recomandată să fie utilizată în codul nou.

Modificatori

În JavaScript, expresiile regulate (**RegExp**) pot fi create cu ajutorul unor modificatori (**modifiers**) care afectează modul în care expresia regulată este interpretată. Iată câteva exemple de modificatori utilizate în expresiile regulate:

1. **g** - Modificatorul global. Acesta indică că trebuie căutate toate potrivirile, nu doar prima. **De exemplu:**

```
let str = "Acesta este un test de expresii regulate.";
let regex = /e/g;
let rezultat = str.match(regex);
console.log(rezultat);
```

Acest cod va returna toate instanțele caracterului "e" din șirul de text.

2. **i** - Modificatorul de insensibilitate la majuscule și minuscule. Acesta indică faptul că expresia regulată trebuie să fie potrivită, indiferent de literele mari sau mici. **De exemplu:**

```
let str = "Acesta este un test de expresii regulate.";
let regex = /test/i;
let rezultat = str.match(regex);
console.log(rezultat);
```

Acest cod va returna cuvântul "test" din șirul de text, indiferent de faptul că începe cu o literă mare sau mică.

3. **m** - Modificatorul de căutare în multiple linii. Acesta indică faptul că expresia regulată trebuie să fie potrivită pe mai multe linii. **De exemplu:**

```
let str = "Acesta este un test de expresii regulate.\nAcesta este un al doilea rând.";
let regex = /^Acesta/gm;
let rezultat = str.match(regex);
console.log(rezultat);
```

Acest cod va returna toate cuvintele "Acesta" care încep o linie nouă, indiferent de faptul că acestea apar pe mai multe linii.

4. **s** - Modificatorul de căutare de text. Acesta indică faptul că expresia regulată trebuie să fie potrivită cu textul format din linii multiple. **De exemplu:**

```
let str = "Acesta este un test de expresii regulate.\nAcesta este un al doilea rând.";
let regex = /test.*al doilea/gs;
let rezultat = str.match(regex);
console.log(rezultat);
```

Acest cod va returna textul între cuvintele "test" și "al doilea", indiferent de numărul de linii pe care apar acestea.

Acestea sunt câteva dintre modificatorii utilizate frecvent în expresiile regulate din JavaScript.

Obiectul Array

Obiectul **Array** este utilizat în JavaScript pentru a stoca o colecție de valori într-un singur loc. Aceste valori pot fi orice tip de date JavaScript, inclusiv alte obiecte, iar elementele dintr-un **Array** sunt accesate folosind un index.

Un obiect **Array** poate fi creat prin utilizarea constructorului **Array** sau prin notarea unui set de valori între paranteze pătrate **[]** separate prin virgule.

De exemplu:

```
// Crearea unui array prin constructor
let myArray = new Array();
myArray[0] = "apple";
myArray[1] = "banana";
myArray[2] = "orange";

// Crearea unui array prin notarea valorilor
let myOtherArray = ["apple", "banana", "orange"];
```

Metodele disponibile pe obiectul **Array** includ:

- **concat**(): concatenarea două sau mai multe Array-uri
- **copyWithin**(): copierea unui interval de elemente în cadrul Array-ului
- **every**(): verificarea dacă toate elementele îndeplinesc o anumită condiție
- **fill**(): umplerea tuturor elementelor cu o anumită valoare
- **filter**(): filtrarea elementelor care îndeplinesc o anumită condiție
- **find**(): returnarea primului element care îndeplinește o anumită condiție
- **findIndex**(): returnarea indexului primului element care îndeplinește o anumită condiție
- **forEach**(): executarea unei funcții pentru fiecare element din Array
- **includes**(): verificarea dacă Array-ul conține o anumită valoare

- **indexOf**(): returnarea indexului primei apariții a unei anumite valori
- **join**(): concatenarea tuturor elementelor într-un șir de caractere folosind un separator specificat
- **lastIndexOf**(): returnarea indexului ultimei apariții a unei anumite valori
- **map**(): transformarea fiecărui element într-un alt element folosind o funcție specificată
- **pop**(): eliminarea ultimului element din Array
- **push**(): adăugarea unui element la sfârșitul Array-ului
- **reduce**(): reducerea tuturor elementelor la o singură valoare folosind o funcție specificată
- **reduceRight**(): reducerea tuturor elementelor la o singură valoare începând de la ultimul element al Array-ului
- **reverse**(): inversarea ordinei elementelor în Array
- **shift**(): eliminarea primului element din Array
- **slice**(): crearea unui nou Array care conține doar un interval specificat de elemente din Array-ul original
- **some**(): verificarea dacă cel puțin un element îndeplinește o anumită condiție
- **sort**(): sortarea elementelor în Array folosind o funcție specificată sau ordinea lexicografică implicită

- Metoda filter

din obiectul **Array** este utilizată pentru a filtra elementele unui array, în funcție de o condiție specificată printr-o funcție. Metoda returnează un nou array cu toate elementele care au trecut testul specificat.

Iată câteva exemple de utilizare a metodei **filter**:

1. Filtrarea numerelor pare dintr-un array:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const evenNumbers = numbers.filter((number) => number % 2 === 0);
console.log(evenNumbers); // Output: [2, 4, 6, 8, 10]
```

2. Filtrarea stringurilor care conțin un anumit caracter:

```
const words = ['apple', 'banana', 'cherry', 'date', 'eggplant'];
const wordsContainingA = words.filter((word) => word.includes('a'));
console.log(wordsContainingA); // Output: ['apple', 'banana', 'date']
```

3. Filtrarea obiectelor în funcție de proprietățile lor:

```
const products = [
  { name: 'Apple', category: 'Fruit', price: 1 },
  { name: 'Carrot', category: 'Vegetable', price: 2 },
  { name: 'Banana', category: 'Fruit', price: 2 },
  { name: 'Broccoli', category: 'Vegetable', price: 3 },
  { name: 'Cherry', category: 'Fruit', price: 4 }
];
const fruitProducts = products.filter((product) => product.category === 'Fruit');
console.log(fruitProducts);
/*
Output:
[
  { name: 'Apple', category: 'Fruit', price: 1 },
  { name: 'Banana', category: 'Fruit', price: 2 },
  { name: 'Cherry', category: 'Fruit', price: 4 }
]
*/
```

- Metoda forEach

a obiectului Array este utilizată pentru a itera prin fiecare element dintr-un array și pentru a efectua o acțiune pentru fiecare element. Aceasta primește ca argument o funcție de tip **callback**, care va fi apelată pentru fiecare element al array-ului.

Iată câteva exemple de utilizare a metodei **forEach**:

1. Afișarea fiecărui element al unui array:

```
const myArray = [1, 2, 3, 4, 5];

myArray.forEach(element => console.log(element));
// Output: 1 2 3 4 5
```

2. Adunarea tuturor elementelor dintr-un array:

```
const myArray = [1, 2, 3, 4, 5];
let sum = 0;

myArray.forEach(element => sum += element);
console.log(sum); // Output: 15
```

3. Afișarea tuturor elementelor dintr-un array care îndeplinesc o anumită condiție:

```
const myArray = [1, 2, 3, 4, 5];

myArray.forEach(element => {
  if (element % 2 === 0) {
    console.log(element);
  }
});
// Output: 2 4
```

4. Actualizarea valorii fiecărui element dintr-un array:

```
const myArray = [1, 2, 3, 4, 5];

myArray.forEach((element, index, array) => {
  array[index] = element * 2;
});

console.log(myArray); // Output: [2, 4, 6, 8, 10]
```

5. Iterarea prin fiecare element al unui array multidimensional:

```
const myArray = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

myArray.forEach(row => {
  row.forEach(element => {
    console.log(element);
  });
});
// Output: 1 2 3 4 5 6 7 8 9
```

- Metoda includes

1. Verificarea dacă un șir de caractere conține un anumit cuvânt cheie:

```
const str = "Aceasta este o propozitie de test.";
const keyword = "test";

if (str.includes(keyword)) {
  console.log(`Cuvântul cheie '${keyword}' a fost găsit în șir.`);
} else {
  console.log(`Cuvântul cheie '${keyword}' nu a fost găsit în șir.`);
}
```

2. Verificarea dacă un array conține o anumită valoare:

```
const arr = [1, 2, 3, 4, 5];
const value = 3;

if (arr.includes(value)) {
  console.log(`Valoarea ${value} a fost găsită în array.`);
} else {
  console.log(`Valoarea ${value} nu a fost găsită în array.`);
}
```

- Metoda join

este folosită pentru a uni toate elementele unui array într-un șir de caractere. Iată câteva exemple de utilizare a acestei metode:

1. Transformarea unui array de cuvinte într-un singur șir de caractere:

```
const words = ["Salut", "cum", "merge", "azi?"];
const sentence = words.join(" ");
console.log(sentence);
// Output: "Salut cum merge azi?"
```

2. Transformarea unui array de numere într-un șir de caractere separate prin virgulă:

```
const numbers = [1, 2, 3, 4, 5];
const str = numbers.join(",");
console.log(str);
// Output: "1,2,3,4,5"
```

3. Transformarea unui array de obiecte într-un șir de caractere pentru afișare:

```
const users = [
  { name: "Alex", age: 25 },
  { name: "Maria", age: 32 },
  { name: "Ion", age: 18 }
];
const str = users.map(user => `${user.name} (${user.age})`).join("; ");
console.log(str);
// Output: "Alex (25); Maria (32); Ion (18)"
```

- **Metoda indexOf**

Metoda **indexOf**() este o metodă a obiectului Array în JavaScript care este folosită pentru a căuta și returna indexul primului element dintr-un array care îndeplinește o anumită condiție. Dacă nu este găsit niciun element, metoda returnează -1.

Sintaxa metodei **indexOf**() este următoarea:

```
array.indexOf(elementCautat, indexStart)
```

- **elementCautat**: Elementul căutat în array.

- **indexStart** (opțional): Poziția din array de la care începe căutarea. Dacă nu este specificată, se începe de la indexul 0.

Exemple de utilizare a metodei indexOf():

```
const fruits = ['apple', 'banana', 'orange', 'grape', 'banana'];

console.log(fruits.indexOf('banana')); // Output: 1
console.log(fruits.indexOf('banana', 2)); // Output: 4
console.log(fruits.indexOf('pear')); // Output: -1
```

În primul exemplu, metoda **indexOf**() este utilizată pentru a căuta primul index al elementului **'banana'** în array-ul **fruits**. Deoarece **'banana'** se găsește la indexul 1, metoda returnează 1.

În al doilea exemplu, metoda indexOf() este utilizată pentru a căuta primul index al elementului 'banana' începând cu poziția indexului 2 din array-ul fruits. Deoarece 'banana' apare din nou la indexul 4, metoda returnează 4.

În ultimul exemplu, metoda indexOf() este utilizată pentru a căuta primul index al elementului 'pear' în array-ul fruits. Deoarece 'pear' nu se găsește în array, metoda returnează -1.

- Metoda map

Metoda **map()** este o metodă a obiectului Array în JavaScript, utilizată pentru a itera prin toate elementele dintr-un array și a aplica o funcție asupra fiecărui element. Această metodă nu modifică array-ul original, ci returnează un nou array, care conține valorile transformate.

Sintaxa metodei este următoarea:

```
array.map(function(currentValue, index, array) {
  // returnează element transformt
}, thisArg)
```

Parametrii metodei:

- **function**(currentValue, index, array): Funcția care este apelată pentru fiecare element din array. Funcția primește 3 parametri:
- **currentValue**: Valoarea elementului curent din array.
- **index**: Indexul elementului curent din array.
- **array**: Array-ul asupra căruia a fost apelată metoda map().
- **thisArg** (opțional): Obiectul utilizat drept this în apelul funcției de transformare.

Iată un exemplu de utilizare a metodei map() pentru a crea un nou array, care conține dublul fiecărui element din array-ul original:

```
const array = [1, 2, 3, 4, 5];
const doubledArray = array.map(function(num) {
  return num * 2;
});

console.log(doubledArray); // Output: [2, 4, 6, 8, 10]
```

În acest exemplu, funcția transformatoare primește ca parametru fiecare element din array, îl înmulțește cu 2 și returnează noua valoare. Metoda map() creează un nou array, care conține valorile returnate de funcția transformatoare.

O altă utilizare a metodei map() este de a extrage o anumită proprietate dintr-un array de obiecte. Să presupunem că avem un array de obiecte care conțin informații despre un produs:

```
const products = [
  { id: 1, name: 'Product 1', price: 10 },
  { id: 2, name: 'Product 2', price: 20 },
  { id: 3, name: 'Product 3', price: 30 }
];
```

Pentru a extrage un array de prețuri ale produselor, putem utiliza metoda map() astfel:

```
const prices = products.map(function(product) {
  return product.price;
});

console.log(prices); // Output: [10, 20, 30]
```

Funcția transformatoare primește ca parametru fiecare obiect din array-ul de produse și returnează valoarea proprietății **price**. Metoda **map()** creează un nou array, care conține prețurile extrase din obiectele din array-ul original.

- Metoda pop

Metoda **pop()** este o metodă pentru obiectul Array în JavaScript care elimină ultimul element dintr-un array și returnează valoarea acestuia. Metoda modifică array-ul original și reduce lungimea acestuia cu 1.

```
const colors = ['red', 'green', 'blue'];

console.log(colors); // Output: ["red", "green", "blue"]

const lastColor = colors.pop();

console.log(lastColor); // Output: "blue"

console.log(colors); // Output: ["red", "green"]
```

În acest exemplu, metoda **pop()** este utilizată pentru a elimina ultimul element din array-ul **colors**, adică "blue". Valoarea "blue" este returnată și atribuită variabilei **lastColor**, apoi array-ul **colors** este afișat din nou, fără ultimul element "blue".

- Metoda push

Metoda **push**() este o metodă pentru obiectul Array în JavaScript care adaugă unul sau mai multe elemente la sfârșitul unui array și returnează noua lungime a array-ului.

```
const animals = ['cat', 'dog', 'bird'];

console.log(animals); // Output: ["cat", "dog", "bird"]

const newLength = animals.push('hamster', 'fish');

console.log(newLength); // Output: 5

console.log(animals); // Output: ["cat", "dog", "bird", "hamster", "fish"]
```

În acest exemplu, metoda **push()** este utilizată pentru a adăuga două elemente, "hamster" și "fish", la sfârșitul array-ului **animals**. Noua lungime a array-ului este stocată în variabila **newLength**, apoi array-ul actualizat este afișat folosind metoda **console.log()**.

- Metoda reduce

Metoda **reduce()** este o metodă definită pe obiectul Array în JavaScript și este utilizată pentru a reduce un array la o valoare unică, prin aplicarea unei funcții reducer pe fiecare element din array.

Sintaxa metodei **reduce()** este următoarea:

```
array.reduce(function(reducer, initialValue), contextObject)
```

- **function(reducer, initialValue)** este o funcție care este aplicată pe fiecare element din array pentru a reduce elementele la o valoare unică. Această funcție acceptă două argumente:
  - **reducer** este o funcție care aplică o logică specifică de reducere pe fiecare element din array. Această funcție acceptă două argumente: **accumulator** și **currentValue**.
  - **initialValue** este o valoare inițială a **accumulatorului** și este opțională. Dacă este specificată, aceasta va fi valoarea inițială a **accumulatorului**. Dacă nu este specificată, prima valoare din array va fi utilizată ca valoare inițială a **accumulatorului**.
- **contextObject** este un obiect opțional care va fi utilizat ca valoare **this** pentru **reducer** și **initialValue**.

Funcția **reducer** trebuie să returneze o valoare care va fi stocată în **accumulator**. Această valoare va fi utilizată pentru a reduce următorul element din array. La final, valoarea stocată în **accumulator** va fi valoarea returnată de funcția **reduce()**.

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce(function(accumulator, currentValue) {
  return accumulator + currentValue;
}, 0);

console.log(sum); // Output: 15
```

În acest exemplu, funcția reducer primește două argumente: **accumulator** și **currentValue**. Valoarea inițială a **accumulatorului** este **0**. Pentru fiecare element din array, funcția reducer adaugă valoarea curentă la **accumulator** și returnează rezultatul. La final, valoarea stocată în **accumulator** este **15**, care este valoarea returnată de funcția **reduce()**.

- Metoda reverse

Metoda **reverse** este o metodă disponibilă pentru obiectul Array în JavaScript, care schimbă ordinea elementelor într-un array. Primul element devine ultimul element și așa mai departe, până la ultimul element care devine primul.

Sintaxa metodei este următoarea:

```
array.reverse()
```

Metoda **reverse()** nu necesită niciun argument și nu returnează un nou array. În schimb, metoda modifică array-ul existent.

```
const fruits = ['apple', 'banana', 'orange', 'grape'];
console.log(fruits); // ['apple', 'banana', 'orange', 'grape']

fruits.reverse();
console.log(fruits); // ['grape', 'orange', 'banana', 'apple']
```

În acest exemplu, array-ul **fruits** este inversat prin apelarea metodei **reverse()**. Array-ul modificat este afișat în consolă folosind **console.log()**.

- Metoda shift

Metoda **shift()** este o metodă disponibilă pentru obiectul **Array** din JavaScript, care elimină primul element dintr-un tablou și returnează valoarea acestuia. De asemenea, aceasta modifică lungimea tabloului original.

Iată un exemplu de utilizare a metodei **shift()** pentru a elimina primul element dintr-un tablou:

```
const myArray = ['a', 'b', 'c', 'd'];
const firstElement = myArray.shift();

console.log(firstElement); // afișează 'a'
console.log(myArray); // afișează ['b', 'c', 'd']
```

În exemplul de mai sus, metoda **shift()** este apelată pe tabloul **myArray**, ceea ce determină eliminarea primului element (**'a'**) și returnarea acesteia prin atribuirea valorii sale către variabila **firstElement**. Apoi, tabloul **myArray** este afișat pentru a arăta că primul element a fost eliminat.

- Metoda slice

Metoda **slice** este utilizată pentru a crea o copie a unei porțiuni dintr-un array existent. Ea returnează un nou array care conține elementele selectate din array-ul original, fără a modifica array-ul original. Metoda **slice** poate fi apelată pe un array și primește doi parametri opționali:

- **start** - Poziția de la care se începe extragerea elementelor. Aceasta poate fi un număr negativ care specifică o poziție relativă la sfârșitul array-ului. Dacă acest parametru este omis, se va începe de la începutul array-ului.
- **end** - Poziția până la care se extrag elementele. Aceasta poate fi un număr negativ care specifică o poziție relativă la sfârșitul array-ului. Dacă acest parametru este omis, se vor extrage toate elementele începând de la poziția specificată în primul parametru.

```
const fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry'];

// Extrage elementele de la poziția 2 până la sfârșitul array-ului
const extractedFruits = fruits.slice(2);

console.log(extractedFruits); // ['cherry', 'date', 'elderberry']

// Extrage elementele de la poziția 1 până la 3 (fără a include elementul de la poziția 3)
const extractedFruits2 = fruits.slice(1, 3);

console.log(extractedFruits2); // ['banana', 'cherry']
```

- Metoda sort

Metoda **sort()** este o metodă a obiectului **Array** din JavaScript, utilizată pentru a sorta elementele unui tablou într-o ordine specificată. Această metodă sortază elementele în locul lor, astfel încât ordinea originală a elementelor poate fi schimbată.

Sintaxa pentru metoda **sort()** este următoarea:

```
array.sort([compareFunction])
```

Unde:

- **array** este tabloul care va fi sortat.
- **compareFunction** este o funcție opțională care specifică modul în care elementele trebuie sortate. Dacă este omisă, elementele sunt sortate în ordine lexicografică (alfabetică). Dacă este specificată, funcția ar trebui să ia două argumente și să returneze un număr negativ, zero sau pozitiv, în funcție de relația dintre cele două argumente.

De exemplu:

```
const numere = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5];

// sortare lexicografică (alfabetică)
numere.sort();

console.log(numere); // [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]

// sortare în ordine descrescătoare
numere.sort((a, b) => b - a);

console.log(numere); // [9, 6, 5, 5, 5, 4, 3, 3, 2, 1, 1]
```

În acest exemplu, avem un tablou de numere întregi. Mai întâi, sortăm tabloul folosind metoda **sort()** fără argumente, ceea ce va ordona elementele în ordine lexicografică. După aceea, utilizăm metoda **sort()** cu o funcție de comparare personalizată pentru a sorta elementele în ordine descrescătoare.

Obiectul JSON

JSON (JavaScript Object Notation) este un format de date ușor de citit și de scris, utilizat pentru transmiterea datelor între aplicații. Este bazat pe sintaxa limbajului JavaScript, dar poate fi utilizat în orice limbaj de programare.

Obiectul JSON este o reprezentare a datelor în format JSON într-un mod ușor de utilizat de către programatori. Este un obiect JavaScript care conține metode și proprietăți pentru a manipula datele în format JSON.

Prin intermediul obiectului JSON, se poate converti datele în format JSON într-un obiect JavaScript sau invers, dintr-un obiect JavaScript în format JSON. De asemenea, obiectul JSON permite să se analizeze și să se genereze date JSON cu ușurință, folosind metode precum JSON.parse() și JSON.stringify().

În esență, obiectul JSON este o interfață de programare a aplicațiilor (API) pentru lucru cu formatul JSON, ceea ce îl face un instrument valoros pentru dezvoltatorii care trebuie să transmită sau să primească date în format JSON.

Obiectul JSON din JavaScript are două metode principale:

1. **JSON.parse()**: Această metodă este utilizată pentru a transforma un șir de caractere JSON într-un obiect JavaScript. Metoda acceptă un singur argument, care este șirul JSON de analizat. Dacă șirul JSON nu este valid, metoda va arunca o excepție.

Exemplu de utilizare:

```
const jsonStr = '{"name":"John", "age":30, "city":"New York"}';
const obj = JSON.parse(jsonStr);
console.log(obj); // {name: "John", age: 30, city: "New York"}
```

Metoda **JSON.parse()** poate include un al doilea parametru pentru a transforma datele JSON într-un format personalizat. Următorul exemplu arată cum să utilizăm un al doilea parametru pentru a transforma datele JSON într-un format personalizat.

```
const jsonStr = '{"name":"John", "age":30, "city":"New York", "isActive": "true"}'

// Transformăm valoarea proprietății "isActive" într-un Boolean
const obj = JSON.parse(jsonStr, function(key, value) {
  if (key === "isActive") {
    return value === "true";
  }
  return value;
});

console.log(obj); // {name: "John", age: 30, city: "New York", isActive: true}
```

În acest exemplu, al doilea parametru al metodei **JSON.parse()** este o funcție de transformare care este apelată pentru fiecare pereche cheie-valoare din JSON-ul analizat. Funcția primește două argumente: cheia și valoarea proprietății curente.

În acest caz, funcția de transformare schimbă valoarea proprietății "isActive" într-un Boolean. Dacă valoarea este "true", se returnează adevărat, altfel se returnează valoarea originală. Toate celelalte proprietăți sunt păstrate neschimbate în obiectul rezultat.

Această funcție de transformare poate fi utilă pentru a personaliza procesul de analizare a JSON-ului și a obține un obiect cu un format personalizat.

Un alt exemplu în care putem utiliza al doilea parametru al metodei **JSON.parse()** este atunci când trebuie să analizăm un fișier JSON care conține date într-un format specific. De exemplu, să

presupunem că avem un fișier JSON care conține date referitoare la un utilizator, dar datele de naștere ale utilizatorului sunt stocate sub forma unui șir de caractere în loc de un obiect de tip **Date**. Pentru a transforma această dată într-un obiect **Date**, putem utiliza o funcție de transformare personalizată.

```
const jsonString = '{"name":"John", "age":30, "city":"New York", "birthdate":"1992-10-23"}';

// Transformăm șirul de caractere "birthdate" într-un obiect Date
const obj = JSON.parse(jsonString, function(key, value) {
  if (key === "birthdate") {
    return new Date(value);
  }
  return value;
});

console.log(obj);
// {name: "John", age: 30, city: "New York", birthdate: Wed Oct 23 1992 00:00:00 GMT+0300 (Eastern European Summer Time)}
```

În acest exemplu, funcția de transformare primește ca prim argument cheia și ca al doilea argument valoarea asociată cu acea cheie în fișierul JSON. Dacă cheia este "birthdate", atunci se transformă șirul de caractere asociat cu acea cheie într-un obiect **Date**, utilizând constructorul **Date()**. Toate celelalte chei și valorile asociate sunt păstrate neschimbate.

Funcția de transformare poate fi personalizată în funcție de necesitățile tale și poate fi folosită pentru a transforma datele JSON în formatul dorit.

2. **JSON.stringify()**: Această metodă este utilizată pentru a transforma un obiect JavaScript într-un șir de caractere JSON. Metoda acceptă două argumente: primul este obiectul JavaScript pe care îl doriți să îl transformați în JSON, iar al doilea este un argument opțional care poate fi utilizat pentru a filtra proprietățile obiectului.

```
const obj = {name: "John", age: 30, city: "New York"};
const jsonStr = JSON.stringify(obj);
console.log(jsonStr); // {"name":"John","age":30,"city":"New York"}
```

Metoda **JSON.stringify()** poate include un al doilea parametru pentru a filtra proprietățile obiectului, astfel încât să fie incluse doar acele proprietăți care corespund unui anumit criteriu. Următorul exemplu arată cum să utilizăm un al doilea parametru pentru a exclude proprietățile din obiectul de serializat.

```
const obj = { name: "John", age: 30, city: "New York", job: "Developer" };

// Filtrăm proprietățile obiectului în funcție de numele proprietății
const filteredObj = JSON.stringify(obj, ["name", "age", "city"]);
console.log(filteredObj); // {"name":"John","age":30,"city":"New York"}
```

În acest exemplu, al doilea parametru al metodei **JSON.stringify()** este un șir de caractere care reprezintă numele proprietăților care ar trebui incluse în șirul JSON rezultat. În acest caz, filtrăm obiectul pentru a include doar proprietățile "name", "age" și "city", iar proprietatea "job" este exclusă din șirul JSON rezultat.

De asemenea, putem utiliza un al doilea argument pentru a modifica valorile proprietăților obiectului în timpul serializării JSON. Mai jos, exemplul în care modificăm valorile proprietăților obiectului prin intermediul funcției de transformare.

```
const obj = { name: "John", age: 30, city: "New York" };

// Schimbăm valoarea proprietății "age" în 35 în timpul serializării
const jsonString = JSON.stringify(obj, function(key, value) {
  if (key === "age") {
    return 35;
  }
  return value;
});

console.log(jsonString); // {"name":"John","age":35,"city":"New York"}
```

În acest exemplu, funcția de transformare este pasată ca al doilea argument al metodei **JSON.stringify()**. Funcția primește două argumente: cheia și valoarea proprietății curente. Dacă cheia este "age", valoarea este schimbată în 35, în timp ce alte proprietăți sunt păstrate neschimbate.

Aceste două metode sunt principalele metode ale obiectului JSON și sunt utilizate pentru a transforma datele între formatul JSON și formatul JavaScript.

5. Destructurizare

Destructurizarea în JavaScript este un mod de a extrage valori dintr-un obiect sau un tablou și de a le asigna la variabile separate într-un mod mai concis și mai clar decât ar fi necesar cu alte metode.

Pentru a destructure un obiect, este nevoie de acolade **{}** și se poate utiliza sintaxa de nume variabilei obiectului pentru a extrage valoarea dintr-o proprietate specifică. De exemplu, dacă avem obiectul **{ nume: "John", varsta: 30 }**, putem extrage valorile cu ajutorul codului:

```
const { nume, varsta } = { nume: "John", varsta: 30 };
console.log(nume); // "John"
console.log(varsta); // 30
```

Pentru a destructure un tablou, este nevoie de paranteze pătrate **[]** și se poate utiliza sintaxa indexului pentru a extrage valoarea de la o anumită poziție. De exemplu, dacă avem tabloul **[1, 2, 3]**, putem extrage valorile cu ajutorul codului:

```
const [a, b, c] = [1, 2, 3];
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

Destructurizarea este utilă pentru a face codul mai clar și mai ușor de citit, în special atunci când lucrăm cu obiecte și tablouri mari.

Operatorul **rest** în JavaScript este reprezentat de trei puncte **...** și poate fi utilizat în două contexte diferite:

1. În cadrul unei funcții, operatorul rest este folosit pentru a captura un număr variabil de argumente și a le grupa într-un tablou. De exemplu, putem avea o funcție care primește un număr variabil de argumente și care returnează suma acestora:

```
function aduna(...numere) {
  let suma = 0;
  for (let numar of numere) {
    suma += numar;
  }
  return suma;
}

console.log(aduna(1, 2, 3)); // 6
console.log(aduna(4, 5)); // 9
console.log(aduna(10)); // 10
```

2. În cadrul unui tablou, operatorul rest este folosit pentru a extrage elementele rămase după ce am extras elemente anterioare. De exemplu, putem avea un tablou cu mai multe elemente, dar să dorim să extragem doar primele două:

```
const numere = [1, 2, 3, 4, 5];
const [primul, alDoilea, ...restul] = numere;
console.log(primul); // 1
console.log(alDoilea); // 2
console.log(restul); // [3, 4, 5]
```

În acest exemplu, operatorul rest a fost utilizat pentru a extrage toate elementele din tablou care nu sunt primul sau al doilea element.

Operatorul rest poate fi foarte util pentru a gestiona argumente variabile în funcții sau pentru a extrage elemente din tablouri într-un mod flexibil și eficient.

6. Operatorul Spread

Operatorul Spread, notat cu trei puncte "...", este un operator utilizat în limbajul de programare JavaScript pentru a transforma sau a extinde un element iterabil (cum ar fi un șir de caractere sau un obiect) într-o listă de elemente individuale. Acest operator permite programatorilor să transmită rapid și ușor argumente într-o funcție, să combine obiecte sau liste de elemente și să realizeze alte operațiuni similare.

În mod specific, operatorul Spread poate fi utilizat în următoarele moduri:

1. Pentru a transmite argumente într-o funcție:

```
function myFunction(arg1, arg2, arg3) {
  console.log(arg1, arg2, arg3);
}

let args = [1, 2, 3];
myFunction(...args); // va afișa: 1 2 3
```

2. Pentru a combina două sau mai multe liste într-o singură listă:

```
let list1 = [1, 2, 3];
let list2 = [4, 5, 6];
let combinedList = [...list1, ...list2];
console.log(combinedList); // va afișa: [1, 2, 3, 4, 5, 6]
```

3. Pentru a crea o copie a unui obiect:

```
let myObj = { x: 1, y: 2 };
let newObj = { ...myObj };
console.log(newObj); // va afișa: { x: 1, y: 2 }
```

Acestea sunt doar câteva exemple de utilizare a operatorului Spread. În general, operatorul Spread este o unealtă utilă pentru a lucra cu liste și obiecte în JavaScript și poate fi utilizat în multe alte moduri, în funcție de nevoile programatorului.

Operatorii Spread și Rest sunt doi operatori asemănători din JavaScript care sunt utilizați pentru a manipula liste și argumente într-un mod flexibil. Cu toate acestea, există diferențe semnificative între cei doi operatori.

Operatorul Spread se utilizează pentru a extinde o listă de elemente sau un obiect într-o altă listă sau obiect. Când este utilizat, operatorul Spread transformă un element iterabil, cum ar fi un șir de caractere, un obiect sau o listă, într-o listă de elemente individuale, care poate fi utilizată pentru a extinde o altă listă sau obiect. De exemplu:

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, ...arr1];
console.log(arr2); // va afișa [4, 5, 1, 2, 3]

const obj1 = {a: 1, b: 2};
const obj2 = {c: 3, ...obj1};
console.log(obj2); // va afișa {c: 3, a: 1, b: 2}
```

Operatorul Rest, pe de altă parte, se utilizează pentru a grupa un număr variabil de argumente intr-o singură listă. Acesta poate fi utilizat în definiția unei funcții pentru a captura orice număr de argumente, fie că sunt transmise sau nu prin apelarea funcției. De exemplu:

```
function myFunction(...args) {
  console.log(args);
}

myFunction(1, 2, 3); // va afișa [1, 2, 3]
myFunction(4, 5, 6, 7, 8); // va afișa [4, 5, 6, 7, 8]
```

Astfel, principalul rol al operatorului **Rest** este de a grupa un număr variabil de argumente intr-o singură listă, în timp ce operatorul **Spread** este utilizat pentru a extinde o listă de elemente sau un obiect într-un alt obiect sau listă.

Probleme practice

1. Creați unui obiect care va reprezenta un produs vândut pe un site web, cu proprietăți precum numele produsului, prețul, disponibilitatea și descrierea. Apoi, afișați informații despre produs pe pagina web, utilizând obiectul creat.
2. Scrieți o funcție care primește ca argument un obiect și afișează toate proprietățile și valorile sale în consolă.
3. Creați un obiect care va reprezenta o mașină, cu proprietăți precum marca, modelul, anul fabricației și numărul de kilometri parcurși. Apoi, scrieți o funcție care primește ca argument obiectul mașinii și afișează informații despre aceasta într-un format ușor de citit.
4. Creați un obiect pentru a reprezenta un utilizator al unui site web, cu proprietăți precum numele, adresa de e-mail, numărul de telefon și data nașterii. Apoi, validarea informațiilor introduse de utilizator, utilizând obiectul creat și afișarea unui mesaj de eroare pentru câmpurile introduse greșit.
5. Creați un obiect pentru a reprezenta un meniu de restaurant, cu proprietăți precum numele restaurantului, lista de feluri de mâncare și prețurile acestora. Apoi, scrieți o funcție care primește ca argument obiectul meniului și afișează felurile de mâncare și prețurile lor într-un format ușor de citit.
6. Scrie o funcție care primește ca argument un număr întreg și returnează valoarea sa absolută folosind metoda Math.abs().
7. Scrie o funcție care primește ca argument un număr întreg și returnează un număr întreg aleatoriu între 0 și numărul dat, folosind metoda Math.random() și Math.floor().

8. Creează o funcție care primește două obiecte Date ca argumente și returnează diferența dintre cele două date în zile.

   Exemplu de intrare:

```
const date1 = new Date('2023-02-14');
const date2 = new Date('2023-03-15');
console.log(diffInDays(date1, date2));
```

   Exemplu de ieșire:

```
29
```

9. Creează o funcție care primește un obiect Date ca argument și returnează un șir de caractere care indică ziua săptămânii pentru acea dată.

   Exemplu de intrare:

```
const date = new Date('2023-02-14');
console.log(dayOfWeek(date));
```

   Exemplu de ieșire:

```
Luni
```

10. Creează o funcție care primește un număr întreg ca argument și returnează data de început a anului calendaristic respectiv (1 ianuarie) pentru acel an.

    Exemplu de intrare:

```
console.log(startOfYear(2023));
```

    Exemplu de ieșire:

```
2023-01-01T00:00:00.000Z
```

11. Validați un șir de caractere pentru a verifica dacă respectă un anumit format: Scrieți o funcție care primește un șir de caractere și un model RegExp și verifică dacă șirul respectă formatul specificat de model. De exemplu, puteți verifica dacă un număr de telefon are formatul corect sau dacă un șir conține doar litere și cifre.
12. Înlocuiți un șir de caractere cu alt șir folosind expresii regulate: Scrieți o funcție care primește un șir de caractere, o expresie regulată și un șir de înlocuire. Funcția trebuie să înlocuiască toate

aparițiile șablonului din șirul de intrare cu șirul de înlocuire specificat. Acest lucru poate fi util, de exemplu, pentru a înlocui anumite caractere sau cuvinte într-un text.

13. Extrageți informații relevante dintr-un șir de caractere folosind expresii regulate: Scrieți o funcție care primește un șir de caractere și un model RegExp și extrage informațiile relevante din șirul de intrare. De exemplu, puteți extrage numerele de telefon dintr-un text sau adresele de e-mail. Acest lucru poate fi util în situații în care trebuie să procesați o cantitate mare de date și să extrageți informațiile importante.
14. Sortați un tablou de obiecte după o proprietate comună:
    Scrieți o funcție care primește un tablou de obiecte și o proprietate comună, apoi sortați tabloul în ordine crescătoare sau descrescătoare în funcție de acea proprietate.
15. Selectați un sub-tablou de elemente care îndeplinesc o anumită condiție:
    Scrieți o funcție care primește un tablou și o condiție și returnează un sub-tablou de elemente care îndeplinesc acea condiție. De exemplu, puteți selecta toate numerele pare dintr-un tablou de numere.
16. Calculează suma sau produsul elementelor unui tablou:
    Scrieți o funcție care primește un tablou și calculează suma sau produsul tuturor elementelor. Acest lucru poate fi util pentru calcularea mediei sau pentru obținerea produsului total al unui număr de valori.
17. Găsiți elementul maxim sau minim dintr-un tablou:
    Scrieți o funcție care primește un tablou și returnează cel mai mare sau cel mai mic element din acesta. Acest lucru poate fi util pentru găsirea celei mai mari sau celei mai mici valori dintr-un set de date.
18. Concatenați două sau mai multe tablouri:
    Scrieți o funcție care primește două sau mai multe tablouri și returnează un nou tablou care conține toate elementele din acele tablouri concatenate. Acest lucru poate fi util pentru combinarea mai multor seturi de date într-unul singur.
19. Filtrați elementele unui tablou pe baza unui criteriu: Scrieți o funcție care primește un tablou de numere și un număr minim și returnează un sub-tablou care conține numerele mai mari sau egale cu acel număr minim.

20. Filtrați elementele unui tablou pe baza unui șablon RegExp: Scrieți o funcție care primește un tablou de șiruri de caractere și un șablon RegExp și returnează un sub-tablou care conține doar șirurile de caractere care se potrivesc cu acel șablon.
21. Filtrați elementele unui tablou pe baza proprietăților lor: Scrieți o funcție care primește un tablou de obiecte și un nume de proprietate și returnează un sub-tablou care conține doar obiectele care au acea proprietate setată la o valoare specifică. De exemplu, puteți filtra un tablou de obiecte **student** pe baza notei lor la un test.
22. Scrieți o funcție care primește un tablou de obiecte și un nume de proprietate și returnează un sub-tablou care conține doar obiectele care au acea proprietate setată la o anumită valoare. De exemplu, puteți filtra un tablou de obiecte **utilizator** pe baza rolului utilizatorului.
23. Scrieți o funcție care primește două tablouri de obiecte și un nume de proprietate comună și returnează un sub-tablou din primul tablou care conține obiectele care au o valoare din proprietatea comună care se regăsește în al doilea tablou. De exemplu, puteți filtra un tablou de obiecte **produs** pe baza unui tablou de ID-uri.
24. Scrieți o funcție care primește un tablou de obiecte și un șablon RegExp și returnează un sub-tablou care conține doar obiectele care conțin șiruri de caractere care se potrivesc cu acel șablon. De exemplu, puteți filtra un tablou de obiecte **mesaj** pe baza unui șablon care corespunde cuvintelor cheie.
25. Conversia datelor între obiecte JavaScript și fișiere JSON:
    a. Creează un obiect JavaScript cu mai multe proprietăți, inclusiv proprietăți de tipul array sau obiect.
    b. Utilizează metoda **JSON.stringify()** pentru a converti obiectul JavaScript într-un fișier JSON.
    c. Salvează fișierul JSON și încarcă-l din nou în aplicație utilizând metoda **JSON.parse()**.
    d. Verifică că datele din fișierul JSON au fost convertite corect într-un obiect JavaScript.
26. Validarea datelor JSON:
    a. Creează un șir de caractere JSON valid.
    b. Creează un șir de caractere JSON invalid.
    c. Utilizează metoda **JSON.parse()** pentru a valida cele două șiruri de caractere JSON.
    d. Gestionează eroarea generată de metoda **JSON.parse()** atunci când încerci să validezi șirul de caractere JSON invalid.

27. Transformarea datelor JSON într-un format personalizat:
    a. Creează un șir de caractere JSON care conține date despre un set de utilizatori, inclusiv nume, prenume, dată de naștere etc.
    b. Utilizează metoda **JSON.parse()** cu un al doilea parametru pentru a transforma datele de naștere ale utilizatorilor dintr-un șir de caractere într-un obiect **Date**.
    c. Salvează datele transformate într-un fișier CSV sau XML utilizând o bibliotecă sau un modul adecvat.
28. Creează un obiect cu câteva proprietăți (cel puțin trei) și destructurează-l pentru a crea trei variabile separate.
29. Creează o funcție care primește un obiect cu cel puțin două proprietăți și returnează o propoziție folosind acele proprietăți. Destructurează obiectul în corpul funcției.
30. Extrage dintr-un obiect doar anumite proprietăți și asignă-le la variabile separate.
31. Creează un tablou cu două obiecte și folosește destructurizarea pentru a accesa proprietățile acestora.
32. Extrage proprietățile obiectelor imbricate folosind destructurizarea.
33. Scrieți o funcție care primește ca argumente două liste de numere întregi și returnează o listă care conține toate numerele din cele două liste, fără elementele duplicate. Utilizați operatorul Spread pentru a combina cele două liste și o buclă for pentru a elimina elementele duplicate.
34. Scrieți o funcție care primește ca argumente două obiecte și returnează un nou obiect care combină proprietățile din cele două obiecte. Dacă există proprietăți cu același nume în cele două obiecte, proprietatea din cel de-al doilea obiect va suprascrie proprietatea din primul obiect. Utilizați operatorul Spread pentru a combina cele două obiecte.
35. Scrieți o funcție care primește ca argumente o listă de numere întregi și returnează valoarea maximă din acea listă. Utilizați operatorul Rest pentru a captura argumentele funcției.s

QUIZ

1. Ce metoda de tablou este utilizată pentru a adăuga un element la începutul unui tablou?
   A. shift() B. push() C. unshift() D. pop()
2. Care dintre următoarele metode poate fi utilizată pentru a inversa ordinea elementelor unui tablou?
   A. reverse() B. sort() C. map() D. reduce()
3. Care este scopul metodei forEach() în JavaScript?
   A. Pentru a filtra elementele unui tablou

B. Pentru a aplica o funcție pe fiecare element dintr-un tablou

C. Pentru a găsi primul element care se potrivește cu un criteriu dat

D. Pentru a concatena două tablouri

4. Ce metoda de tablou este utilizată pentru a elimina un element de la începutul unui tablou?

   A. **shift()** B. **push()** C. **unshift()** D. **pop()**

5. Care dintre următoarele metode poate fi utilizată pentru a returna un sub-tablou dintr-un tablou existent?

   A. **slice()** B. **splice()** C. **concat()** D. **reduce()**

6. Ce metoda de tablou este utilizată pentru a elimina un element de la sfârșitul unui tablou?

   A. **shift()** B. **push()** C. **unshift()** D. **pop()**

7. Care dintre următoarele metode poate fi utilizată pentru a filtra elementele unui tablou pe baza unui criteriu specific?

   A. **forEach()** B. **map()** C. **reduce()** D. **filter()**

8. Care dintre următoarele metode poate fi utilizată pentru a verifica dacă un element se găsește într-un tablou?

   A. **includes()** B. **find()** C. **filter()** D. **map()**

9. Care dintre următoarele metode poate fi utilizată pentru a sorta elementele unui tablou?

   A. **sort()** B. **reverse()** C. **slice()** D. **splice()**

10. Ce metoda de tablou este utilizată pentru a adăuga un element la sfârșitul unui tablou?

    A. **shift()** B. **push()** C. **unshift()** D. **pop()**

11. Care dintre următoarele afirmații este corectă despre obiectele din JavaScript?

    a) Obiectele din JavaScript sunt structuri de date statice

    b) Obiectele din JavaScript sunt instanțe ale clasei Object

    c) Obiectele din JavaScript nu pot avea metode

12. Cum se adaugă o nouă pereche cheie-valoare într-un obiect existent în JavaScript?

    a) Cu operatorul "+" între cheie și valoare

    b) Prin atribuirea valorii la o cheie inexistentă din obiect

    c) Prin adăugarea unei proprietăți noi la obiect cu metoda .addProperty()

13. Ce se întâmplă atunci când încercăm să accesez o cheie inexistentă dintr-un obiect?

    a) Se afișează un mesaj de eroare

    b) Se returnează valoarea null

c) Se returnează valoarea undefined

14. Cum se șterge o pereche cheie-valoare dintr-un obiect din JavaScript?
    a) Prin atribuirea valorii null la cheia respectivă
    b) Prin utilizarea metodei .removeProperty() a obiectului
    c) Prin utilizarea operatorului delete cu cheia respective
15. Ce se întâmplă atunci când încercăm să convertim un obiect într-un șir de caractere folosind metoda .toString()?
    a) Se returnează șirul de caractere "[object Object]"
    b) Se afișează un mesaj de eroare
    c) Se returnează o reprezentare string a obiectului, cu toate proprietățile sale
16. Care este scopul obiectului RegExp în JavaScript?
    a) De a oferi funcții matematice complexe
    b) De a testa dacă un șir de caractere se potrivește cu un anumit model
    c) De a oferi metode de sortare a șirurilor de caractere
17. Care dintre următoarele afirmații este corectă despre constructorul RegExp în JavaScript?
    a) Constructorul acceptă un singur argument care reprezintă modelul de potrivire
    b) Constructorul acceptă două argumente: modelul de potrivire și un mod de potrivire
    c) Constructorul nu este utilizat pentru crearea obiectelor RegExp
18. Care este metoda folosită pentru a testa dacă un șir de caractere se potrivește cu un model dat în JavaScript?
    a) .search()
    b) .match()
    c) .test()
19. Care dintre următoarele afirmații este corectă despre metodele exec() și test() ale obiectului RegExp?
    a) Metoda exec() returnează o valoare booleană, iar metoda test() returnează o corespondență găsită.
    b) Metoda exec() returnează o corespondență găsită, iar metoda test() returnează o valoare booleană.
    c) Ambele metode returnează o corespondență găsită.
20. Ce face operatorul Spread în JavaScript?

A. Gruparea unui număr variabil de argumente într-o singură listă.

B. Extinderea unei liste de elemente sau un obiect într-o altă listă sau obiect.

C. Transformarea unei liste de elemente într-un obiect.

21. Ce face operatorul Rest în JavaScript?

A. Extinderea unei liste de elemente sau un obiect într-o altă listă sau obiect.

B. Gruparea unui număr variabil de argumente într-o singură listă.

C. Transformarea unei liste de elemente într-un obiect.

22. Care este rezultatul apelării următoarei funcții?

```
function mergeLists(list1, list2) {
  let mergedList = [...list1, ...list2];
  let uniqueList = [];
  for (let i = 0; i < mergedList.length; i++) {
    if (!uniqueList.includes(mergedList[i])) {
      uniqueList.push(mergedList[i]);
    }
  }
  return uniqueList;
}

console.log(mergeLists([1, 2, 3], [3, 4, 5]));
```

A. [1, 2, 3, 4, 5]

B. [1, 2, 3, 3, 4, 5]

C. [3, 4, 5]

Capitol V. Modelul de obiecte ale documentului (DOM)

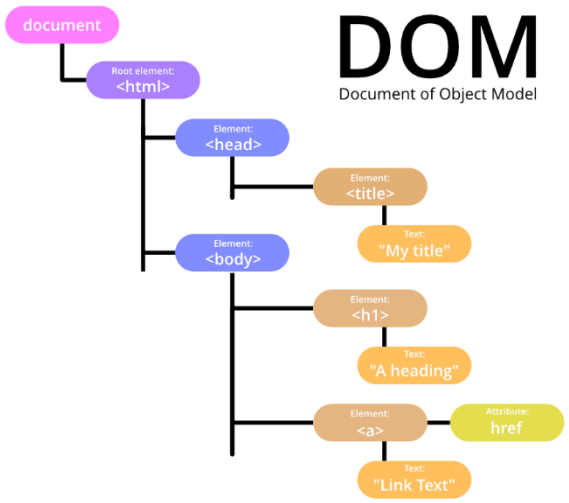
UNITĂȚI DE CONȚINUT

1) Arborele DOM al paginii web.
2) Metode de acces, modificare, adăugare și ștergere a elementelor paginii web.

ABILITĂȚI

- Modificarea elementelor HTML din pagina web.
- Modificarea stilului CSS al elementelor HTML.
- Modificarea valorilor atributelor al elementelor HTML.
- Accesare elementelor după identificator, nume clasa, nume tag, tip selector.
- Modificarea elementelor HTML, stilului CSS al elementelor și a valorilor atributelor elementelor.
- Adăugarea elementelor.
- Ștergerea elementelor.

Arborele DOM (**Document Object Model**) reprezintă o reprezentare structurată a documentului HTML (și a altor documente XML) în formă de arborescență, care permite interacțiunea și manipularea acestuia prin intermediul codului JavaScript.



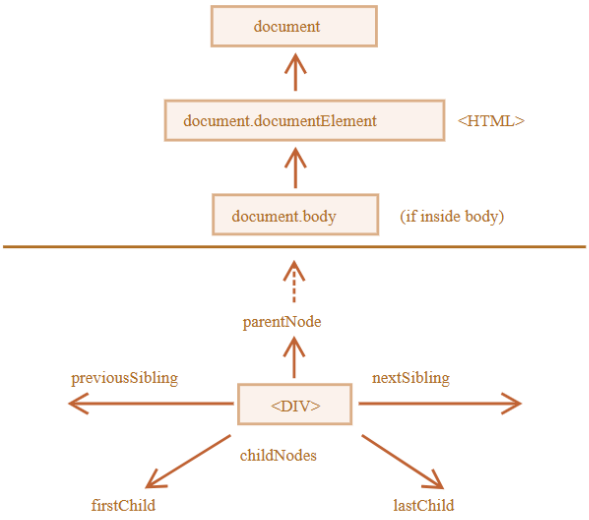
Fiecare element din documentul HTML este reprezentat ca un nod în arborele DOM, iar relațiile dintre aceste noduri sunt reprezentate ca legături între ele. De exemplu, nodul care reprezintă elementul părinte are legături către toți copiii săi.

JavaScript poate accesa și manipula nodurile din arborele DOM folosind metode și proprietăți specifice, ceea ce permite programatorilor să interacționeze cu conținutul HTML dintr-un mod dinamic și interactiv. De exemplu, prin intermediul JavaScript, se poate schimba textul dintr-un element HTML, se poate adăuga sau șterge un element HTML, se poate schimba stilul unui element HTML sau se poate adăuga un eveniment pentru a răspunde la acțiunile utilizatorului.

În general, arborele DOM reprezintă o componentă esențială a dezvoltării web, care permite interacțiunea dintre conținutul HTML și codul JavaScript pentru a crea aplicații web complexe și dinamice.

1. Arborele DOM al paginii web

Arborele DOM al unei pagini web reprezintă o reprezentare structurată în formă de arbore a elementelor HTML ale paginii. Acest arbore este creat în momentul încărcării paginii și poate fi accesat și manipulat prin intermediul codului JavaScript pentru a crea interactivitate și dinamism în pagină.



Structura arborelui DOM reflectă structura HTML a paginii. Fiecare element HTML este reprezentat în arborele DOM ca un nod și este conectat cu alte noduri prin intermediul legăturilor părinte-copil. În acest fel, se formează un arbore ierarhic care reprezintă relațiile dintre elementele HTML ale paginii.

Mai mult decât atât, arborele DOM include și alte tipuri de noduri, cum ar fi nodurile pentru text, comentarii și atribute. De asemenea, elementele HTML pot avea atribuțiile care sunt stocate ca perechi nume-valoare și pot fi accesate prin intermediul codului JavaScript.

Accesarea și manipularea arborelui DOM prin intermediul JavaScript este posibilă prin utilizarea metodelor și proprietăților specifice, cum ar fi **getElementById**, **getElementsByTagName**, **querySelector**, **appendChild**, **removeChild**, **innerHTML** etc. Acestea permit programatorilor să interacționeze cu elementele HTML și să modifice conținutul și stilul acestora, să adauge și să șteargă elemente, să creeze animații și să răspundă la evenimente precum clicuri de mouse sau apăsări de tastatură.

În concluzie, arborele DOM este un element cheie în dezvoltarea web, deoarece permite interacțiunea și manipularea dinamică a conținutului HTML al paginilor web prin intermediul JavaScript.

2. Metode de acces, modificare, adăugare și ștergere a elementelor paginii web

JavaScript oferă o gamă largă de metode pentru a accesa, modifica, adăuga și șterge elemente de pe o pagină web. Iată câteva dintre cele mai comune metode:

Accesarea unui element Pentru a accesa un element din HTML, trebuie să utilizați metoda **getElementById**, care returnează primul element din documentul HTML cu un ID specific.

Exemplu:

```
var element = document.getElementById("id_element");
```

Modificarea unui element După ce ați accesat un element, puteți să îl modificați prin actualizarea valorii proprietăților sale.

Exemplu:

```
var element = document.getElementById("id_element");
element.innerHTML = "Noua valoare";
```

Adăugarea unui element Pentru a adăuga un element nou într-o pagină web, utilizați metoda **createElement** pentru a crea un nou element, apoi utilizați metoda **appendChild** pentru a adăuga elementul nou în document.

Exemplu:

```
var nouElement = document.createElement("div");
nouElement.innerHTML = "Textul elementului nou";
document.body.appendChild(nouElement);
```

Ștergerea unui element Pentru a șterge un element dintr-o pagină web, utilizați metoda **removeChild** pentru a elimina elementul dorit.

Exemplu:

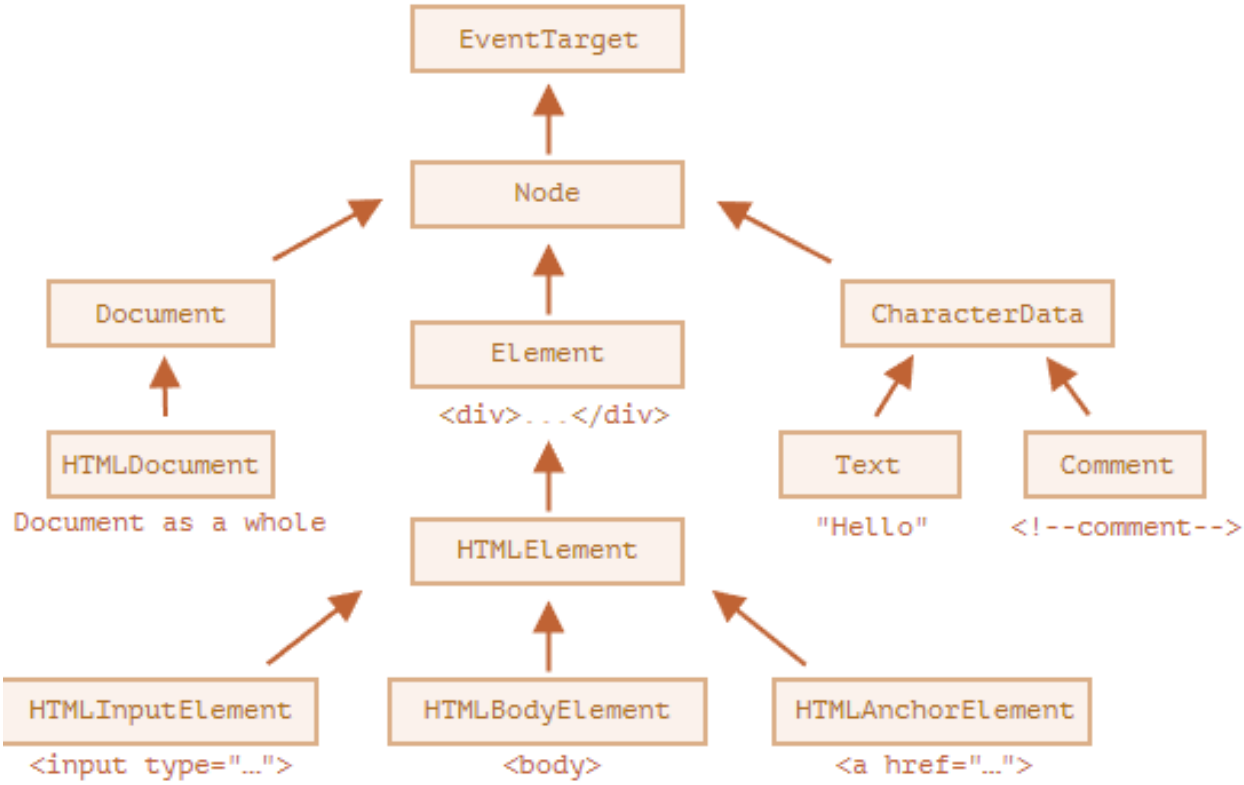
```
var element = document.getElementById("id_element");
element.parentNode.removeChild(element);
```

Acestea sunt câteva dintre cele mai comune metode pentru a accesa, modifica, adăuga și șterge elemente pe o pagină web folosind JavaScript.

În JavaScript, puteți accesa documentul DOM (Modelul Obiect al Documentului) și toate elementele sale folosind diferite metode. Iată câteva dintre cele mai comune metode de acces ale documentului DOM:

1. **document.getElementById()** - această metodă permite accesarea unui element HTML după ID-ul său. De exemplu, dacă există un element cu ID-ul "titlu", puteți accesa acel element folosind **document.getElementById("titlu")**.
2. **document.getElementsByTagName()** - această metodă permite accesarea tuturor elementelor HTML cu un anumit nume de tag, cum ar fi "p", "div" sau "a". De exemplu, puteți accesa toate elementele "p" din document folosind **document.getElementsByTagName("p")**.
3. **document.getElementsByClassName()** - această metodă permite accesarea tuturor elementelor HTML care au o anumită clasă. De exemplu, puteți accesa toate elementele cu clasa "meniu" folosind **document.getElementsByClassName("meniu")**.
4. **document.querySelector()** - această metodă permite accesarea primului element HTML care corespunde unui selector CSS dat. De exemplu, puteți accesa primul element "h1" din document folosind **document.querySelector("h1")**.
5. **document.querySelectorAll()** - această metodă permite accesarea tuturor elementelor HTML care corespund unui selector CSS dat. De exemplu, puteți accesa toate elementele "p" care se află într-un div cu clasa "container" folosind **document.querySelectorAll(".container p")**.

Acestea sunt câteva dintre cele mai comune metode de acces ale documentului DOM folosind JavaScript.



Proprietățile nodului (Node) sunt utilizate în JavaScript pentru a accesa, modifica și crea elemente HTML într-o pagină web. Iată trei dintre cele mai importante proprietăți ale nodului:

1. **type** - Această proprietate indică tipul de nod. Există mai multe tipuri de noduri, cum ar fi elementul (element), textul (text) sau comentariul (comment). De exemplu, dacă aveți un nod care reprezintă un element HTML, proprietatea **type** va fi setată la "element".
2. **tag** - Această proprietate este utilizată pentru a accesa numele etichetei (tag) a unui element HTML. De exemplu, dacă aveți un nod care reprezintă un element HTML "div", proprietatea **tag** va fi setată la "div".
3. **contents** - Această proprietate conține conținutul unui nod, cum ar fi textul, valorile atributelor sau alte noduri. De exemplu, dacă aveți un nod care reprezintă un element HTML "p" cu textul "Exemplu de text", proprietatea **contents** va conține textul "Exemplu de text".

Acestea sunt trei dintre cele mai importante proprietăți ale nodului în JavaScript și sunt utilizate frecvent în procesul de accesare, modificare și creare a elementelor HTML într-o pagină web.

Cele trei proprietăți (**innerHTML**, **innerText** și **textContent**) sunt utilizate pentru a obține sau a seta conținutul unui element HTML în JavaScript, dar există unele diferențe subtile între ele:

1. **innerHTML** este o proprietate care poate fi accesată și modificată în JavaScript pentru a obține sau a seta conținutul HTML al unui element. Atunci când setați **innerHTML** la o anumită valoare, acea valoare va fi interpretată ca HTML. De exemplu, dacă aveți un element cu id-ul "titlu" și doriți să îl modificați astfel încât să conțină un paragraf HTML, puteți folosi următorul cod:
2. **innerText** este o proprietate utilizată pentru a accesa textul din interiorul unui element. De exemplu, dacă aveți un element cu id-ul "paragraf" și doriți să obțineți textul din interiorul lui, puteți folosi următorul cod:
3. **textContent** este o proprietate utilizată pentru a accesa sau a seta textul brut din interiorul unui element. Acesta include orice etichete, dar nu interpretează aceste etichete ca HTML. De exemplu, dacă aveți un element cu id-ul "paragraf" și doriți să obțineți textul brut din interiorul lui, puteți folosi următorul cod:

În general, **innerHTML** este utilizat atunci când trebuie să modificați sau să adăugați conținut HTML la un element, iar **innerText** și **textContent** sunt utilizate atunci când trebuie să modificați sau să accesați textul din interiorul unui element.

1. Atribute HTML

In JavaScript, **getAttribute()** este o metodă pe care o puteți folosi pentru a obține valoarea unui atribut specific al unui element HTML. Această metodă acceptă un singur argument, care reprezintă numele atributului căutat.

Sintaxa pentru utilizarea metodei getAttribute() este următoarea:

```
element.getAttribute(name);
```

unde **element** este elementul HTML pentru care se dorește obținerea valorii atributului, iar **name** este numele atributului căutat.

De exemplu, dacă aveți următorul element HTML:

```
<a href="https://www.example.com">Click here</a>
```

Puteți utiliza **getAttribute()** pentru a obține valoarea atributului **href** astfel:

```
var link = document.querySelector('a');
var href = link.getAttribute('href');
console.log(href); // "https://www.example.com"
```

În acest exemplu, **querySelector()** este utilizat pentru a selecta primul element **<a>** din document, iar **getAttribute()** este utilizat pentru a obține valoarea atributului **href** al acestuia.

In JavaScript, **setAttribute()** este o metodă pe care o puteți folosi pentru a seta valoarea unui atribut specific al unui element HTML. Această metodă acceptă doi parametri: numele atributului și valoarea atributului.

Sintaxa pentru utilizarea metodei **setAttribute()** este următoarea:

```
element.setAttribute(name, value);
```

unde **element** este elementul HTML pentru care se dorește setarea valorii atributului, **name** este numele atributului și **value** este valoarea atributului.

De exemplu, dacă aveți un element **<img>** fără atributul **alt** și doriți să setați valoarea acestuia la "Imaginea mea", puteți utiliza **setAttribute()** astfel:

```
var img = document.querySelector('img');
img.setAttribute('alt', 'Imaginea mea');
```

După această instrucțiune, codul HTML va arăta astfel:

```
<img src="path/to/image.jpg" alt="Imaginea mea">
```

În acest exemplu, **querySelector()** este utilizat pentru a selecta primul element **<img>** din document, iar **setAttribute()** este utilizat pentru a seta valoarea atributului **alt** al acestuia la "Imaginea mea".

2. Creare de elemente și inserții

In JavaScript, **createElement()** este o metodă pe care o puteți utiliza pentru a crea un element HTML nou. Această metodă acceptă un singur argument, care reprezintă numele elementului pe care doriți să-l creați.

Sintaxa pentru utilizarea metodei **createElement()** este următoarea:

```
document.createElement(tagName);
```

unde **tagName** este numele elementului HTML pe care doriți să îl creați.

De exemplu, dacă doriți să creați un element ** ** nou în document, puteți utiliza **createElement()** astfel:

```
var div = document.createElement('div');
```

După această instrucțiune, variabila **div** va fi un obiect de tipul **Element** care reprezintă noul element ** ** creat.

Puteți utiliza apoi alte metode, precum **appendChild()** sau **insertBefore()**, pentru a adăuga noul element în document sau pentru a-l insera într-un element existent.

De exemplu, pentru a adăuga noul element ** ** creat anterior în interiorul unui element cu clasa "container", puteți utiliza următorul cod:

```
var container = document.querySelector('.container');
container.appendChild(div);
```

În acest exemplu, **querySelector()** este utilizat pentru a selecta primul element din document cu clasa "container", iar **appendChild()** este utilizat pentru a adăuga noul element ** ** în interiorul acestuia.

De exemplu:

```
// Crearea unui element <h1>
var heading = document.createElement('h1');

// Setarea textului pentru elementul <h1>
heading.textContent = 'Bun venit pe site-ul meu!';

// Adăugarea elementului <h1> în interiorul elementului <body>
document.body.appendChild(heading);
```

În acest exemplu, metoda **createElement()** este utilizată pentru a crea un nou element **<h1>**. Apoi, textul "Bun venit pe site-ul meu!" este adăugat în interiorul elementului nou creat folosind proprietatea **textContent**. În cele din urmă, elementul **<h1>** este adăugat în interiorul elementului **<body>** folosind metoda **appendChild()**. Astfel, elementul **<h1>** cu textul "Bun venit pe site-ul meu!" va fi afișat pe pagina web atunci când acest script JavaScript este rulat.

In JavaScript, **append()** este o metodă pe care o puteți utiliza pentru a adăuga unul sau mai multe elemente la sfârșitul unui alt element existent. Această metodă acceptă unul sau mai multe argumente, care pot fi obiecte Element sau string-uri.

Sintaxa pentru utilizarea metodei **append()** este următoarea:

```
element.append(child1, child2, ..., childN);
```

unde **element** este elementul HTML la care doriți să adăugați noi elemente, iar **child1**, **child2**, ..., **childN** sunt elementele sau string-urile pe care doriți să le adăugați.

De exemplu, dacă aveți un element **<ul>** și doriți să adăugați un element **<li>** nou cu textul "Element nou" la sfârșitul acestuia, puteți utiliza **append()** astfel:

```
var ul = document.querySelector('ul');
var li = document.createElement('li');
li.textContent = 'Element nou';
ul.append(li);
```

După această instrucțiune, elementul nou creat **<li>** cu textul "Element nou" va fi adăugat la sfârșitul elementului **<ul>** existent.

Metoda **append()** este foarte flexibilă și poate fi folosită pentru a adăuga mai multe elemente sau string-uri la un element existent. De asemenea, puteți utiliza această metodă pentru a adăuga elemente în interiorul altor elemente, precum ** **, **<section>** sau alte elemente.

In JavaScript, **prepend()** este o metodă pe care o puteți utiliza pentru a adăuga unul sau mai multe elemente la începutul unui alt element existent. Această metodă acceptă unul sau mai multe argumente, care pot fi obiecte Element sau string-uri.

Sintaxa pentru utilizarea metodei **prepend()** este următoarea:

```
element.prepend(child1, child2, ..., childN);
```

unde **element** este elementul HTML la care doriți să adăugați noi elemente, iar **child1**, **child2**, ..., **childN** sunt elementele sau string-urile pe care doriți să le adăugați.

De exemplu, dacă aveți un element **<ul>** și doriți să adăugați un element **<li>** nou cu textul "Element nou" la începutul acestuia, puteți utiliza **prepend()** astfel:

```
var ul = document.querySelector('ul');
var li = document.createElement('li');
li.textContent = 'Element nou';
ul.prepend(li);
```

După această instrucțiune, elementul nou creat **<li>** cu textul "Element nou" va fi adăugat la începutul elementului **<ul>** existent.

Metoda **prepend()** este foarte flexibilă și poate fi folosită pentru a adăuga mai multe elemente sau string-uri la un element existent. De asemenea, puteți utiliza această metodă pentru a adăuga elemente în interiorul altor elemente, precum ** **, **<section>** sau alte elemente.

In JavaScript, **before()** este o metodă pe care o puteți utiliza pentru a adăuga unul sau mai multe elemente înaintea unui alt element existent. Această metodă acceptă unul sau mai multe argumente, care pot fi obiecte Element sau string-uri.

Sintaxa pentru utilizarea metodei **before()** este următoarea:

```
existingElement.before(newElement1, newElement2, ..., newElementN);
```

unde **existingElement** este elementul HTML înaintea căruia doriți să adăugați noi elemente, iar **newElement1**, **newElement2**, ..., **newElementN** sunt elementele sau string-urile pe care doriți să le adăugați.

De exemplu, dacă aveți un element **<p>** și doriți să adăugați un element **<h1>** nou cu textul "Titlu nou" înaintea acestuia, puteți utiliza **before()** astfel:

```
var p = document.querySelector('p');
var h1 = document.createElement('h1');
h1.textContent = 'Titlu nou';
p.before(h1);
```

După această instrucțiune, elementul nou creat **<h1>** cu textul "Titlu nou" va fi adăugat înaintea elementului **<p>** existent.

Metoda **before()** este foarte flexibilă și poate fi folosită pentru a adăuga mai multe elemente sau string-uri înaintea unui element existent. De asemenea, puteți utiliza această metodă pentru a adăuga elemente înaintea altor elemente, precum ** **, **<section>** sau alte elemente.

In JavaScript, **after()** este o metodă pe care o puteți utiliza pentru a adăuga unul sau mai multe elemente după un alt element existent. Această metodă acceptă unul sau mai multe argumente, care pot fi obiecte Element sau string-uri.

Sintaxa pentru utilizarea metodei **after()** este următoarea:

```
existingElement.after(newElement1, newElement2, ..., newElementN);
```

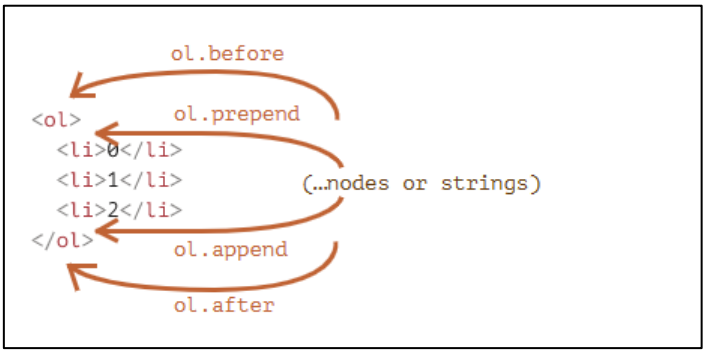
unde **existingElement** este elementul HTML după care doriți să adăugați noi elemente, iar **newElement1**, **newElement2**, ..., **newElementN** sunt elementele sau string-urile pe care doriți să le adăugați.

De exemplu, dacă aveți un element **<p>** și doriți să adăugați un element **<h1>** nou cu textul "Titlu nou" după acesta, puteți utiliza **after()** astfel:

```
var p = document.querySelector('p');
var h1 = document.createElement('h1');
h1.textContent = 'Titlu nou';
p.after(h1);
```

După această instrucțiune, elementul nou creat **<h1>** cu textul "Titlu nou" va fi adăugat după elementul **<p>** existent.

Metoda **after()** este foarte flexibilă și poate fi folosită pentru a adăuga mai multe elemente sau string-uri după un element existent. De asemenea, puteți utiliza această metodă pentru a adăuga elemente după alte elemente, precum ** **, **<section>** sau alte elemente.



În JavaScript, **replaceWith()** este o metodă pe care o puteți utiliza pentru a înlocui un element HTML existent cu un alt element HTML. Această metodă acceptă unul sau mai multe argumente, care pot fi obiecte Element sau string-uri.

Sintaxa pentru utilizarea metodei **replaceWith()** este următoarea:

În JavaScript, **remove()** este o metodă pe care o puteți utiliza pentru a elimina un element HTML din pagină. Această metodă nu acceptă niciun argument.

Sintaxa pentru utilizarea metodei **remove()** este următoarea:

```
existingElement.remove();
```

unde existingElement este elementul HTML pe care doriți să îl eliminați.

De exemplu, dacă aveți un element <p> și doriți să îl eliminați din pagină, puteți utiliza remove() astfel:

```
var p = document.querySelector('p');
p.remove();
```

După această instrucțiune, elementul **<p>** va fi eliminat din pagină.

Metoda **remove()** este foarte utilă atunci când doriți să eliminați un element din pagină pentru a face loc altor elemente. De asemenea, puteți utiliza această metodă în combinație cu alte metode pentru a muta sau înlocui elemente într-o pagină web.

3. Stilizare în Javascript

În JavaScript, **style** este o proprietate pe care o puteți utiliza pentru a accesa și modifica stilurile CSS ale unui element HTML. Această proprietate permite modificarea stilurilor CSS prin intermediul JavaScript-ului.

Sintaxa pentru utilizarea proprietății **style** este următoarea:

```
element.style.property = "value";
```

unde **element** este elementul HTML pe care doriți să îi modificați stilurile, **property** este numele proprietății CSS pe care doriți să o modificați (de exemplu, **color**, **font-size**, **background-color**, etc.), iar **value** este valoarea pe care doriți să o setați pentru această proprietate.

De exemplu, dacă aveți un element **<p>** și doriți să îi setați culoarea textului la roșu, puteți utiliza **style** astfel:

```
var p = document.querySelector('p');
p.style.color = "red";
```

După această instrucțiune, textul din elementul **<p>** va fi de culoare roșie.

Proprietatea **style** este foarte utilă atunci când doriți să modificați stilurile CSS ale unui element în funcție de acțiunile utilizatorului sau în timp real. De asemenea, puteți utiliza această proprietate pentru a anima sau a schimba stilurile elementelor într-o pagină web.

ClassName - **className** este o proprietate a obiectelor Element care returnează sau setează clasa CSS a elementului. Prin utilizarea acestei proprietăți, puteți obține sau seta clasa CSS a unui element din codul JavaScript.

```
element.className = "class-name";
```

unde **element** este elementul HTML pe care doriți să îi setați clasa CSS, iar **class-name** este numele clasei CSS pe care doriți să o setați.

De exemplu, dacă aveți un element **<p>** și doriți să îi setați clasa CSS la "red-text", puteți utiliza **className** astfel:

```
var p = document.querySelector('p');
p.className = "red-text";
```

După această instrucțiune, elementul **<p>** va avea clasa CSS "red-text".

Proprietatea **className** este utilă atunci când doriți să modificați clasa CSS a unui element din codul JavaScript. Puteți utiliza această proprietate în combinație cu alte proprietăți și metode pentru a crea și modifica dinamic stilurile elementelor dintr-o pagină web.

Proprietatea **classList** din JavaScript este o proprietate a obiectului Element care oferă o interfață pentru a lucra cu clasele CSS ale elementului respectiv. Această proprietate este disponibilă pe toate elementele HTML, inclusiv pe elementele create dinamic prin intermediul metodei **createElement()**. **classList** este de tip **DOMTokenList**, care este o colecție de obiecte **DOMToken** reprezentând clasele CSS ale elementului HTML. **DOMToken** este un obiect simplu care conține un șir de caractere reprezentând o singură clasă CSS. Metodele disponibile pe **classList** permit adăugarea, eliminarea, comutarea și verificarea claselor CSS.

Proprietatea **classList** din JavaScript este o proprietate a obiectului Element care oferă o interfață pentru a lucra cu clasele CSS ale elementului respectiv. Această proprietate este disponibilă pe toate elementele HTML, inclusiv pe elementele create dinamic prin intermediul metodei **createElement()**. **classList** este de tip **DOMTokenList**, care este o colecție de obiecte **DOMToken** reprezentând clasele CSS ale elementului HTML. **DOMToken** este un obiect simplu care conține un șir de caractere reprezentând o singură clasă CSS. Metodele disponibile pe **classList** permit adăugarea, eliminarea, comutarea și verificarea claselor CSS.

1. **add()**: Această metodă adaugă o clasă CSS la elementul HTML, fără a afecta clasele CSS existente ale elementului.
2. **remove()**: Această metodă elimină o clasă CSS din elementul HTML.
3. **toggle()**: Această metodă adaugă o clasă CSS la elementul HTML dacă nu există și o elimină dacă deja există.
4. **contains()**: Această metodă returnează **true** dacă elementul HTML conține clasa CSS specificată și **false** în caz contrar.

- **add**(): Această metodă adaugă o clasă CSS la elementul HTML, fără a afecta clasele CSS existente ale elementului.

Sintaxa pentru utilizarea metodei add() este următoarea:

```
element.classList.add("class-name");
```

unde **element** este elementul HTML căruia doriți să îi adăugați o clasă CSS, iar **class-name** este numele clasei CSS pe care doriți să o adăugați.

De exemplu, pentru a adăuga clasa CSS "red-text" la un element **<p>**, puteți utiliza metoda **add()** astfel:

```
var p = document.querySelector('p');
p.classList.add("red-text");
```

- **remove()**: Această metodă elimină o clasă CSS din elementul HTML.

Sintaxa pentru utilizarea metodei **remove()** este următoarea:

```
element.classList.remove("class-name");
```

unde **element** este elementul HTML căruia doriți să îi eliminați o clasă CSS, iar **class-name** este numele clasei CSS pe care doriți să o eliminați.

De exemplu, pentru a elimina clasa CSS "red-text" de la un element **<p>**, puteți utiliza metoda **remove()** astfel:

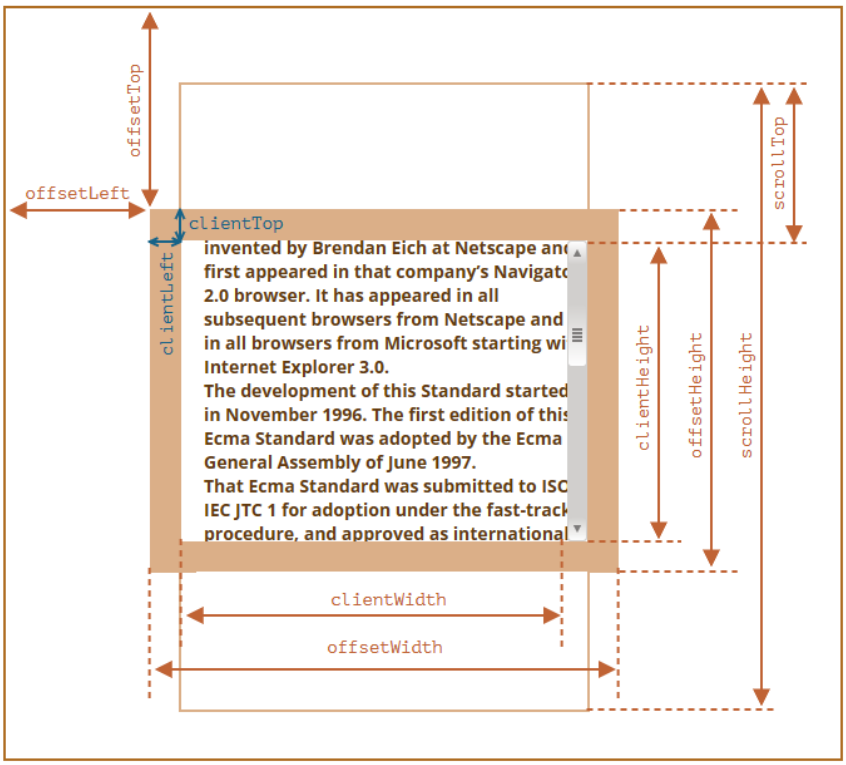
```
var p = document.querySelector('p');
p.classList.remove("red-text");
```

Proprietatea **classList** din JavaScript este o proprietate a obiectului Element care oferă o interfață pentru a lucra cu clasele CSS ale elementului respectiv. Această proprietate este disponibilă pe toate elementele HTML, inclusiv pe elementele create dinamic prin intermediul metodei **createElement()**.

4. Mărimea și poziția elementelor din documentul HTML

Proprietatea **offsetWidth** a obiectului Element este folosită pentru a obține lățimea efectivă a elementului, inclusiv marginile, bordurile și bara de derulare (în cazul în care este afișată). **offsetHeight** este folosit pentru a obține înălțimea efectivă a elementului, cu aceleași proprietăți de calcul ca **offsetWidth**. Aceste proprietăți sunt utile pentru a determina mărimea exactă a unui element și pentru a îl poziționa în mod corespunzător în documentul HTML.

Proprietatea **scrollWidth** este folosită pentru a obține lățimea conținutului efectiv al elementului, astfel încât să se poată determina dacă există o bară de derulare orizontală. **scrollHeight** este folosit pentru a obține înălțimea conținutului efectiv al elementului, astfel încât să se poată determina dacă există o bară de derulare verticală. Aceste proprietăți sunt utile pentru a determina dacă un element poate fi derulat și pentru a obține dimensiunea reală a conținutului său.



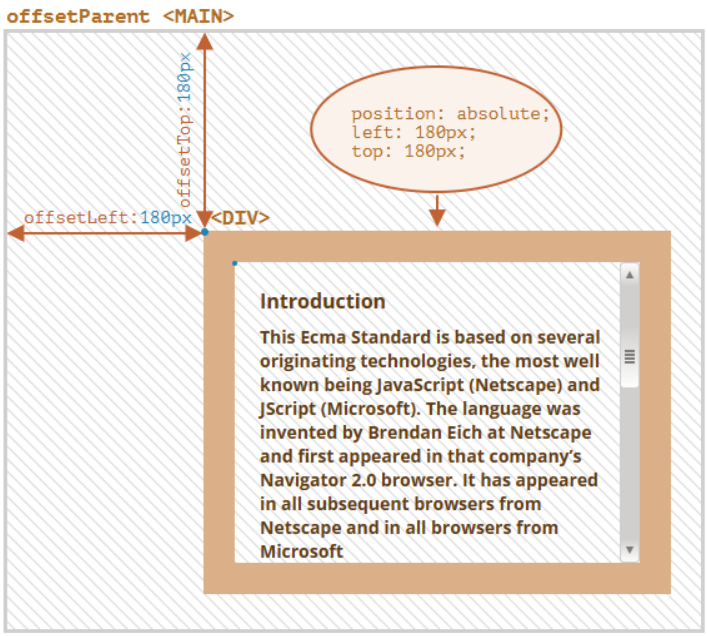
Pentru a accesa poziția de derulare a unui element, se pot folosi proprietățile **scrollLeft** și **scrollTop**. **scrollLeft** indică cât de mult s-a derulat orizontal elementul, iar **scrollTop** indică cât de mult s-a derulat vertical. Aceste proprietăți sunt utile pentru a determina poziția curentă a elementului în cadrul conținutului său și pentru a implementa funcționalitatea de derulare programatică a elementelor.

Aceste proprietăți sunt utile în dezvoltarea de interfețe web interactive, în care se poate accesa și manipula poziția și dimensiunea elementelor în timpul rulării paginii.

Proprietățile **offsetParent**, **offsetLeft** și **offsetTop** sunt utilizate pentru a accesa și manipula poziția unui element în raport cu părintele său de referință.

Proprietatea **offsetParent** reprezintă elementul părinte cel mai apropiat în arborele DOM care are o poziție setată explicit (de exemplu, care nu este **position: static**). Dacă nu există astfel de elemente,

atunci **offsetParent** va fi **body** sau **html**. Această proprietate este utilă pentru a determina poziția relativă a unui element față de un alt element din arborele DOM.



Proprietățile **offsetLeft** și **offsetTop** reprezintă distanța dintre marginea stângă /superioară a elementului și marginea stângă/superioară a elementului părinte de referință (adică **offsetParent**). Aceste proprietăți sunt utile pentru a poziționa un element relativ la părintele său de referință.

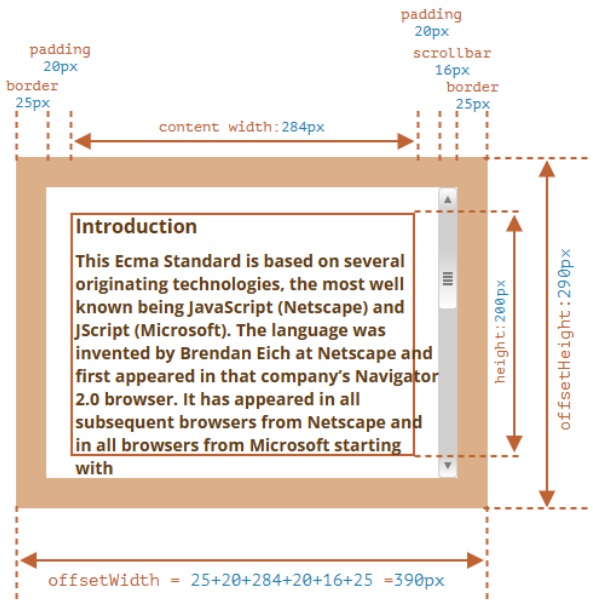
De exemplu, dacă avem un element **div** poziționat relativ la un alt element **p**, putem accesa **offsetParent** pentru a accesa elementul **p** și **offsetLeft** și **offsetTop** pentru a determina poziția relativă a elementului **div** față de **p**.

```
const div = document.querySelector('div');
console.log(div.offsetParent); // <p> elementul de mai sus
console.log(div.offsetLeft); // 50
console.log(div.offsetTop); // 50
```

HTML

```
<p style="position: relative;">
  Aceasta este o propozitie.
  <div style="position: absolute; left: 50px; top: 50px;">Aceasta este o diviziune.</div>
</p>
```

În general, aceste proprietăți sunt utile pentru a manipula poziția relativă a elementelor în cadrul documentului HTML și pentru a implementa efecte de animație și interacțiune personalizate.



Proprietățile **offsetWidth** și **offsetHeight** în JavaScript reprezintă dimensiunile unui element inclusiv marginile, chenarele și bara de derulare orizontală și verticală (dacă acestea sunt prezente). Aceste proprietăți sunt utile pentru a determina dimensiunea totală a unui element în interiorul arborelui DOM.

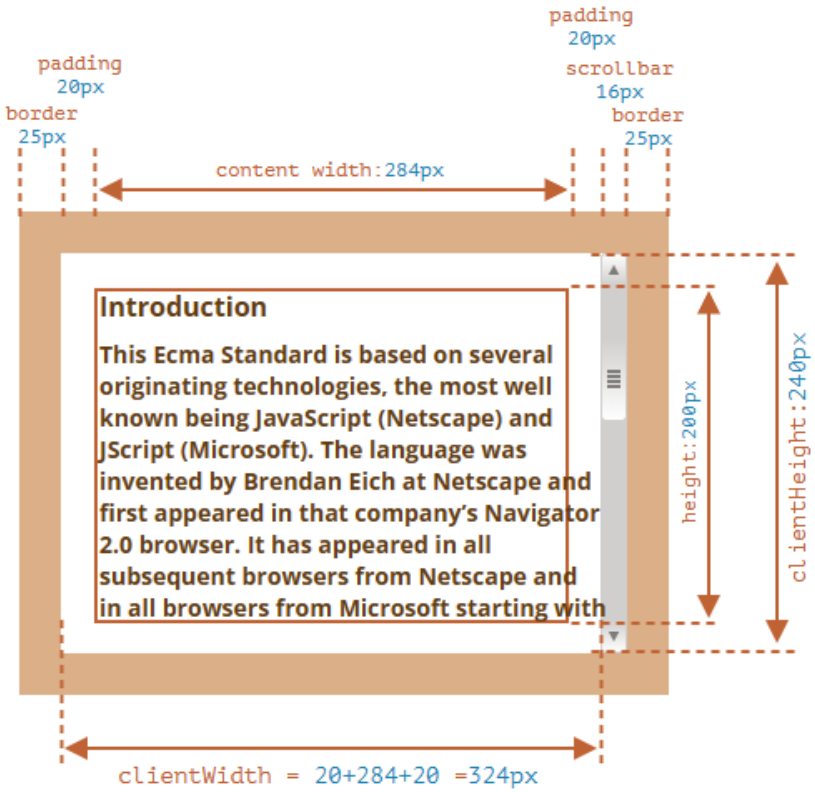
De exemplu, dacă avem un element ** ** cu o lățime și o înălțime fixă, putem accesa proprietățile **offsetWidth** și **offsetHeight** pentru a determina dimensiunea totală a elementului.

În general, aceste proprietăți sunt utile pentru a determina dimensiunea totală a unui element și pentru a-l poziționa corect în cadrul documentului HTML. De asemenea, acestea pot fi utile pentru a implementa efecte de animație și interacțiune personalizate.

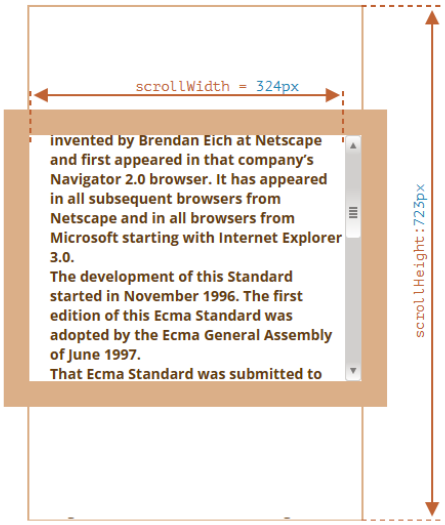
Proprietățile **clientTop** și **clientLeft** în JavaScript reprezintă dimensiunile chenarului superior și stâng al unui element, adică grosimea marginii superioare și a marginii stângi. Aceste proprietăți nu includ marginile interioare și exterioare ale elementului și pot fi utilizate pentru a determina poziția absolută a unui element în raport cu documentul HTML.

De exemplu, dacă avem un element ** ** cu o bordură și o margine, putem accesa proprietățile **clientTop** și **clientLeft** pentru a determina grosimea marginii superioare și a marginii stângi.

Proprietățile **clientWidth** și **clientHeight** în JavaScript reprezintă dimensiunile interioare ale unui element, adică dimensiunea totală a conținutului elementului, inclusiv spațiile pentru margini interioare, dar nu include marginile exterioare sau bara de derulare, dacă există. Aceste proprietăți se referă la dimensiunea vizibilă a elementului în fereastra browserului.



De exemplu, dacă avem un element ** ** cu o lățime și o înălțime definite, dar fără margini sau padding, putem accesa proprietățile **clientWidth** și **clientHeight** pentru a obține dimensiunile reale ale conținutului elementului.



Proprietățile **scrollWidth** și **scrollHeight** în JavaScript reprezintă dimensiunile totale ale unui element, inclusiv conținutul ascuns prin bară de derulare, dacă există. Aceste proprietăți se referă la dimensiunea totală a elementului, inclusiv spațiile pentru margini, dar nu include marginile exterioare.

De exemplu, dacă avem un element ** ** cu un conținut mai mare decât dimensiunea sa vizibilă, putem accesa proprietățile **scrollWidth** și **scrollHeight** pentru a obține dimensiunea totală a elementului, inclusiv zona ascunsă.

Proprietățile **scrollLeft** și **scrollTop** în JavaScript reprezintă distanța în pixeli cu care un element este derulat în orizontală (pentru **scrollLeft**) sau în verticală (pentru **scrollTop**). Aceste proprietăți sunt utilizate pentru a obține sau seta poziția de derulare a conținutului unui element.

De exemplu, dacă avem un element cu bară de derulare orizontală, putem accesa proprietatea **scrollLeft** pentru a obține sau seta poziția curentă a barei de derulare.

4. Dimensiunile ferestrelor și derularea

Dimensiunile ferestrelor și derularea în JavaScript se referă la măsurarea dimensiunilor și alocării zonei de afișare a ferestrei browserului, precum și la gestionarea derulării acesteia. Dimensiunile ferestrei se pot schimba în funcție de dimensiunile ecranului sau de opțiunile utilizatorului, cum ar fi redimensionarea manuală a ferestrei sau trecerea în modul ecran complet. Scrolling-ul se referă la posibilitatea de a derula pagina web în sus sau în jos, pentru a afișa conținutul care nu este vizibil în zona de afișare a ferestrei. Acest lucru se poate face prin intermediul băii de derulare a ferestrei sau prin intermediul unor elemente specifice, cum ar fi bara de derulare verticală.

Pentru a obține dimensiunile curente ale ferestrei browserului, putem folosi proprietățile **window.innerWidth** și **window.innerHeight**. Aceste proprietăți returnează dimensiunea ferestrei exprimată în pixeli. Dacă sunt prezente bare de derulare orizontale sau verticale, acestea sunt incluse în dimensiunile raportate de aceste proprietăți.

De exemplu, pentru a afișa dimensiunile ferestrei curente în consolă, putem folosi următorul cod:

```
console.log(`Window width: ${window.innerWidth}`);
console.log(`Window height: ${window.innerHeight}`);
```

Rezultatul afișat în consolă arată lățimea și înălțimea ferestrei curente în pixeli.

Este important de menționat că aceste proprietăți pot fi afectate de redimensionarea ferestrei sau de prezența sau absența unor elemente ale interfeței utilizatorului, cum ar fi barele de derulare sau bara de adrese.

Pentru a obține valoarea curentă a deplasării verticale sau orizontale a băii de derulare în JavaScript, putem utiliza proprietățile **window.scrollX** și **window.scrollY**. Aceste proprietăți returnează numărul de pixeli cu care băia de derulare este deplasată în direcția orizontală și verticală, respectiv.

De exemplu, pentru a afișa valoarea curentă a deplasării verticale și orizontale în consolă, putem folosi următorul cod:

```
console.log(`Vertical scroll: ${window.scrollY}`);
console.log(`Horizontal scroll: ${window.scrollX}`);
```

Valoarea returnată de aceste proprietăți este un număr întreg pozitiv sau negativ, care reprezintă numărul de pixeli cu care băia de derulare a fost deplasată într-o anumită direcție.

Este important de menționat că aceste proprietăți pot fi afectate de tipul de dispozitiv utilizat și de modul în care este afișată pagina web. De exemplu, pe un dispozitiv mobil sau pe o pagină web care se adaptează la dimensiunea ecranului, băia de derulare poate fi afișată într-un mod diferit față de un ecran de calculator tradițional.

5. Scrolling

Scrolling-ul este o acțiune comună pe paginile web, iar JavaScript oferă câteva metode pentru a efectua această acțiune. Iată câteva dintre cele mai utilizate metode pentru scrolling în JavaScript:

1. **scrollTo(x, y)** - Această metodă este utilizată pentru a deplasa băia de derulare la o anumită poziție. Argumentele x și y sunt coordonatele orizontale și, respectiv, verticale la care trebuie să se deplaseze băia de derulare. De exemplu, pentru a deplasa băia de derulare la poziția (100, 500), putem utiliza următorul cod:

```
window.scrollTo(100, 500);
```

2. scrollBy(x, y) - Această metodă este utilizată pentru a deplasa băia de derulare cu un anumit număr de pixeli în direcția orizontală și/sau verticală. Argumentele x și y sunt numărul de pixeli cu care trebuie să se deplaseze băia de derulare în direcția orizontală și, respectiv, verticală. De exemplu, pentru a deplasa băia de derulare cu 100 de pixeli în direcția orizontală și 500 de pixeli în direcția verticală, putem utiliza următorul cod:

```
window.scrollBy(100, 500);
```

3. scrollIntoView() - Această metodă este utilizată pentru a deplasa băia de derulare astfel încât elementul curent să fie vizibil în zona vizibilă a paginii. Această metodă nu are argumente și poate fi apelată pe un element HTML. De exemplu, pentru a face elementul cu id-ul "my-element" vizibil în zona vizibilă a paginii, putem utiliza următorul cod:

```
document.getElementById("my-element").scrollIntoView();
```

6. Coordonate

În JavaScript, coordonatele sunt utilizate pentru a descrie poziția unui element pe pagină. Există mai multe proprietăți pe care le putem utiliza pentru a obține coordonatele unui element.

1. **offsetLeft** și **offsetTop** - Aceste proprietăți oferă coordonatele elementului în raport cu elementul părinte. De exemplu, dacă avem un element **div** cu un **offsetLeft** de 100 și un **offsetTop** de 50, acest lucru înseamnă că elementul **div** este poziționat la 100 pixeli în direcția orizontală și 50 de pixeli în direcția verticală față de elementul părinte.
2. **offsetWidth** și **offsetHeight** - Aceste proprietăți reprezintă lățimea și înălțimea elementului, inclusiv margini, borduri și padding-uri.
3. **clientLeft** și **clientTop** - Aceste proprietăți reprezintă distanța dintre marginea stângă și superioară a elementului și marginea stângă și superioară a zonei vizibile a paginii.
4. **clientWidth** și **clientHeight** - Aceste proprietăți reprezintă lățimea și înălțimea zonei vizibile a elementului. Aceste proprietăți nu includ margini, borduri și padding-uri.
5. **scrollLeft** și **scrollTop** - Aceste proprietăți reprezintă distanța de deplasare a elementului în direcția orizontală și, respectiv, verticală. De exemplu, dacă un element are un **scrollLeft** de 100 și un **scrollTop** de 50, acest lucru înseamnă că elementul a fost deplasat la 100 pixeli în direcția orizontală și 50 de pixeli în direcția verticală.

Coordonatele sunt utile atunci când trebuie să poziționăm un element sau să efectuăm alte acțiuni în funcție de poziția sa pe pagină.

3. Promise

În programarea JavaScript, Promise este un obiect care reprezintă un rezultat care nu este încă disponibil, dar care ar putea fi în viitor. Promise este o modalitate de a face programarea asincronă mai ușoară și mai clară de citit.

Un Promise poate fi într-unul din cele trei stări:

- Pending: Promise este încă în așteptarea rezultatului.
- Resolved: Promise s-a finalizat cu succes și a produs un rezultat.
- Rejected: Promise nu s-a finalizat cu succes și a produs o eroare.

Promisiunile sunt utilizate în special pentru a face apeluri asincrone către servere sau alte resurse externe, cum ar fi citirea de fișiere sau efectuarea de cereri HTTP, și așteptarea rezultatului acestora. Acestea permit ca operațiile de lungă durată să fie efectuate în fundal, fără a bloca firul principal de execuție, ceea ce poate îmbunătăți performanța și timpul de răspuns al aplicațiilor.

Sintaxa pentru crearea unui obiect Promise în JavaScript este următoarea:

```
const myPromise = new Promise((resolve, reject) => {
  // Aici se află codul asincron care poate dura ceva timp
  // ...

  if (rezultatul este ok) {
    resolve(rezultatul); // Promise este rezolvat cu succes
  } else {
    reject(eroarea); // Promise este respins cu eroare
  }
});
```

Metoda **new Promise()** primește o funcție de tip executor cu doi parametri, **resolve** și **reject**, care sunt funcții de retur care sunt apelate atunci când Promise este rezolvat cu succes sau respins cu eroare. În interiorul funcției executor se află codul asincron care trebuie să fie executat și care poate dura ceva timp. Dacă operația este finalizată cu succes, atunci se va apela **resolve()** cu rezultatul, iar dacă nu, se va apela **reject()** cu o eroare.

După ce obiectul Promise a fost creat, este posibil să se înregistreze funcții de retur cu metode precum **then()** sau **catch()** pentru a procesa rezultatul sau eroarea obținute din Promise.

then() și **catch()** sunt metode utilizate pentru a procesa rezultatul sau eroarea obținute dintr-un obiect Promise. Aceste metode sunt disponibile pe obiectul Promise și sunt utilizate pentru a înregistra funcții de retur care vor fi apelate atunci când obiectul Promise este rezolvat sau respins.

Metoda **then()** este utilizată pentru a înregistra o funcție de retur care va fi apelată atunci când obiectul Promise este rezolvat cu succes. Această funcție primește rezultatul produs de obiectul Promise și poate fi utilizată pentru a procesa sau a afișa aceste date.

```
myPromise.then((rezultat) => {
  console.log("Rezultatul este: ", rezultat);
});
```

Metoda **catch()** este utilizată pentru a înregistra o funcție de retur care va fi apelată atunci când obiectul Promise este respins cu o eroare. Această funcție primește eroarea produsă de obiectul Promise și poate fi utilizată pentru a afișa sau a trata această eroare.

```
myPromise.catch((eroare) => {
  console.log("A apărut o eroare: ", eroare);
});
```

Este important de reținut că metoda **catch()** trebuie să fie apelată după **then()** în cazul în care se dorește tratarea unei erori produse în cadrul obiectului Promise.

resolve() și **reject()** sunt funcții de retur utilizate într-un obiect Promise pentru a indica faptul că operația asincronă a fost finalizată cu succes sau cu eroare.

Metoda **resolve()** este utilizată pentru a indica faptul că obiectul Promise a fost finalizat cu succes și a produs un rezultat. Această funcție primește ca argument rezultatul produs de obiectul Promise și va declanșa executarea tuturor funcțiilor de retur înregistrate cu metoda **then()**.

```
const myPromise = new Promise((resolve, reject) => {
  const rezultat = "Aceasta este o valoare de succes";
  resolve(rezultat);
});

myPromise.then((rezultat) => {
  console.log(rezultat); // "Aceasta este o valoare de succes"
});
```

Metoda **reject()** este utilizată pentru a indica faptul că obiectul Promise a eșuat și a produs o eroare. Această funcție primește ca argument un obiect de tip Error care descrie eroarea și va declanșa executarea tuturor funcțiilor de retur înregistrate cu metoda **catch()**.

```
const myPromise = new Promise((resolve, reject) => {
  const eroare = new Error("A apărut o eroare");
  reject(eroare);
});

myPromise.catch((eroare) => {
  console.log(eroare.message); // "A apărut o eroare"
});
```

Este important de reținut că fiecare obiect Promise trebuie să aibă o metodă de rezolvare (**resolve()**) sau de respingere (**reject()**) pentru a finaliza procesul asincron.

De exemplu:

```
const myPromise = new Promise((resolve, reject) => {
  // Simulăm o operație asincronă care durează 2 secunde
  setTimeout(() => {
    const randomNumber = Math.random();
    if (randomNumber >= 0.5) {
      resolve(randomNumber); // Promise-ul este rezolvat cu succes
    } else {
      reject(new Error("Valoarea generată este mai mică decât 0.5"));
    }
  }, 2000);
});

myPromise.then((rezultat) => {
  console.log("Rezultatul este: ", rezultat);
}).catch((eroare) => {
  console.log("A apărut o eroare: ", eroare.message);
});
```

În acest exemplu, am creat un obiect Promise cu o operație asincronă simulată care durează 2 secunde. În funcția executor a Promise-ului, am generat un număr aleatoriu și am decis să rezolvăm sau să respingem Promise-ul în funcție de valoarea generată. Dacă valoarea generată este mai mare sau egală cu 0.5, atunci Promise-ul este rezolvat cu succes și valoarea generată este transmisă funcției înregistrate cu **then()**. În caz contrar, Promise-ul este respins cu o eroare și aceasta este transmisă funcției înregistrate cu **catch()**.

Obiectul Promise are câteva metode disponibile pentru a gestiona starea Promise-ului și rezultatul acestuia. Cele mai utilizate metode sunt:

1. **.then()**: Această metodă este apelată atunci când Promise-ul este rezolvat cu succes. Primește o funcție care va fi apelată cu valoarea rezultatului Promise-ului ca argument.
2. **.catch()**: Această metodă este apelată atunci când Promise-ul este respins. Primește o funcție care va fi apelată cu motivul respingerii Promise-ului ca argument.

3. **.finally()**: Această metodă este apelată indiferent dacă Promise-ul este rezolvat cu succes sau respins. Primește o funcție care va fi apelată fără argumente.
4. **.all()**: Această metodă este utilizată pentru a combina mai multe Promise-uri într-un singur Promise. Primește un array de Promise-uri și va returna un Promise care va fi rezolvat atunci când toate Promise-urile din array-ul dat sunt rezolvate.
5. **.race()**: Această metodă este utilizată pentru a combina mai multe Promise-uri într-un singur Promise. Primește un array de Promise-uri și va returna un Promise care va fi rezolvat atunci când primul Promise din array-ul dat este rezolvat.

Acestea sunt cele mai utilizate metode ale obiectului Promise în JavaScript. În general, aceste metode sunt utilizate pentru a gestiona starea și rezultatul unui Promise, astfel încât să poți prelucra datele returnate sau să gestionezi erorile care ar putea apărea în timpul execuției Promise-ului.

1. Fetch API

Fetch API este o interfață JavaScript modernă pentru a face cereri HTTP asincrone la un server. Această interfață utilizează obiecte Promise pentru a face cereri HTTP și a procesa răspunsurile obținute.

Iată un exemplu de utilizare a metodei **fetch()** pentru a efectua o cerere GET către un server și a procesa răspunsul primit cu o promisiune:

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then((response) => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then((data) => {
    console.log(data);
  })
  .catch((error) => {
    console.error('There was an error:', error);
  });
```

În acest exemplu, utilizăm metoda **fetch()** pentru a efectua o cerere GET către un server la adresa https://jsonplaceholder.typicode.com/posts/1.

Metoda **fetch()** returnează o promisiune care va fi rezolvată cu un obiect Response.

În prima metodă **then()**, verificăm dacă răspunsul primit este "ok" și parsăm răspunsul primit în format JSON, returnând astfel o altă promisiune care va fi rezolvată cu datele parsate.

În a doua metodă **then()**, procesăm datele obținute din răspunsul primit prin afișarea lor în consolă.

În cazul în care apare o eroare în procesul de obținere a datelor, folosim metoda **catch()** pentru a afișa un mesaj de eroare în consolă.

În general, utilizarea metodei **fetch()** împreună cu obiectele Promise este o modalitate simplă și eficientă de a efectua cereri HTTP asincrone și de a procesa răspunsurile obținute în aplicații JavaScript.

2. Async and Await

Async/await este o funcționalitate introdusă în JavaScript în ECMAScript 2017, care facilitează scrierea codului asincron.

Înainte de async/await, JavaScript folosea callback-uri sau Promisiuni pentru a gestiona operațiile asincrone. Cu toate acestea, aceste abordări pot fi complicate și pot duce la crearea unui cod greu de întreținut și de înțeles.

Async/await oferă o modalitate mai simplă și mai clară de a gestiona operațiile asincrone. Permite programatorului să definească o funcție asincronă prin adăugarea cuvântului cheie **async** în fața definiției funcției. În cadrul acestei funcții, programatorul poate utiliza cuvântul cheie **await** pentru a aștepta finalizarea unei alte operații asincrone.

De exemplu, dacă programatorul dorește să facă o cerere HTTP și să proceseze răspunsul, el poate folosi **async/await** astfel:

```
async function getAPI() {
  const response = await fetch('https://example.com/api/data');
  const data = await response.json();
  console.log(data);
}
```

În acest exemplu, **fetch** este o operație asincronă care returnează o Promisiune care se va rezolva atunci când cererea HTTP este finalizată. Folosind cuvântul cheie **await**, funcția **getAPI** așteaptă finalizarea acestei Promisiuni și obține răspunsul primit. Apoi, **response.json()** este, de asemenea, o operație asincronă care returnează o Promisiune care se va rezolva atunci când datele JSON sunt

procesate. Folosind din nou **await**, programatorul așteaptă finalizarea acestei Promisiuni și obține datele procesate.

Async/await face codul mult mai ușor de citit și de înțeles și reduce nevoia de nesting-uri profunde sau de crearea de structuri complicate de Promisiuni sau callback-uri.

În plus, async/await poate fi utilizat împreună cu Promisiuni pentru a crea un cod asincron mai complex. Funcția asincronă poate returna o Promisiune care se va rezolva atunci când toate operațiile asincrone sunt finalizate.

Probleme practice

Sarcină practică Scrie un script JavaScript care să afișeze conținutul text al unui element HTML utilizând proprietatea **innerText**. Asigură-te că scriptul tău selectează elementul corect și afișează textul în mod corect.

Sarcină practică Scrie un script JavaScript care să afișeze conținutul HTML al unui element HTML utilizând proprietatea **innerHTML**. Asigură-te că scriptul tău selectează elementul corect și afișează HTML-ul în mod corect.

Sarcină practică Scrie un script JavaScript care să modifice conținutul text al unui element HTML utilizând proprietatea **innerText**. Asigură-te că scriptul tău selectează elementul corect și modifică textul în mod corect.

Sarcină practică Scrie un script JavaScript care să modifice conținutul HTML al unui element HTML utilizând proprietatea **innerHTML**. Asigură-te că scriptul tău selectează elementul corect și modifică HTML-ul în mod corect.

Sarcină practică Scrie un script JavaScript care să modifice stilul CSS al unui element HTML utilizând proprietatea **style**. Asigură-te că scriptul tău selectează elementul corect și modifică stilul în mod corect.

Sarcină practică Scrie un script JavaScript care să adauge un element HTML într-un element existent utilizând metoda **appendChild()**. Asigură-te că scriptul tău selectează elementul existent și creează noul element în mod corect.

Sarcină practică Scrie un script JavaScript care să creeze un element HTML nou și să-l adauge într-un element existent utilizând metoda **appendChild()**. Asigură-te că scriptul tău creează noul element și îl adaugă în elementul existent în mod corect.

Sarcină practică Scrie un script JavaScript care să șteargă un element HTML utilizând metoda **removeChild()**. Asigură-te că scriptul tău selectează elementul corect și îl șterge în mod corect.

Sarcină practică Scrie un script JavaScript care să sorteze elementele HTML în funcție de anumite criterii (de exemplu, ordine alfabetică sau ordine numerică). Asigură-te că scriptul tău selectează elementele corecte și le sortează în mod corect.

Sarcină practică Scrie un script JavaScript care să selecteze elemente HTML dintr-o pagină web și să le modifice folosind metodele native ale DOM-ului. Asigură-te că scriptul tău utilizează metodele corecte ale DOM-ului și este optimizat pentru performanță.

Pentru această sarcină, poți încerca să folosești metode precum **querySelector()** și **querySelectorAll()** pentru a selecta elemente din pagină și metodele native ale DOM-ului pentru a le modifica, cum ar fi **setAttribute()**, **appendChild()**, **removeChild()**, **classList.add()**, **classList.remove()** etc. Poți încerca să creezi și să adaugi elemente noi în pagină, să schimbi stilul CSS al elementelor, să aplici animații și multe altele. Important este să te asiguri că scriptul tău este scris într-un mod eficient și optimizat pentru performanță, pentru a reduce timpul de încărcare și pentru a menține o experiență de utilizare plăcută pentru utilizator.

Sarcină practică Implementează o aplicație simplă în care utilizatorii să poată căuta informații despre vremea curentă într-o anumită locație, folosind API-ul OpenWeatherMap și async/await.

Sarcină practică Creează o aplicație care folosește metoda **.fetch()** pentru a prelua informații dintr-un API extern, precum lista de cărți disponibile într-o librărie, și afișează aceste informații într-un mod atractiv.

QUIZ

1. Ce este un Promise în JavaScript?
   a) Un obiect ce permite lucrul cu operații asincrone într-un mod mai ușor.
   b) O funcție simplă utilizată pentru a returna valori într-un mod sincron.
   c) O metodă de încapsulare a funcțiilor în cadrul obiectelor.
2. Ce este async/await în JavaScript?
   a) O metodă de a face mai ușoară lucrul cu Promise-uri într-un mod secvențial.
   b) O metodă de a transforma o funcție asincronă într-una sincronă.
   c) O metodă de a crea obiecte asincrone.
3. Ce este o cerere (request) în cadrul unei aplicații web?
   a) O cerere HTTP făcută către un server web.
   b) Un proces prin care se extrag date dintr-o bază de date.
   c) O funcție utilizată pentru a executa cod JavaScript în mod asincron.

4. Cum se poate face o cerere HTTP în JavaScript?
   a) Utilizând obiectul XMLHttpRequest.
   b) Utilizând metoda fetch().
   c) Ambele variante sunt corecte.
5. Ce este JSON?
   a) Un format de stocare și schimb de date sub formă de text.
   b) O metodă de criptare a datelor în JavaScript.
   c) O bibliotecă de funcții utilizate pentru operarea cu obiecte.
6. Ce este un loader în cadrul unei aplicații web?
   a) Un indicator vizual utilizat pentru a indica că se așteaptă încărcarea unor date.
   b) Un tip de obiect JavaScript utilizat pentru a stoca date în mod persistent.
   c) O metodă de optimizare a imaginilor folosite în cadrul unei aplicații.
7. Cum se poate utiliza async/await pentru a extrage datele dintr-un JSON și a le afișa pe pagină?
   a) Utilizând metoda parse() pentru a transforma datele JSON într-un obiect JavaScript, apoi utilizând o buclă for pentru a le afișa.
   b) Folosind o funcție asincronă pentru a extrage datele JSON, apoi utilizând await pentru a aștepta finalizarea operației, apoi afișând datele pe pagină.
   c) Prin utilizarea obiectului Promise pentru a asigura finalizarea extragerii datelor, apoi utilizând o funcție sincronă pentru a le afișa pe pagină.
8. Ce sunt callback-urile în JavaScript?
   a) Funcții utilizate pentru a prelua și trata rezultatele obținute în urma unor operații asincrone.
   b) Obiecte utilizate pentru a comunica între diferite aplicații web.
   c) Funcții utilizate pentru a apela alte funcții într-un mod secvențial.
9. Cum se poate folosi o funcție callback pentru a trata rezultatul unei cereri HTTP?
   a) Prin utilizarea unei metode de callback a obiectului XMLHttpRequest.
   b) Prin folosirea unei funcții callback în cadrul metodei fetch().
   c) Prin utilizarea obiectului Promise pentru a trata rezultatul cererii HTTP.
10. Ce este un modul în JavaScript?
    a) Un fișier care conține cod JavaScript ce poate fi folosit în alte părți ale aplicației.
    b) Un obiect folosit pentru a encapsula date în cadrul unei aplicații.
    c) O funcție utilizată pentru a modifica obiecte existente în cadrul aplicației.

11. Ce este metoda **scrollTo** în Javascript? a) O metodă pentru a determina coordonatele curente ale ferestrei b) O metodă pentru a efectua o derulare până la o anumită poziție c) O metodă pentru a obține dimensiunile unei ferestre d) O metodă pentru a obține coordonatele mouse-ului pe ecran
12. Ce proprietate Javascript este folosită pentru a obține dimensiunile unei ferestre de browser? a) **size** b) **width** c) **height** d) **dimensions**
13. Ce proprietate Javascript este folosită pentru a obține coordonatele unui element în interiorul documentului? a) **offsetCoordinates** b) **position** c) **coordinates** d) **offsetTop** și **offsetLeft**
14. Ce metodă Javascript poate fi folosită pentru a efectua o derulare la un anumit element din pagină? a) **scrollTo** b) **scrollBy** c) **scrollIntoView** d) **scrollTop**
15. Ce proprietate Javascript poate fi folosită pentru a obține coordonatele actuale de derulare ale unei pagini? a) **currentScroll** b) **scrollPosition** c) **scrollTop** d) **scrollCoordinates**

Capitol VI. Evenimente

UNITĂȚI DE CONȚINUT

1) Evenimente provocate de:
   - Mouse;
   - Tastatură;
   - Fereastra browser-ului.
2) Atribuirea evenimentelor prin atributele event al elementului și prin metoda addEventListener().

ABILITĂȚI

- Utilizarea evenimentelor preluate de fereastra browser-ului.
- Utilizarea evenimentelor preluate de la mouse.
- Utilizarea evenimentelor preluate de la tastatură.

Evenimentele în JavaScript sunt acțiuni sau comportamente care au loc într-un document HTML, cum ar fi un clic pe un element, o apăsare de tastă sau o modificare a valorii unui câmp de intrare. Aceste evenimente sunt generate de utilizator sau de sistem și pot fi capturate și tratate de către codul JavaScript pentru a răspunde la aceste evenimente și a efectua acțiuni specifice.

JavaScript oferă posibilitatea de a atașa funcții de tratare a evenimentelor la elemente HTML pentru a răspunde la evenimente specifice. Aceste funcții sunt numite funcții de eveniment și sunt declanșate automat când are loc evenimentul asociat. Există multe tipuri diferite de evenimente, cum ar fi evenimentele de click, de **submit**, de **hover**, de tastatură sau de încărcare a paginii.

Un exemplu de cod JavaScript care utilizează evenimente ar putea arăta astfel:

```
// obținem elementul HTML de căutat
var button = document.getElementById("myButton");

// atașăm funcția de tratare a evenimentului onclick
button.onclick = function() {
    alert("Butonul a fost apăsat!");
};
```

În acest exemplu, funcția anonimă atașată evenimentului onclick va fi declanșată automat când utilizatorul face clic pe butonul HTML cu ID-ul "myButton". Acest lucru va afișa o casetă de dialog cu un mesaj de informare.

În concluzie, evenimentele sunt o componentă importantă a dezvoltării web, deoarece permit interacțiunea utilizatorului cu paginile web și oferă posibilitatea programatorilor de a crea interactivitate și dinamism prin intermediul codului JavaScript.

1. Evenimente provocate de mouse, tastatură și fereastra browser-ului.

Evenimentele mous-ului

În JavaScript, **mouse**-ul poate fi utilizat pentru a detecta mai multe evenimente, cum ar fi:

1. click - Acest eveniment este declanșat atunci când se face clic pe un element cu mouse-ul.
2. mouseover - Acest eveniment este declanșat atunci când mouse-ul intră într-un element.
3. mouseout - Acest eveniment este declanșat atunci când mouse-ul iese dintr-un element.
4. mousemove - Acest eveniment este declanșat atunci când mouse-ul se mișcă peste un element.
5. mousedown - Acest eveniment este declanșat atunci când se apasă butonul din stânga al mouse-ului pe un element.
6. mouseup - Acest eveniment este declanșat atunci când se eliberează butonul din stânga al mouse-ului pe un element.

Pentru a gestiona aceste evenimente, puteți utiliza funcții de manipulare a evenimentelor în JavaScript, cum ar fi addEventListener(). De exemplu, următorul cod demonstrează cum să detectați evenimentul click pe un element și să afișați un mesaj în consola browser-ului:

```
document.getElementById("myElement").addEventListener("click", function() {
  console.log("Elementul a fost clicat!");
});
```

Acest cod va atașa o funcție anonimă ca manipulator de eveniment pentru elementul cu id-ul "myElement". Atunci când acest element este clicat, se va afișa un mesaj în consola browser-ului.

Evenimentele tastaturii

În JavaScript, tastatura poate fi utilizată pentru a detecta mai multe evenimente, cum ar fi:

1. keydown - Acest eveniment este declanșat atunci când o tastă este apăsată.
2. keyup - Acest eveniment este declanșat atunci când o tastă este eliberată.
3. keypress - Acest eveniment este declanșat atunci când o tastă este apăsată și eliberată (înainte de keyup).

Pentru a gestiona aceste evenimente, puteți utiliza funcții de manipulare a evenimentelor în JavaScript, cum ar fi addEventListener(). De exemplu, următorul cod demonstrează cum să detectați evenimentul keydown pe pagina și să afișați codul ASCII al tastelor apăsate în consola browser-ului:

```
document.addEventListener("keydown", function(event) {
  console.log("Tasta apasata are codul ASCII: " + event.keyCode);
});
```

Acest cod va atașa o funcție anonimă ca manipulator de eveniment pentru întreaga pagină. Atunci când o tastă este apăsată, se va afișa codul ASCII al tastelor apăsate în consola browser-ului.

Evenimentele provocate de fereastră

În JavaScript, obiectul global **window** poate declanșa mai multe evenimente, cum ar fi:

1. load - Acest eveniment este declanșat atunci când o pagină web a fost încărcată complet.
2. resize - Acest eveniment este declanșat atunci când dimensiunea ferestrei browser-ului este modificată.
3. scroll - Acest eveniment este declanșat atunci când fereastra browser-ului este derulată în sus sau în jos.
4. unload - Acest eveniment este declanșat atunci când o pagină web este închisă sau când utilizatorul navighează la o altă pagină.

Pentru a gestiona aceste evenimente, puteți utiliza funcții de manipulare a evenimentelor în JavaScript, cum ar fi addEventListener(). De exemplu, următorul cod demonstrează cum să detectați evenimentul load și să afișați un mesaj în consola browser-ului:

```
window.addEventListener("load", function() {
  console.log("Pagina a fost încărcată complet!");
});
```

Acest cod va atașa o funcție anonimă ca manipulator de eveniment pentru obiectul global **window**. Atunci când pagina este încărcată complet, se va afișa un mesaj în consola browser-ului.

De asemenea, puteți utiliza aceeași metodă pentru a gestiona celelalte evenimente provocate de obiectul global **window**.

2. Atribuirea evenimentelor prin atributele event al elementului și prin metoda addEventListener().

În JavaScript, puteți atribui gestionari de evenimente direct folosind atributele eveniment ale elementului HTML. Aceasta este o metodă mai veche de a gestiona evenimentele și este cunoscută sub numele de "modelul de atribut al evenimentelor". Cu toate acestea, este recomandat să utilizați metoda **addEventListener** prezentată anterior, deoarece oferă o mai mare flexibilitate și separă logica JavaScript de markup-ul HTML.

Pentru a atribui un gestionar de eveniment folosind atributele eveniment, se adaugă un atribut cu numele evenimentului dorit la elementul HTML. Valoarea atributului va fi codul JavaScript care va fi executat atunci când evenimentul respectiv are loc.

Iată un exemplu de atribuire a unui gestionar de eveniment pentru clic folosind atributul **onclick**:

```
<button onclick="myFunction()">Apasă-mă</button>

<script>
function myFunction() {
  console.log('Buton apăsat!');
}
</script>
```

În acest exemplu, atunci când butonul este apăsat, funcția **myFunction** va fi apelată și va afișa mesajul "Buton apăsat!" în consolă.

Este important de menționat că folosirea atributelor eveniment în exces poate duce la cod greu de întreținut și depanat, deoarece logica JavaScript este amestecată cu markup-ul HTML. Prin urmare, metoda recomandată pentru atribuirea gestionarilor de evenimente este utilizarea **addEventListener**.

Sintaxa generală a metodei **addEventListener** este următoarea:

```
element.addEventListener(eveniment, funcție, useCapture);
```

- **element**: Elementul DOM la care doriți să atașați gestionarul de eveniment.
- **eveniment**: Un șir de caractere care reprezintă tipul evenimentului pe care doriți să-l ascultați, cum ar fi "click", "mouseover", "keydown", etc.
- **funcție**: Funcția care va fi executată atunci când apare evenimentul specificat. Această funcție este în mod obișnuit denumită gestionarul de eveniment sau funcția de apel invers.
- **useCapture** (opțional): O valoare booleană care determină utilizarea capturării evenimentului (**true**) sau a bubbling-ului evenimentului (**false**) ca mecanism de propagare a evenimentului. Acest parametru este în mod obișnuit setat la **false** dacă nu este specificat.

Iată un exemplu care arată cum se utilizează **addEventListener** pentru a atașa un gestionar de eveniment pentru clic la un element de tip buton:

```
// HTML: <button id="myButton">Apasă-mă</button>

const button = document.getElementById('myButton');

button.addEventListener('click', function(event) {
  console.log('Buton apăsat!');
});
```

Pentru a adăuga un eveniment la un element fără a utiliza **addEventListener**, puteți folosi metoda **on[event]** a elementului respectiv. În loc de **[event]**, înlocuiți cu numele evenimentului dorit, cum ar fi **click**, **mouseover**, **keydown**, etc. Iată un exemplu:

```
const button = document.getElementById('myButton');

button.onclick = function(event) {
  console.log('Buton apăsat!');
};
```

În acest exemplu, am atribuit funcția anonimă către proprietatea **onclick** a elementului **button**. Atunci când butonul este apăsat, funcția respectivă va fi apelată și va afișa mesajul "Buton apăsat!" în consolă.

Este important de menționat că atunci când utilizați metoda **on[event]**, suprascrieți orice funcție de gestionare a evenimentului anterioară atașată elementului. De asemenea, nu puteți atașa mai multe funcții de gestionare a evenimentului folosind această metodă, așa cum ați putea face cu **addEventListener**.

Este recomandat să utilizați **addEventListener** pentru a adăuga și gestiona evenimentele, deoarece oferă mai multă flexibilitate și permite atașarea mai multor funcții de gestionare a evenimentului la același element.

3. Formulare, metode și proprietăți.

Un formular este o interfață de utilizator prin intermediul căreia utilizatorii pot trimite date sau informații către un server web pentru a fi prelucrate și stocate. Formularele sunt utilizate pe scară largă pe site-urile web pentru a permite utilizatorilor să interacționeze cu acestea și să furnizeze informații.

Formularele sunt create folosind etichete HTML, cum ar fi **<form>** și **<input>**, care permit utilizatorilor să introducă informații cum ar fi numele, adresa de e-mail, comentarii sau să selecteze opțiuni dintr-un meniu. După ce utilizatorul a completat informațiile într-un formular, acestea sunt trimise serverului web pentru prelucrare și stocare.

Formularele pot fi folosite pentru o varietate de scopuri, inclusiv:

- Înregistrarea utilizatorilor pe un site web
- Completarea unei comenzi pentru cumpărarea unui produs
- Contactarea proprietarului unui site web prin intermediul unei pagini de contact
- Înscrierea la newsletter-ul unui site web
- Furnizarea de feedback sau evaluări pentru un produs sau serviciu

Unii utilizatori pot fi reticenți în a furniza informații personale sau confidențiale într-un formular, astfel încât este important ca site-urile web să ofere utilizatorilor încredere și asigurarea confidențialității datelor lor. Acest lucru poate fi realizat prin utilizarea unor protocoale de securitate, cum ar fi HTTPS și SSL, pentru a proteja datele transferate prin intermediul formularului.

În JavaScript, formularul este reprezentat prin obiectul **HTMLFormElement**, iar acesta dispune de o serie de proprietăți utile pentru a accesa și manipula datele introduse de utilizatori într-un formular. Iată câteva dintre cele mai importante proprietăți ale unui formular în JavaScript:

1. **action**: specifică adresa URL către care trebuie trimise datele din formular atunci când acesta este trimis. De exemplu, **document.forms[0].action = "procesare.php";**.

2. **elements**: oferă o colecție cu toate elementele formularului, cum ar fi câmpurile text, meniurile derulante sau butoanele. De exemplu, **var elemente = document.forms[0].elements;**.

3. **length**: indică numărul de elemente din formular. De exemplu, **var numar_elemente = document.forms[0].length;**.

4. **method**: specifică metoda HTTP utilizată pentru a trimite datele formularului către server. Această valoare poate fi "GET" sau "POST". De exemplu, **document.forms[0].method = "POST";**.

5. **name**: specifică numele formularului, care poate fi utilizat pentru a face referire la acesta din JavaScript. De exemplu, **document.forms[0].name = "formular_contact";**.

6. **target**: specifică locația unde ar trebui afișat rezultatul trimiterii formularului. De exemplu, **document.forms[0].target = "_blank";** va deschide o pagină nouă pentru afișarea rezultatului.

Acestea sunt doar câteva dintre proprietățile disponibile pentru obiectul **HTMLFormElement** în JavaScript. Există și alte proprietăți și metode utile pentru a accesa și manipula datele dintr-un formular, cum ar fi **reset()**, care elimină toate datele din formular și revine la starea inițială, sau **submit()**, care trimite datele formularului către server pentru prelucrare.

În JavaScript, obiectul **HTMLFormElement** are și o serie de metode utile pentru a gestiona și manipula datele dintr-un formular. Iată câteva dintre cele mai comune metode ale unui formular în JavaScript:

1. **submit()**: această metodă trimite datele formularului către server pentru prelucrare. De exemplu, **document.forms[0].submit();**.

2. **reset()**: această metodă elimină toate datele din formular și revine la starea inițială. De exemplu, **document.forms[0].reset();**.

3. **checkValidity()**: această metodă verifică dacă toate câmpurile din formular sunt completate corect și în conformitate cu regulile de validare specificate. De exemplu, **var valid = document.forms[0].checkValidity();**.

4. **reportValidity()**: această metodă afișează mesajele de eroare pentru câmpurile care nu sunt completate corect sau nu sunt valide. De exemplu, **var valid = document.forms[0].reportValidity();**.

5. **resetValidation()**: această metodă resetează toate mesajele de eroare pentru câmpurile din formular. De exemplu, **document.forms[0].resetValidation();**.

6. **submit(event)**: această metodă este utilizată în principal în combinație cu evenimentul "submit" pentru a preveni comportamentul implicit de trimitere a formularului și a permite gestionarea datelor în JavaScript. De exemplu, **document.forms[0].addEventListener("submit", function(event) { event.preventDefault(); });**.

Acestea sunt doar câteva dintre metodele disponibile pentru obiectul **HTMLFormElement** în JavaScript. Există și alte metode utile pentru a gestiona și manipula datele dintr-un formular, cum ar fi **submit(event)** și **reset(event)**, care permit gestionarea datelor în JavaScript înainte de a fi trimise către server.

Accesarea elementelor formularului

Pentru a accesa elementele dintr-un formular în JavaScript, poți utiliza proprietatea **elements** a obiectului **HTMLFormElement**. Această proprietate returnează o colecție cu toate elementele formularului, cum ar fi câmpurile text, meniurile derulante sau butoanele.

Pentru a accesa un element specific din formular, poți folosi indicele sau numele acestuia. De exemplu, pentru a accesa primul câmp text dintr-un formular, poți folosi:

```
var formular = document.forms[0]; // Accesează primul formular de pe pagină
var camp_text = formular.elements[0]; // Accesează primul câmp text din formular
```

Dacă preferi să accesezi elementele formularului după numele acestora, poți utiliza proprietatea **name** a fiecărui element. De exemplu, pentru a accesa un câmp text cu numele "nume_utilizator", poți folosi:

```
var formular = document.forms[0]; // Accesează primul formular de pe pagină
var camp_text = formular.elements["nume_utilizator"]; // Accesează câmpul text
```

După ce ai accesat un element din formular, poți accesa și modifica proprietățile și valorile acestuia în JavaScript. De exemplu, pentru a accesa valoarea unui câmp text și a o afișa în consolă, poți folosi:

```
var formular = document.forms[0]; // Accesează primul formular de pe pagină
var camp_text = formular.elements["nume_utilizator"]; // Accesează câmpul text cu
var valoare = camp_text.value; // Accesează valoarea introdusă de utilizator
console.log("Valoarea introdusă este: " + valoare); // Afișează valoarea în consol
```

Acestea sunt câteva exemple de cum poți accesa și manipula elementele dintr-un formular în JavaScript.

4. Evenimentele formularelor.

Există mai multe evenimente disponibile pentru formulare în JavaScript, iar mai jos am listat cele mai comune evenimente pentru formulare:

1. **onsubmit** - Acest eveniment este declanșat atunci când utilizatorul trimite formularul. Poate fi folosit pentru a verifica și valida datele introduse de utilizator înainte de a trimite formularul către server.

```
document.querySelector('form').onsubmit = function() {
  // validarea datelor din formular
  return false; // oprește trimiterea formularului către server
}
```

2. **onreset** - Acest eveniment este declanșat atunci când utilizatorul reseta formularul la valorile implicite. Poate fi folosit pentru a reseta câmpurile personalizate din formular, cum ar fi notificările de eroare sau valorile selectate.

```
document.querySelector('form').onreset = function() {
  // resetează valorile personalizate din formular
}
```

3. **onfocus** - Acest eveniment este declanșat atunci când utilizatorul focalizează un câmp din formular. Poate fi folosit pentru a oferi feedback vizual utilizatorului atunci când un câmp este selectat.

```
document.querySelector('input').onfocus = function() {
  // schimbă culoarea fundalului câmpului
}
```

4. **onblur** - Acest eveniment este declanșat atunci când utilizatorul părăsește un câmp din formular. Poate fi folosit pentru a verifica și valida datele introduse de utilizator atunci când părăsește câmpul.

```
document.querySelector('input').onblur = function() {
  // verifică și validează datele introduse în câmp
}
```

5. **onchange** - Acest eveniment este declanșat atunci când utilizatorul modifică valoarea unui câmp din formular, cum ar fi un câmp select sau un câmp de tip checkbox. Poate fi folosit pentru a actualiza interfața utilizatorului sau pentru a verifica și valida datele introduse.

```
document.querySelector('select').onchange = function() {
  // actualizează interfața utilizatorului
}
```

6. **onkeyup** - Acest eveniment este declanșat atunci când utilizatorul eliberează o tastă după ce a fost apăsată într-un câmp din formular. Poate fi folosit pentru a verifica și valida datele introduse de utilizator.

```
document.querySelector('input').onkeyup = function() {
  // verifică și validează datele introduse în câmp
}
```

7. **onkeydown** - Acest eveniment este declanșat atunci când utilizatorul apasă o tastă într-un câmp din formular. Poate fi folosit pentru a controla interacțiunea utilizatorului cu formularul și pentru a verifica și valida datele introduse.

```
document.querySelector('input').onkeydown = function() {
  // controlează interacțiunea utilizatorului cu formularul
}
```

8. **onkeypress** - Acest eveniment este declanșat atunci când utilizatorul apasă o tastă într-un câmp din formular și acea tastă poate fi afișată. Poate fi folosit pentru a controla interacțiunea utilizatorului cu formularul și pentru a verifica și valida datele introduse.

```
document.querySelector('input').onkeypress = function() {
  // controlează interacțiunea utilizatorului cu formularul
}
```

9. **oninput** - Acest eveniment este declanșat atunci când valoarea unui element de intrare (cum ar fi un câmp de text) este modificată de către utilizator sau prin script. Poate fi folosit pentru a actualiza interfața utilizatorului sau pentru a verifica și valida datele introduse.

```
document.querySelector('input').oninput = function() {
  // actualizează interfața utilizatorului sau verifică și validează datele
}
```

Acestea sunt cele mai comune evenimente pentru formulare în JavaScript, dar există și alte evenimente disponibile, cum ar fi **oninput**, **onselect**, **onpaste**, **oncut**, etc., care pot fi folosite pentru a controla interacțiunea utilizatorului cu formularul și pentru a valida și verifica datele introduse.

Pentru a identifica ce taste se vor apăsa în JavaScript, putem utiliza evenimentul **keydown** sau **keyup**. Acest eveniment este declanșat atunci când utilizatorul apasă sau eliberează o tastă.

Pentru a afla ce tastă a fost apăsată, putem folosi proprietatea **keyCode** a obiectului evenimentului, care va returna codul unic asociat cu tastă apăsată. Codul unic este un număr întreg, care variază în funcție de tastatura folosită.

Mai jos este un exemplu simplu care afișează codul unic al tastelor apăsate în consolă:

```
document.addEventListener('keydown', function(event) {
  console.log(event.keyCode);
});
```

Acest cod va afișa codul unic al tastelor apăsate în consolă atunci când se apasă o tastă în pagina web.

Este important să menționăm că proprietatea **keyCode** este învechită, iar de la standardul ECMAScript 6, aceasta a fost înlocuită cu proprietatea **key**. Proprietatea **key** furnizează o valoare mai semnificativă, care este independentă de configurația tastaturii și este mai ușor de utilizat.

Probleme practice

1. ***Sarcină practică*** Creați un formular simplu cu diferite tipuri de elemente (text, select, radio, etc.) și folosiți JavaScript pentru a accesa și a manipula valorile acestor elemente.
2. ***Sarcină practică*** Implementați un sistem de validare a formularului în JavaScript, astfel încât să se afișeze un mesaj de eroare dacă utilizatorul nu completează corect toate câmpurile obligatorii.
3. ***Sarcină practică*** Utilizați evenimentul **keydown** pentru a crea o aplicație de joc în care utilizatorul trebuie să apese o combinație specifică de taste pentru a trece la nivelul următor.
4. ***Sarcină practică*** Implementați o căutare live într-un formular cu ajutorul evenimentului **input**, astfel încât să se afișeze opțiunile corespunzătoare în timp ce utilizatorul scrie.

5. ***Sarcină practică*** Creați un sistem de filtrare pentru o listă de elemente utilizând un formular cu elemente de tip checkbox și evenimentul **change**.
6. ***Sarcină practică*** Creați un formular cu elemente de tip range și afișați valorile selectate în timp real cu ajutorul evenimentului **input**.
7. ***Sarcină practică*** Implementați un sistem de autocompletare într-un formular cu ajutorul evenimentului **keyup**, astfel încât să se afișeze sugestii de completare pentru cuvintele scrise de utilizator.
8. ***Sarcină practică*** Implementați un sistem de scroll infinit într-o pagină web cu ajutorul evenimentului scroll, astfel încât să încărcați noi elemente atunci când utilizatorul ajunge la sfârșitul paginii.

Sarcină practică complexă Nr. 1:

Imaginează-ți că trebuie să construiești o pagină web care afișează o listă de produse, împreună cu imaginea, prețul și descrierea fiecăruia. Această listă de produse este stocată într-un API extern la care trebuie să accesezi prin intermediul Fetch API. Scopul acestei sarcini este să permiți utilizatorului să filtreze această listă de produse în funcție de preț și descriere, prin intermediul unui formular și a evenimentului click.

Iată specificațiile detaliate pentru această sarcină:

1. Accesează lista de produse utilizând Fetch API

- URL-ul către API trebuie să fie stocat într-o constantă numită **API_URL**.
- Utilizează metoda **fetch()** pentru a obține lista de produse.
- Transformă răspunsul primit de la API în format JSON, folosind metoda **json()**.

2. Afiseaza lista de produse pe pagina web

- Creați o listă HTML goală cu un **id** specificat ca "produse".
- Pentru fiecare element din lista de produse primită de la API, creează un element de listă și adaugă-l la lista HTML, conținând imaginea, prețul și descrierea produsului.

3. Creează un formular de filtrare

- Creați un formular HTML care conține două câmpuri de text și un buton de submit.
- Primul câmp de text este pentru prețul minim, al doilea câmp de text este pentru cuvinte cheie în descriere.
- Butonul de submit trebuie să aibă un eveniment click atașat.

4. Implementează funcția de filtrare

- Când utilizatorul face clic pe butonul de submit, preia valorile din câmpurile de text.
- Utilizează metoda **filter()** pentru a filtra lista de produse în funcție de valorile introduse de utilizator.
- După filtrare, golește lista HTML și afișează doar produsele care corespund criteriilor de filtrare.

5. Actualizarea listei de produse

- Dacă utilizatorul modifica sau șterge textul din câmpurile de filtrare, lista de produse trebuie actualizată automat fără a fi necesar un nou clic pe butonul de submit.

Această sarcină combină utilizarea Fetch API pentru a accesa datele dintr-un API extern, evenimentul click pentru a iniția procesul de filtrare și filtrarea elementelor pentru a afișa doar elementele care corespund criteriilor de filtrare introduse de utilizator.

Sarcină practică complex Nr. 2:

Imaginează-ți că ai o pagină web care afișează o listă de imagini, împreună cu un buton de încărcare suplimentară. Scopul acestei sarcini este să încărcați imagini suplimentare de la un API extern atunci când utilizatorul face clic pe butonul de încărcare și să permiteți utilizatorului să filtreze această listă de imagini în funcție de dimensiune și cuvinte cheie în titlu.

Iată specificațiile detaliate pentru această sarcină:

1. Accesează lista de imagini utilizând Fetch API

- URL-ul către API trebuie să fie stocat într-o constantă numită **API_URL**.
- Utilizează metoda **fetch()** pentru a obține lista de imagini.
- Transformă răspunsul primit de la API în format JSON, folosind metoda **json()**.

2. Afiseaza lista de imagini pe pagina web

- Creați o listă HTML goală cu un **id** specificat ca "imagini".
- Pentru fiecare element din lista de imagini primită de la API, creează un element de imagine și adaugă-l la lista HTML, conținând titlul și dimensiunea imaginii.

3. Creează un buton de încărcare suplimentară

- Creați un buton HTML care poate fi apăsat pentru încărcarea imaginilor suplimentare.
- Butonul trebuie să aibă un eveniment click atașat.

4. Implementează funcția de încărcare suplimentară

- Când utilizatorul face clic pe butonul de încărcare, obțineți următorul set de imagini de la API-ul extern și adăugați-le la lista de imagini existentă.

5. Implementează funcția de filtrare

- Creați două câmpuri de text pentru dimensiune și cuvinte cheie în titlu.
- Atunci când utilizatorul introduce valori în câmpurile de filtrare și face clic pe butonul de filtrare, lista de imagini trebuie să fie filtrată în funcție de valorile introduse de utilizator.
- După filtrare, golește lista HTML și afișează doar imaginile care corespund criteriilor de filtrare.

6. Actualizarea listei de imagini

- Dacă utilizatorul modifica sau șterge textul din câmpurile de filtrare, lista de imagini trebuie actualizată automat fără a fi necesar un nou clic pe butonul de filtrare.

Această sarcină combină utilizarea Fetch API pentru a accesa datele dintr-un API extern, evenimentul click pentru a iniția procesul de încărcare suplimentară și filtrarea elementelor pentru a afișa doar elementele care corespund criteriilor de filtrare introduse de utilizator.

Sarcină practică complexă: Nr. 3:

Scopul acestei sarcini este să creezi un calculator de buzunar electronic utilizând HTML, CSS și JavaScript.

Specificatiile sarcinii:

1. Interfața utilizatorului:
    - Creați un layout de calculator cu butoane pentru cifre, operatori și funcții matematice.
    - Afișați o casetă de text pentru a afișa inputul și rezultatul.
    - Butonul "C" va sterge toate datele din caseta de text.
2. Funcționalitatea calculatorului:
    - Calculatorul trebuie să poată efectua operațiile de bază, precum adunarea, scăderea, înmulțirea și împărțirea.
    - Calculatorul trebuie să poată gestiona parantezele.
    - Calculatorul trebuie să poată efectua funcții matematice, precum radicalul, logaritmul și trigonometria.
3. Evenimentul click:
    - Utilizați evenimentul click pentru a permite utilizatorului să introducă cifre și operatori.
    - Utilizați evenimentul click pentru a permite utilizatorului să efectueze operații.
    - Utilizați evenimentul click pentru a permite utilizatorului să șteargă inputul.

4. Filtarea elementelor:
    - Utilizați metodele de filtrare ale elementelor pentru a filtra butoanele în funcție de tipul lor (cifre, operatori, funcții matematice).
5. Utilizarea de funcții și metode:
    - Folosiți funcții pentru a gestiona operațiile matematice și pentru a efectua calculele.
    - Utilizați metode pentru a valida inputul utilizatorului și pentru a manipula datele introduse.
6. Implementarea:
    - Implementați calculatorul utilizând HTML, CSS și JavaScript.
7. Creativitate:
    - Adăugați elemente creative la calculator, precum o opțiune pentru selectarea temei de culoare sau un istoric al operațiilor.

QUIZ

1. Care este evenimentul care este declanșat atunci când utilizatorul apasă și apoi eliberează butonul mouse-ului?

   a. click b. mouseover c. mouseup d. mousedown

2. Ce eveniment este declanșat atunci când utilizatorul mișcă cursorul mouse-ului peste un element?

   a. mouseover b. mousemove c. mouseenter d. mouseleave

3. Ce eveniment este declanșat atunci când utilizatorul apasă o tastă a tastaturii?

   a. keypress b. keyup c. keydown d. keyboard

4. Ce eveniment este declanșat atunci când utilizatorul eliberează o tastă a tastaturii?

   a. keyup b. keypress c. keydown d. keyboard

5. Care este evenimentul care este declanșat atunci când utilizatorul dă clic pe butonul dreapta al mouse-ului?

   a. contextmenu b. click c. mouseover d. mouseup

6. Care este evenimentul care este declanșat atunci când utilizatorul mișcă cursorul mouse-ului în afara unui element?

   a. mouseout b. mousemove c. mouseleave d. mouseover

7. Ce eveniment este declanșat atunci când utilizatorul redimensionează fereastra browser-ului?

   a. resize b. scroll c. load d. unload

8. Ce eveniment este declanșat atunci când utilizatorul încarcă o pagină web?

   a. resize b. scroll c. load d. unload

9. Care este evenimentul care este declanșat atunci când utilizatorul dă scroll pe o pagină web?

   a. resize b. scroll c. load d. unload

10. Ce eveniment este declanșat atunci când utilizatorul închide fereastra browser-ului?

    a. resize b. scroll c. load d. unload

Capitol VII. Modelul de obiecte ale browser-ului (BOM)

UNITĂȚI DE CONȚINUT

1. Obiectele browser-ului:
¬ Window;
¬ Location;
¬ History;
¬ Navigator;
¬ Screen;
¬ Cookies.
2. Metode de păstrare a datelor:
¬ Local Storage;
¬ Session Storage.

ABILITĂȚI

- Utilizarea proprietăților și metodelor obiectului Window.
- Prelucrarea adresei URL a paginii web.
- Afișarea informației din istoricului browser-ului.
- Obținerea informațiilor despre browser-ul vizitatorului.
- Afișarea informației despre parametrii ecranul utilizatorului.
- Stocare a datelor utilizatorului

Modelul de obiecte ale browser-ului (BOM) este o colecție de obiecte JavaScript furnizate de browser pentru a interacționa cu elemente specifice ale acestuia, cum ar fi fereastra browser-ului, istoricul navigării, adresa URL, meniul de navigare și altele. Aceste obiecte sunt considerate parte din modelul obiectului documentului (DOM), dar sunt separate de acesta din urmă și reprezintă un set de funcționalități specifice browser-ului.

Unele dintre obiectele BOM incluse în majoritatea browserelor sunt:

- **window**: Reprezintă fereastra browser-ului curent și oferă metode pentru manipularea ei (dimensiune, locație, deschidere/închidere, etc.).
- **history**: Furnizează informații despre istoricul navigării utilizatorului și oferă posibilitatea de a naviga înainte și înapoi în acesta.
- **location**: Reprezintă adresa URL a documentului curent și oferă metode pentru a naviga la alte adrese URL.
- **navigator**: Furnizează informații despre browserul utilizatorului (nume, versiune, limbă, etc.) și sistemul de operare.
- **screen**: Furnizează informații despre ecranul utilizatorului (rezoluție, dimensiuni, etc.).

Deși aceste obiecte sunt disponibile în majoritatea browserelor, există și alte obiecte BOM specifice fiecărui browser în parte.

1. Obiectele browser-ului.

Window

Obiectul Window este unul dintre cele mai importante obiecte din JavaScript și face parte din Modelul de Obiecte al Browser-ului (BOM). Reprezintă fereastra curentă a browserului și este accesibil prin intermediul variabilei globale **window**.

Obiectul Window conține proprietăți și metode care permit manipularea și controlul ferestrei curente, precum și a altor ferestre deschise. De exemplu, prin intermediul obiectului Window putem seta dimensiunea și poziția ferestrei, putem deschide o fereastră nouă, putem naviga către o altă pagină web sau putem seta un timer pentru a declanșa o anumită acțiune la un moment dat.

Unele dintre proprietățile obiectului Window sunt:

- **window.innerWidth** și **window.innerHeight**: reprezintă lățimea și înălțimea ferestrei curente, exprimate în pixeli.
- **window.location**: reprezintă locația actuală a ferestrei, inclusiv URL-ul.
- **window.document**: reprezintă documentul încărcat în fereastra curentă.
- **window.parent**: reprezintă obiectul Window al părintelui fereastrei curente, dacă există.
- **window.opener**: reprezintă obiectul Window al ferestrei care a deschis fereastra curentă, dacă există.

Unele dintre metodele obiectului Window sunt:

- **window.alert()**: afișează un mesaj de tip alertă.
- **window.prompt()**: afișează o fereastră de prompt și permite utilizatorului să introducă un text.

- **window.setTimeout()**: setează un timer pentru a declanșa o anumită acțiune după un anumit interval de timp.
- **window.open()**: deschide o fereastră nouă cu un URL specificat.
- **window.close()**: închide fereastra curentă.

Este important de reținut că obiectul Window poate fi utilizat atât în codul JavaScript al paginii, cât și în consola browserului, fără a fi nevoie să îl declarăm explicit.

Location

Obiectul **location** este unul dintre obiectele disponibile în Modelul de Obiecte al Browser-ului (BOM) și este utilizat pentru a accesa și modifica adresa URL a paginii curente. Acesta este accesibil prin intermediul obiectului **window.location** și conține o serie de proprietăți și metode care permit accesarea și manipularea adresei URL.

Unele dintre proprietățile obiectului **location** sunt:

- **location.href**: reprezintă adresa URL a paginii curente.
- **location.protocol**: reprezintă protocolul utilizat pentru a accesa pagina curentă (de exemplu, "http" sau "https").
- **location.hostname**: reprezintă numele de domeniu al paginii curente (de exemplu, "www.exemplu.com").
- **location.port**: reprezintă numărul portului utilizat pentru conexiunea la server (de exemplu, 80 sau 443).
- **location.pathname**: reprezintă calea către fișierul HTML curent.
- **location.hash**: reprezintă partea din URL-ul care urmează imediat după caracterul "#" și este utilizată pentru a accesa un anumit element din pagina curentă.

Unele dintre metodele obiectului **location** sunt:

- **location.assign(URL)**: navighează la adresa URL specificată.

- **location.reload()**: reîncarcă pagina curentă.
- **location.replace(URL)**: înlocuiește adresa URL a paginii curente cu o altă adresă URL.

Deși obiectul **location** poate fi accesat și modificat direct prin intermediul codului JavaScript, este important să se acorde atenție modificării adresei URL a paginii curente, deoarece aceasta poate duce la pierderea datelor sau la erori de securitate.

History

Obiectul **History** este unul dintre obiectele disponibile în Modelul de Obiecte al Browser-ului (BOM) și este utilizat pentru a accesa și manipula istoricul de navigare a ferestrei curente a browserului. Acesta este accesibil prin intermediul obiectului **window.history** și conține o serie de metode care permit navigarea în istoricul de navigare și modificarea acestuia.

Unele dintre metodele obiectului **History** sunt:

- **history.back()**: navighează la pagina anterioară din istoricul de navigare.
- **history.forward()**: navighează la pagina următoare din istoricul de navigare.
- **history.go(n)**: navighează la pagina din istoricul de navigare de la poziția n (unde n poate fi pozitiv, negativ sau zero).

De asemenea, obiectul **History** oferă și o proprietate **length**, care reprezintă numărul de pagini din istoricul de navigare.

Este important de reținut că modificarea istoricului de navigare poate duce la comportamente neașteptate sau erori în aplicațiile web și, prin urmare, trebuie utilizată cu precauție. De exemplu, navigarea în istoricul de navigare prin intermediul metodelor **back()** și **forward()** nu ar trebui să fie utilizată pentru a înlocui funcționalitatea normală de navigare a utilizatorului.

Navigator

Obiectul **Navigator** este unul dintre obiectele disponibile în Modelul de Obiecte al Browser-ului (BOM) și este utilizat pentru a obține informații despre browserul și dispozitivul utilizatorului. Acesta este accesibil prin intermediul obiectului **window.navigator** și conține o serie de proprietăți și metode care permit accesarea informațiilor despre browser și dispozitiv.

Unele dintre proprietățile obiectului **Navigator** sunt:

- **navigator.userAgent**: reprezintă informațiile despre browser și sistemul de operare utilizat.
- **navigator.language**: reprezintă codul de limbă a browserului.
- **navigator.platform**: reprezintă sistemul de operare pe care rulează browserul.

Unele dintre metodele obiectului **Navigator** sunt:

- **navigator.geolocation.getCurrentPosition(successCallback, errorCallback)**: obține informații despre locația geografică a utilizatorului.
- **navigator.sendBeacon(url, data)**: trimite o cerere HTTP asincronă cu date către server folosind metoda HTTP **POST**.

De asemenea, obiectul **Navigator** oferă și proprietăți și metode pentru a obține informații despre capacitatea hardware a dispozitivului, cum ar fi **navigator.hardwareConcurrency** pentru a obține numărul de nuclee de procesor disponibile și **navigator.deviceMemory** pentru a obține cantitatea de memorie disponibilă pentru dispozitiv.

Este important de reținut că unele proprietăți și metode ale obiectului **Navigator** pot fi folosite pentru a crea fingerprint-uri ale utilizatorului, ceea ce poate ridica probleme de securitate și confidențialitate. Prin urmare, utilizarea acestora trebuie făcută cu precauție.

Screen

Obiectul **Screen** este unul dintre obiectele disponibile în Modelul de Obiecte al Browser-ului (BOM) și este utilizat pentru a obține informații despre ecranul utilizatorului. Acesta este accesibil prin intermediul obiectului **window.screen** și conține o serie de proprietăți care permit accesarea informațiilor despre ecran.

Unele dintre proprietățile obiectului **Screen** sunt:

- **screen.width**: reprezintă lățimea ecranului în pixeli.
- **screen.height**: reprezintă înălțimea ecranului în pixeli.

- **screen.availWidth**: reprezintă lățimea disponibilă a ecranului în pixeli, luând în considerare bara de instrumente și orice alte elemente afișate în mod constant pe ecran.
- **screen.availHeight**: reprezintă înălțimea disponibilă a ecranului în pixeli, luând în considerare bara de instrumente și orice alte elemente afișate în mod constant pe ecran.

Aceste proprietăți pot fi folosite pentru a ajusta dimensiunile elementelor afișate pe ecran, pentru a asigura o experiență de utilizare optimă.

De asemenea, obiectul **Screen** oferă și alte proprietăți care furnizează informații despre densitatea pixelilor, culoarea și adâncimea de culoare a ecranului.

Este important de reținut că proprietățile obiectului **Screen** pot fi influențate de setările utilizatorului, cum ar fi rezoluția ecranului și mărimea fontului, și pot varia în funcție de dispozitivul utilizat.

Cookies

Obiectul **Cookies** este unul dintre obiectele disponibile în JavaScript și este utilizat pentru a crea, citi și șterge cookie-uri din browser-ul utilizatorului. Cookie-urile sunt fișiere text mici care sunt stocate pe dispozitivul utilizatorului și sunt utilizate pentru a păstra anumite informații între sesiuni, cum ar fi preferințele utilizatorului sau datele de autentificare.

Obiectul **Cookies** nu este parte a Modelului de Obiecte al Browser-ului (BOM) și nu este accesibil prin intermediul obiectului **window**. În schimb, el poate fi accesat prin intermediul obiectului **document.cookie**. Acesta este un șir de caractere care conține toate cookie-urile stocate pentru pagina curentă, separate prin punct și virgulă.

Pentru a crea un cookie, se poate utiliza următoarea sintaxă:

```
document.cookie = "nume=valoare; expirare=data";
```

nume este numele cookie-ului, **valoare** este valoarea cookie-ului și **expirare** este data la care cookie-ul expiră. Data trebuie să fie în formatul UTC și poate fi specificată ca un obiect **Date**. Alte opțiuni pentru cookie-uri includ specificarea domeniului sau calea pentru care sunt valabile și specificarea dacă cookie-ul poate fi accesat prin intermediul codului JavaScript.

Pentru a citi un cookie, se poate utiliza următoarea sintaxă:

```
var cookieValue = document.cookie.replace(/(?:(?:^|.*;\s*)nume\s*\=\s*([^;]*).*$)|^.*$/, "$1");
```

nume este numele cookie-ului, iar **cookieValue** va conține valoarea asociată acestuia.

Pentru a șterge un cookie, se poate seta valoarea sa la un șir gol și a expirației sale la o dată anterioară celei actuale, astfel:

```
document.cookie = "nume=; expires=Thu, 01 Jan 1970 00:00:00 UTC";
```

Este important de reținut că cookie-urile pot fi folosite pentru a urmări utilizatorul și că acestea pot fi dezactivate în setările browser-ului. De asemenea, este important să se asigure că informațiile stocate în cookie-uri nu conțin informații confidențiale și că acestea sunt utilizate într-un mod responsabil.

2. Metode de păstrare a datelor.

Local Storage

Local Storage este o caracteristică disponibilă în HTML5 care permite stocarea unor cantități mari de date într-un mod persistent pe dispozitivul utilizatorului. Este similar cu cookie-urile, dar are o capacitate mult mai mare și nu trimite date către serverul web în fiecare cerere.

Obiectul **localStorage** este utilizat pentru a accesa și modifica datele stocate în **Local Storage**. Acesta poate fi accesat prin intermediul obiectului **window** în felul următor:

```
window.localStorage
```

Datele stocate în **localStorage** sunt reținute pe dispozitivul utilizatorului chiar și după închiderea browser-ului sau a sistemului de operare. Acestea sunt stocate într-un obiect JSON (JavaScript Object Notation), iar valorile trebuie să fie de tipul string. Pentru a stoca un obiect sau o valoare de alt tip, acesta trebuie să fie convertit într-un șir de caractere utilizând metoda **JSON.stringify()** și apoi salvat în **localStorage**. Pentru a recupera datele, trebuie să folosim metoda **JSON.parse()**.

Pentru a stoca o valoare în **localStorage**, se poate utiliza următoarea sintaxă:

```
localStorage.setItem('nume', 'valoare');
```

nume este numele cheii iar **valoare** este valoarea asociată cheii.

Pentru a recupera o valoare din **localStorage**, se poate utiliza următoarea sintaxă:

```
var valoare = localStorage.getItem('nume');
```

Pentru a șterge o valoare din **localStorage**, se poate utiliza următoarea sintaxă:

```
localStorage.removeItem('nume');
```

Pentru a șterge toate datele stocate în **localStorage**, se poate utiliza metoda **localStorage.clear()**. Este important de reținut că datele stocate în **localStorage** sunt disponibile pentru toate paginile web care utilizează același domeniu și protocol. Prin urmare, acestea ar trebui să fie utilizate cu precauție pentru a evita dezvăluirea sau accesul neautorizat la informațiile utilizatorului.

Capacitatea de memorare a **localStorage** variază în funcție de browser și sistemul de operare utilizat. În general, capacitatea de stocare a **localStorage** este mai mare decât cea a cookie-urilor și poate ajunge la câteva megaocteți.

Mai exact, conform specificației HTML5, **localStorage** ar trebui să aibă o capacitate de cel puțin 5MB pe fiecare origin (adresă de protocol și domeniu) și cel puțin 25MB pe fiecare tab/parcurgere a istoricului. Cu toate acestea, în practică, capacitatea de stocare poate varia în funcție de browser și de setările utilizatorului.

Este important să se țină cont de aceste limite de capacitate atunci când se utilizează **localStorage**. În plus, din cauza caracterului persistent al datelor stocate în **localStorage**, acestea trebuie gestionate cu grijă pentru a evita consumul inutil al spațiului de stocare și pentru a proteja datele utilizatorului.

Pentru a salva un array în **localStorage**, trebuie să îl transformăm într-un șir de caractere utilizând metoda **JSON.stringify()** și apoi să îl salvăm în **localStorage**. De exemplu:

```
var array = [1, 2, 3, 4, 5];
localStorage.setItem('myArray', JSON.stringify(array));
```

Aici am creat un array numit **array** și am salvat acest array în **localStorage** utilizând cheia **'myArray'**. Am convertit array-ul într-un șir de caractere utilizând metoda **JSON.stringify()**.

Pentru a recupera acest array din **localStorage**, trebuie să utilizăm metoda **JSON.parse()** pentru a-l converti înapoi într-un obiect JavaScript:

```
var retrievedArray = JSON.parse(localStorage.getItem('myArray'));
console.log(retrievedArray); // [1, 2, 3, 4, 5]
```

Aici am recuperat șirul de caractere din **localStorage** utilizând cheia **'myArray'**, l-am convertit înapoi într-un obiect JavaScript utilizând metoda **JSON.parse()** și l-am stocat în variabila **retrievedArray**.

Session Storage

sessionStorage este un obiect JavaScript care permite stocarea de date într-o sesiune de utilizator. Datele stocate în **sessionStorage** sunt disponibile doar pentru durata sesiunii de utilizator, adică atât timp cât fereastra sau tab-ul de browser în care a fost deschisă pagina rămâne deschis.

Spre deosebire de **localStorage**, care poate fi accesat din orice fereastră sau tab de browser, **sessionStorage** este specific pentru o fereastră sau un tab de browser. Aceasta înseamnă că dacă utilizatorul deschide o pagină web într-un alt tab sau fereastră, datele stocate în **sessionStorage** nu vor fi disponibile acolo.

Pentru a salva date în **sessionStorage**, se poate utiliza metoda **sessionStorage.setItem()**:

```
sessionStorage.setItem("nume", "John");
```

Pentru a accesa datele stocate în **sessionStorage**, se poate utiliza metoda **sessionStorage.getItem()**:

```
var nume = sessionStorage.getItem("nume");
console.log(nume); // "John"
```

Pentru a șterge o cheie și valoarea corespunzătoare din **sessionStorage**, se poate utiliza metoda **sessionStorage.removeItem()**:

```
sessionStorage.removeItem("nume");
```

Pentru a șterge toate cheile și valorile din **sessionStorage**, se poate utiliza metoda **sessionStorage.clear()**:

```
sessionStorage.clear();
```

Un exemplu de utilizare a **sessionStorage** ar fi stocarea preferințelor utilizatorului în cadrul sesiunii curente. De exemplu, să presupunem că un utilizator accesează un site care permite vizualizarea și sortarea produselor după anumite criterii, cum ar fi prețul, data adăugării sau popularitatea. Utilizatorul poate selecta una sau mai multe opțiuni de sortare și preferințele lor ar trebui să fie păstrate și aplicate pe toate paginile site-ului pe care le vizitează în cadrul aceleiași sesiuni de navigare.

Pentru a realiza acest lucru, se poate utiliza **sessionStorage**. De exemplu, să presupunem că utilizatorul selectează sortarea după preț. Codul JavaScript pentru a stoca această preferință în **sessionStorage** ar arăta astfel:

Probleme practice

1. ***Sarcină practică*** Creați o pagină web care afișează toate locațiile vizitate de utilizator în ultima săptămână, utilizând obiectul **history** și **localStorage** pentru a stoca și afișa datele.
2. ***Sarcină practică*** Implementați un sistem de autentificare pentru o aplicație web, utilizând cookie-uri și obiectul **localStorage** pentru a stoca datele utilizatorului.
3. ***Sarcină practică*** Creați o pagină web care utilizează obiectul **navigator** pentru a afișa informații despre browser-ul utilizatorului și caracteristicile acestuia.
4. ***Sarcină practică*** Implementați un sistem de cos de cumpărături pentru o aplicație web, utilizând obiectul **localStorage** pentru a stoca și afișa produsele selectate de utilizator.
5.

QUIZ

1. Ce este obiectul **location** în JavaScript? A. Obiectul care reprezintă fereastra curentă a browser-ului. B. Obiectul care conține istoricul browser-ului. C. Obiectul care conține informații despre adresa URL a paginii curente.

Răspuns corect: C

2. Ce este **localStorage** în JavaScript? A. O metodă pentru a stoca date pe server. B. Un obiect care permite stocarea de date în browser-ul utilizatorului. C. Un mod de a afișa informații despre utilizatorul curent.

Răspuns corect: B

3. Cum se poate stoca un obiect JavaScript în **localStorage**? A. Prin utilizarea metodei **localStorage.store()**. B. Prin utilizarea metodei **JSON.stringify()** și apoi prin salvarea rezultatului în **localStorage**. C. Prin utilizarea metodei **localStorage.setItem()**.

Răspuns corect: B

4. Ce poate fi stocat în **localStorage**? A. Numai șiruri de caractere. B. Numai numere. C. Orice tip de date suportat de JavaScript.

Răspuns corect: C

5. Ce face obiectul **screen** în JavaScript? A. Afișează informații despre ecranul utilizatorului. B. Permite manipularea ferestrelor din browser. C. Permite accesul la istoricul browser-ului.

Răspuns corect: A

6. Ce face obiectul **navigator** în JavaScript? A. Permite manipularea ferestrelor din browser. B. Afișează informații despre browser-ul și sistemul de operare al utilizatorului. C. Permite accesul la istoricul browser-ului.

Răspuns corect: B

7. Cum se poate verifica dacă browser-ul acceptă **localStorage**? A. Prin utilizarea metodei **localStorage.supported()**. B. Prin verificarea dacă **localStorage** este diferit de **undefined**. C. Prin utilizarea metodei **typeof localStorage !== "undefined"**.

Răspuns corect: C

8. Ce face obiectul **history** în JavaScript? A. Permite manipularea ferestrelor din browser. B. Afișează informații despre browser-ul și sistemul de operare al utilizatorului. C. Conține informații despre istoricul de navigare al utilizatorului.

Răspuns corect: C

9. Ce face obiectul **Window** în JavaScript? A. Afișează informații despre ecranul utilizatorului. B. Permite manipularea ferestrelor din browser. C. Permite accesul la istoricul browser-ului.

Răspuns corect: B

10. Ce face obiectul **Cookies** în JavaScript? A. Permite manipularea cookie-urilor browser-ului. B. Afișează informații despre browser-ul și sistemul de operare al utilizatorului. C. Permite accesul la istoricul browser-ului.

Răspuns corect: A

Capitol VIII. Propuneri de proiecte finale pentru rezolvare

Titlu proiect 1: Platformă de gestionare a sarcinilor utilizând JavaScript și MySQL

Specificatii tehnice:

1. **Front-end**:

   a. Dezvoltarea interfeței utilizator (UI) folosind HTML, CSS și JavaScript.

   b. Implementarea logicii front-end utilizând JavaScript fără utilizarea unui framework sau bibliotecă suplimentară. c. Asigurarea unui design atractiv și responsiv, care să se adapteze la diferite dimensiuni de ecran și dispozitive, prin utilizarea CSS și media queries. d. Utilizarea event handling-ului pentru a gestiona interacțiunea utilizatorului cu aplicația. e. Manipularea DOM-ului în mod dinamic pentru a reflecta modificările și acțiunile utilizatorului.

2. **Back-end**:

   a. Utilizarea Node.js ca mediu de execuție JavaScript pe server.

   b. Implementarea unui server HTTP pentru a gestiona cererile de la client și a oferi răspunsurile corespunzătoare.

   c. Utilizarea bibliotecilor native Node.js pentru a gestiona rutele și cererile HTTP.

   d. Integrarea bazei de date MySQL pentru a stoca informațiile despre sarcini și utilizatori.

   e. Utilizarea unui modul Node.js pentru a efectua operațiile de interogare și manipulare a datelor în baza de date MySQL.

3. **Funcționalități principale**:

   a. Înregistrarea și autentificarea utilizatorilor.

   b. Adăugarea, modificarea și ștergerea sarcinilor de către utilizatori.

   c. Atribuirea sarcinilor utilizatorilor specifici.

d. Sortarea și filtrarea sarcinilor în funcție de diferite criterii, cum ar fi termenul limită sau prioritatea.

e. Notificarea utilizatorilor despre sarcinile atribuite sau termenele limită iminente prin e-mail sau alte canale de comunicare.

f. Generarea de rapoarte și statistici legate de sarcinile gestionate.

4. **Cerințe tehnice**:

a. Platforma trebuie să fie compatibilă cu cele mai recente versiuni ale principalelor browsere web, cum ar fi Chrome, Firefox, Safari și Edge.

b. Interfața utilizator trebuie să fie intuitivă și ușor de utilizat, oferind o experiență fluidă utilizatorilor.

c. Aplicația trebuie să fie scalabilă, permițând gestionarea unui număr mare de utilizatori și sarcini.

d. Se recomandă utilizarea design patterns și best practices în dezvoltarea aplicației pentru a asigura un cod modular, ușor de întreținut și extensibil.

e. Asigurarea securității datelor și protejarea împotriva atacurilor comune, cum ar fi injecția de cod sau cross-site scripting (XSS).

f. Documentarea completă a codului sursă și a tuturor funcționalităților implementate.

Pentru integrarea bazei de date MySQL, vei avea nevoie de un modul Node.js pentru a interacționa cu baza de date. În cadrul logicii back-end, vei utiliza acest modul pentru a realiza operațiile de interogare și manipulare a datelor în baza de date MySQL, cum ar fi inserarea, actualizarea, ștergerea și interogarea datelor. Asigură-te că respecti principiile de securitate și eviți vulnerabilitățile comune la nivel de baza de date, cum ar fi SQL injection.

Titlu proiect 2: Aplicație de notițe utilizând JavaScript și MySQL

Specificatii tehnice:

1. **Front-end**: a. Dezvoltarea interfeței utilizator (UI) folosind HTML, CSS și JavaScript pur. b. Implementarea logicii front-end utilizând JavaScript fără utilizarea unui framework sau bibliotecă suplimentară. c. Manipularea DOM-ului în mod dinamic pentru a afișa și interacționa cu notițele utilizatorului. d. Utilizarea event handling-ului pentru a gestiona acțiunile utilizatorului, cum ar fi adăugarea, modificarea și ștergerea notițelor.

2. **Back-end**: a. Utilizarea Node.js ca mediu de execuție JavaScript pe server. b. Implementarea unui server HTTP pentru a gestiona cererile de la client și a oferi răspunsurile corespunzătoare. c. Utilizarea bibliotecilor native Node.js pentru a gestiona rutele și cererile HTTP. d. Integrarea bazei de date MySQL pentru a stoca și gestiona notițele utilizatorului.

3. **Funcționalități principale**: a. Înregistrarea și autentificarea utilizatorilor. b. Adăugarea, modificarea și ștergerea notițelor de către utilizatori. c. Căutarea și filtrarea notițelor în funcție de diferite criterii, cum ar fi titlul sau data de creare. d. Organizarea notițelor în categorii sau etichete pentru o gestionare mai eficientă. e. Exportarea sau partajarea notițelor în diverse formate, cum ar fi fișiere text sau PDF. f. Posibilitatea de a adăuga imagini sau atașamente în cadrul notițelor.

4. **Cerințe tehnice**: a. Platforma trebuie să fie compatibilă cu cele mai recente versiuni ale principalelor browsere web, cum ar fi Chrome, Firefox, Safari și Edge. b. Interfața utilizator trebuie să fie intuitivă și ușor de utilizat, oferind o experiență plăcută utilizatorilor. c. Aplicația trebuie să fie scalabilă, permițând gestionarea unui număr mare de utilizatori și notițe. d. Asigurarea securității datelor și protejarea împotriva vulnerabilităților comune la nivel de bază de date, cum ar fi SQL injection. e. Documentarea completă a codului sursă și a tuturor funcționalităților implementate.

Aceste specificații și cerințe tehnice reprezintă o orientare generală pentru elaborarea unui proiect final bazat pe JavaScript pur și utilizarea bazei de date MySQL pentru dezvoltarea unei aplicații de notițe. Asigură-te că respecți cele mai bune practici de dezvoltare și securitate pentru a crea o aplicație robustă și eficientă.

Titlu proiect 3: Aplicație de gestionare a produselor cu JavaScript și MySQL

Specificatii tehnice:

1. **Front-end**: a. Dezvoltarea interfeței utilizator (UI) folosind HTML, CSS și JavaScript pur. b. Implementarea logicii front-end utilizând JavaScript fără utilizarea unui framework sau bibliotecă suplimentară. c. Manipularea DOM-ului pentru a afișa și interacționa cu informațiile despre produse. d. Utilizarea event handling-ului pentru a gestiona acțiunile utilizatorului, cum ar fi adăugarea, modificarea și ștergerea produselor.

2. **Back-end**: a. Utilizarea Node.js ca mediu de execuție JavaScript pe server. b. Implementarea unui server HTTP pentru a gestiona cererile de la client și a oferi răspunsurile corespunzătoare. c. Utilizarea bibliotecilor native Node.js pentru a gestiona rutele și cererile HTTP. d. Integrarea bazei de date MySQL pentru a stoca și gestiona informațiile despre produse.

3. **Funcționalități principale**: a. Adăugarea, modificarea și ștergerea produselor. b. Afișarea listei de produse existente și a detaliilor acestora. c. Căutarea și filtrarea produselor în funcție de diferite criterii, cum ar fi numele, categorie sau preț. d. Implementarea funcționalității de coș de cumpărături pentru utilizatori. e. Generarea de rapoarte și statistici legate de produsele gestionate.

4. **Cerințe tehnice**: a. Platforma trebuie să fie compatibilă cu cele mai recente versiuni ale principalelor browsere web, cum ar fi Chrome, Firefox, Safari și Edge. b. Interfața utilizator trebuie să fie intuitivă și ușor de utilizat, oferind o experiență plăcută utilizatorilor. c. Aplicația trebuie să fie scalabilă, permițând gestionarea unui număr mare de produse și utilizatori. d. Asigurarea securității datelor și protejarea împotriva vulnerabilităților comune la nivel de bază de date, cum ar fi SQL injection. e. Documentarea completă a codului sursă și a tuturor funcționalităților implementate.

Aceste specificații și cerințe tehnice reprezintă o orientare generală pentru elaborarea unui proiect final bazat pe JavaScript pur și utilizarea bazei de date MySQL pentru dezvoltarea unei aplicații de gestionare a produselor. Asigură-te că respecți cele mai bune practici de dezvoltare și securitate pentru a crea o aplicație robustă și eficientă.

Titlu proiect 3: Aplicație de gestionare a bugetului personal cu JavaScript pur și SQLite

Specificatii tehnice:

1. **Front-end**: a. Dezvoltarea interfeței utilizator (UI) folosind HTML, CSS și JavaScript pur. b. Implementarea logicii front-end utilizând JavaScript fără utilizarea unui framework sau bibliotecă suplimentară. c. Manipularea DOM-ului în mod dinamic pentru a afișa și interacționa cu informațiile despre buget. d. Utilizarea event handling-ului pentru a gestiona acțiunile utilizatorului, cum ar fi adăugarea, modificarea și ștergerea tranzacțiilor.

2. **Back-end**: a. Utilizarea Node.js ca mediu de execuție JavaScript pe server. b. Implementarea unui server HTTP pentru a gestiona cererile de la client și a oferi răspunsurile corespunzătoare. c. Utilizarea bibliotecilor native Node.js pentru a gestiona rutele și cererile HTTP. d. Integrarea bazei de date SQLite pentru a stoca și gestiona informațiile despre buget și tranzacții.

3. **Funcționalități principale**: a. Înregistrarea și autentificarea utilizatorilor. b. Adăugarea, modificarea și ștergerea tranzacțiilor de către utilizatori. c. Afișarea bilanțului și rapoartelor financiare, inclusiv cheltuieli, venituri și sold curent. d. Planificarea bugetului lunar și urmărirea cheltuielilor pe categorii. e. Generarea de grafice și diagrame pentru o vizualizare mai clară a datelor financiare. f. Notificarea utilizatorului în cazul atingerii unor limite prestabilite pentru anumite categorii de cheltuieli.

4. **Cerințe tehnice**: a. Platforma trebuie să fie compatibilă cu cele mai recente versiuni ale principalelor browsere web, cum ar fi Chrome, Firefox, Safari și Edge. b. Interfața utilizator trebuie să fie intuitivă și ușor de utilizat, oferind o experiență plăcută utilizatorilor. c. Aplicația trebuie să fie scalabilă, permițând gestionarea unui număr mare de tranzacții și utilizatori. d. Asigurarea securității datelor și protejarea împotriva vulnerabilităților comune la nivel de bază de date. e. Documentarea completă a codului sursă și a tuturor funcționalităților implementate.

Aceste specificații și cerințe tehnice reprezintă o orientare generală pentru elaborarea unui proiect final bazat pe JavaScript pur și utilizarea bazei de date SQLite pentru dezvoltarea unei aplicații de gestionare a bugetului personal. Asigură-te că respecți cele mai bune practici de dezvoltare și securitate pentru a crea o aplicație robustă și eficientă.

Resursele didactice recomandate elevilor

Nr	Nume resursă	Link-ul către resursă
1.	Documentația de baza	https://developer.mozilla.org/en-US/docs/Web/JavaScript
2.	JavaScript Info	https://javascript.info/
3.	W3 Schools	https://www.w3schools.com/js/default.asp
4.	Codecademy	https://www.codecademy.com/catalog/language/javascript
5.	Web Dev	https://web.dev/learn/javascript