Univ. Babeș-Bolyai,

Facultatea de Matematică și Informatică

Lect. dr. Darius Bufnea

Notițe de curs Programare Web: JavaScript (săptămâna 6 și 7 de școală)

Pe lângă prezentul material legat de limbajul JavaScript, studenți sunt rugați „ferm" să parcurgă si următorul material: http://www.w3schools.com/js/default.asp (toate secțiunile din stânga până la JS AJAX (fără JS AJAX) plus JS Examples).

JavaScript s-a născut ca un limbaj interpretat client-side, inițial destinat browser-elor. A fost dezvoltat de Brendan Eich de la Netscape (Netscape fiind considerat bunicul „Firefox"), inițial denumit LiveScript. A apărut în decembrie 1995, sub denumirea de JavaScript, denumire data în urma unui agreement între Sun Microsystems (inventatoarea Java) și Netscape.

Paranteză: La acea vreme Sun dorea să își popularizeze applet-urile Java ca tehnologie client-side (in confruntare directă cu Macromedia Flash) și pentru acest lucru acordul cu Netscape prevedea ca browser-ul celor de la Netscape (Netscape Navigator) să suporte mașina virtuala Java care să permite rularea applet-urilor Java, iar in schimb Netscape să poate folosi in denumirea limbajului cuvantul „Java" pentru a face limbajul ce avea să fie cunoscut mai târziu sub denumirea de JavaScript mai popular. In timp, applet-urile Java au pierdut lupta cu Macromedia Flash aceasta din urmă impunându-se ca și tehnologie pe frontend, iar și mai tarziu Flash-ul (tehnologie ajunsa între timp in ograda Adobe) a pierdut definitiv „războiul" in favoarea JavaScript.

La început limbajul a fost destul de nestandardizat, existând mai multe variante, printre cele mai populare numărându-se:

- JavaScript – varianta Netscape
- JScript – varianta Microsoft implementata de primele versiuni de Internet Explorer
- ActionScript – varianta Macromedia – in perioada in care animațiile Flash ca tehnologie client-side erau la moda, ActionScript-ul fiind limbajul de facto pentru Macromedia Flash și mai târziu pentru Adobe Flash (când Adobe a cumpărat Macromedia)

Toate aceste „variante" (sau dialecte) s-au dovedit a fi o corvoadă pentru programator – nu erau rare situațiile in care trebuiau scrise variante de „script" diferite pentru a suporta multiple browsere (spre exemplu o variantă de script pentru Internet Explorer și o varianta de script pentru Netscape Navigator). S-a ajuns la situația / nevoia de a standardiza acest limbaj în ceea ce se numește ECMAScript.

Deși inițial limbajul a fost gândit pentru a fi folosit client-side (adică a fi rulat de către browser-e), in ultimul timp se folosește și ca tehnologie pe backend (NodeJS).

Ce este / ce nu este și ce se poate face in JavaScript?

Ce este:

- limbaj interpretat de către browserul Web (client-side);
- ca limbaj urmează mai multe paradigme: imperativ, funcțional, orientat obiect, case sensitive, orientat pe evenimente (event-driven).

Ce permite:

- interacțiunea cu pagina (documentul HTML);
- modificarea dinamică a conținutului documentului, crearea de noi elemente (tag-uri) in cadrul pagini, ștergerea unor elemente (tag-uri), modificarea atributelor HTML (adăugarea, ștergerea, schimbarea valorilor) a unor anumite tag-uri. Se permite astfel crearea de conținut dinamic in cadrul documentului HTML / crearea de documente HTML dinamice la nivelul clientului (DHTML);
- permite execuția anumitor funcții la apariția anumitor evenimente (interactivitate cu utilizatorul);
- validări pe frontend – spre exemplu validarea datelor introduse în câmpurile unui formular (observație: validările pe front-end sunt doar „de dragul" de a face pagina / interfața cât mai user-friendly. Validările de pe frontend trebuie OBLIGATORIU însoțite și de validări pe backend care sunt VITALE din punct de vedere al securității).

Ce nu este JavaScript:

- Nu este un limbaj înrudit cu limbajul Java - in afara de nume, singura legătura este sintaxa C comună specifică ambelor limbaje;
- nu este un limbaj de programare strong typed, e weakly typed - o variabilă poate primi inițial ca valoare un număr întreg, iar ulterior un string (în timp ce JavaScript e considerat weakly typed, Java este strongly type).

Inserarea codului JavaScript in documentul HTML

Codul JavaScript în interiorul documentului HTML se inserează cu ajutorului tag-ului `script`, fie in interiorul tag-ului, fie specificat cu ajutorul atributului `src` a acestui tag:

```
<script type="text/javascript">
… cod JavaScript …
</script>
```

sau

```
<script type="text/javascript" src="myscript.js">
</script>
```

Observații:

- atributul `src` poate indica spre un fișier local cu extensia .js (găzduit în același loc ca și documentul HTML) sau spre un URL absolut (care începe cu http:// sau https://);
- dacă, codul JavaScript se specifică cu ajutorul atributului `src`, nu uitați să închideți tag-ul script (trebuie sa apară și marcajul de sfârșit de tag, fiind un tag cu corp).

In unele situații este util după folosirea tag-ului script, folosirea tag-ului `noscript`, util in situații în care browser-ul nu suportă JavaScript din diverse motive – puține probabile în prezent (browser vechi, browser pentru dispozitive mobile mai vechi, engine-ul JavaScript al browserului este dezactivat). Exemplu:

```
<script>
   alert("Hello World!");
</script>
<noscript>I can't say hello because your browser doesn't support JavaScript!</noscript>
```

O practică des întâlnită pentru a preîntâmpina unele erori pe unele browser-e incapabile sa ruleze cod JavaScript este plasarea codului JavaScript în interiorul unui comentariu HTML `<!-- -->`. Exemplu:

```
<script type="text/javascript">
   <!--
   // începutul codului JavaScript
   alert("Hello World!");
   /*
   sfârșitul codului JavaScript
   */
   -->
</script>
```

Observație: pe exemplu de mai sus am exemplificat și folosirea comentariilor in limbajul JavaScript. Acestea sunt similar cu cele din limbajele C/C++/Java, respective folosire `//` pentru comentariu single-line și `/* */` pentru comentarii multi-line.

Localizarea codului JavaScript in cadrul pagini HTML

In mod tradițional, codul JavaScript era plasat în cadrul unui tag `<script>` in secțiunea `<head></head>` a documentului HTML. Acest lucru nu este obligatoriu, tag-ul `<script>` putând fi plasat oriunde in cadrul documentului.

IMPORTANT: CAT TIMP BROWSER-UL EXECUTA COD JAVA SCRIPT NU „FACE NIMIC ALTCEVA” – NU (RE)RANDEAZA PAGINA, NU INTERACTIONEAZA CU UTILIZATORUL (NU RAPSUNDE LA COMENZI/CLICK-URI DE MOUSE ETC).

Exemplu:

```
<a href="http://www.google.com">Click me</a>
<script>
while(1);
```

```
</script>
```

Cateva recomandari tinand cont de observatia importanta de mai sus:

- executia secventelor de cod JavaScript e important sa dureze cat mai putin pentru a da „sansa” browserului sa re(randeze) pagina si sa raspunda la input-ul utilizatorului;
- in sectiunea head pot fi incluse definitii de functii sau incluse fisiere .js care contin definitii de funcții JavaScript (aceste funcții nu se executa chiar atunci, sunt doar definite – vedem mai târziu ce se întâmplă cu ele);
- se recomandă pe cât posibil modelul de apel asincron a unor funcții: nu se apelează funcția direct așteptându-se („timp mort”) terminarea ei pentru obținerea rezultatului, ci se va seta apelul funcției ca funcție de „callback” la apariția unui anumit eveniment. Important în acest context e ca funcția dorită a fi apelată nu se executa instant, iar valoarea returnată de aceasta nu este disponibilă imediat, ci doar mai târziu când se termină de executat funcția – moment în care se poate utiliza și valoarea returnată de această funcție.
- crearea de elemente (tag-uri) `<script>` dinamic care sa se executa ulterior după randarea documentului HTML (o astfel de logica poate fi plasata la sfârșitul documentului HTML) – mai târziu un exemplu in acest sens.

Exemplul 1:

```
<html><head><title>Exemplul 1</title>
<script type="text/javascript">
    function clickMe() { // executia functiei nu are loc acum
        alert('Hello world');
    }
</script>
</head>
<body>
    <a href="javascript:clickMe();">Click here</a>
</body></html>
```

Exemplul 2:

```
<html>
    <head><title>Exemplul 2 JavaScript</title></head>
<body>
    Astazi este:
    <script type="text/javascript">
        document.write(new Date()); // executia are loc acum
    </script>
</body>
</html>
```

Elemente de bază ale limbajului JavaScript: Funcții, variabile, tablouri, obiecte, instrucțiuni de control JavaScript

Funcții

O funcție JavaScript se definește cu cuvântul rezervat `function` și poate fi apelată oricând după definiția acesteia:

```
<script>
function sum(a, b) {
    return a + b;
```

```
}
alert(sum(1,2));
</script>
```

Returnarea unei valori se face cu `return` (asemănător cu C/C++/Java), iar în lista parametrilor formali (`a` și `b` în exemplul de mai sus) aceștia nu trebuie sa aibă declarat un tip.

In JavaScript este deosebit de uzuală folosirea funcțiilor anonime. Exemplu:

```
element.onclick = function () {
    alert('Aceasta este o functie anonima');
}
```

O funcție anonimă poate fi chiar și apelată după definirea ei:

```
<script>
(function (a, b) { // a si b sunt parametrii formali ai functiei
    // functie anonima care afiseaza suma a doua numere
    var s = a + b;
    alert(s);
})(2, 3); // apelul cu parametrii actuali ai functiei
</script>
```

Variabile și tipuri in JavaScript

Variabilele JavaScript se declară cu cuvântul rezervat `var`. Tipul acestora este nedefinit (de fapt tipul unei variabile este dat de tipul valorii asociate acesteia). Astfel, o variabilă poate primi la un moment dat ca și valoare un număr, iar ulterior un șir de caractere – JavaScript fiind considerat un limbaj *weakly* și *dynamically typed*.

Exemplu:

```
var i = 7;
i = 'Ana are mere';
var s = "Cocosul canta";
// șirurile de caractere pot fi delimitate atât cu ' cat și cu "
s = 3.1415;
var c = true;
```

Printre tipurile de valori pe care le pot fi atribute unei variabile sunt: `number`, `string`, `boolean`, `object`, `function` (funcțiile sunt de fapt niște obiecte mai speciale), `undefined`.

Exerciții: Pentru a vă familiariza cu tipurile din JavaScript, operatorul `typeof`, precum și cu câteva funcții de conversie precum `eval()`, `Number()`, `String()`, puteți încerca să rulați următoarele linii de cod în consola JavaScript a browserului preferat. Consola JavaScript este accesibilă sub forma unui tab separat în cadrul *Developer Tools*-ului din cadrul browserului (F12 în orice browser – dar mă aștept să știți acest lucru daca ați făcut debugging la laboratorul de CSS ☺).

```
typeof 1
typeof 1.5
```

```
typeof '1.5'
typeof eval('1.5')
typeof Number('1.5')
typeof String(2)
typeof 'Ana are mere'
typeof "Cocosul canta"
typeof true
typeof {}
typeof x
f = function (){ return 2;}
typeof f
typeof [1, 2, 3, 4, 5]
punct = { x: 7, y: 9}
typeof punct
1/0
typeof Infinity
typeof 1/0
```

```
>> typeof 1
<- "number"
>> typeof 1.5
<- "number"
>> typeof '1.5'
<- "string"
>> typeof eval('1.5')
<- "number"
>> typeof Number('1.5')
<- "number"
>> typeof String(2)
<- "string"
>> typeof 'Ana are mere'
<- "string"
>> typeof "Cocosul canta"
<- "string"
>> typeof true
<- "boolean"
>> typeof {}
<- "object"
```

```
>> typeof x
<- "undefined"
>> f = function (){ return 2;}
<- ▸ function f()
>> typeof f
<- "function"
>> typeof [1, 2, 3, 4, 5]
<- "object"
>> punct = { x: 7, y: 9}
<- ▸ Object { x: 7, y: 9 }
>> typeof punct
<- "object"
>> 1/0
<- Infinity
>> typeof Infinity
<- "number"
>> typeof 1/0
<- NaN
```

Întrebare: Dacă `1/0` este `Infinity` și `typeof Infinity` este `number`, de ce `typeof 1/0` este `NaN` (Not a Number)? Si dacă asta vi se pare simpla ☺, găsiți alte „problemuțe drăguțe” specifice limbajului JavaScript aici.

In unele contexte, folosirea funcției `eval` este periculoasă (este posibil sa primiți un mesaj de eroare la execuția ei). `eval` în JavaScript face mult mai mult decât să convertească un string la număr, `eval` poate sa evalueze inclusiv o secvență de cod (adică să o execute).

Exemplu:

```
var s = "alert('Hello World')"; // acesta este un string
eval(s);
```

Observații:

- Variabilele pot fi declarate în JavaScript și cu cuvântul rezervat `let`, mai multe despre acesta mai târziu in acest material.
- exista limbaje „derivate" din JavaScript (sau mai degrabă construite peste JavaScript), a căror cod se compilează/translatează în cod JavaScript și care sunt *strongly typed*. Un exemplu in acest sens este TypeScript.

Tablouri in JavaScript

Tablourile in JavaScript sunt de fapt niște obiecte mai speciale. Prezentam mai jos câteva modalități de declarare a acestora:

```
var tari = new Array();
tari[0] = 'Romania';
tari[1] = 'Franta';
// tari.length este 2
tari[6] = 'Germania';
// tari.length este 7, cu 4 elemente ale tabloului tari undefined
var x=[1, 2, 3, 4]; // se declara un Array cu cele 4 elemente
var y = new Array(5, 6, 7, 8);
// la fel, se declara un Array cu cele 4 elemente
var z = new Array(11); // se declara un Array cu 11 elemente, toate undefined
```

Observații:

- lungimea unui tablou (`Array`) poate fi aflată prin intermediul proprietății `.length` a unui array. Atenție, aceasta se comporta ca o dată membră publică (în accepțiunea OOP), nu ca o metodă ce se va invoca cu: `tari.length()` – (paranteză: de fapt proprietatea `.length` este implementată folosind o metodă getter – mai multe detalii despre metodele getter și setter in JavaScript găsiți aici).
- Observații comportamentul diferit a constructorului `new Array()` in funcție de numărul de parametrii. Apelat cu un parametru, `new Array(11)` declară un tablou cu 11 elemente neinițializate (`undefined`), apelat cu mai mulți parametrii, spre exemplu `new Array(5, 6, 7, 8)` declară un `Array` cu 4 elemente inițializate cu valorile specificate;
- elementele unui `Array` pot sa fie de tipuri diferite. Exemplu:

```
var varza = new Array(1, 'Covid-19', false, {x:5, y:9});
```

Elementele tabloului de mai sus sunt de tip `number`, `string`, `boolean` și respetiv `object`.

Tablouri multidimensionale

JavaScript acceptă și tablouri multidimensionale. Spre exemplu, o matrice de 3x3 cu elemente întregi (tablou bidimensional) poate fi declarată astfel:

```
M = new Array(new Array(1, 2, 3), new Array(4, 5, 6), new Array(7, 8, 9));
// M.length va avea lungimea 3
// M[1][1] va avea valoarea 5
```

Elementele unui tablou multidimensional se accesează conform notației clasice folosind indecși numerici din limbajele C/C++/Java (spre exemplu `M[i][j]`). Pe exemplu de mai sus M este de fapt un `array` cu 3 elemente, fiecare element la rândul sau fiind un `array` cu alte 3 elemente.

Funcții uzuale de lucru cu tablouri in JavaScript

API-ul JavaScript oferă o serie de operații ce se pot efectua pe un `Array`. Lista completă a acestora este disponibilă aici. Printre cele mai populare operații (vă recomand totuși să vă uitați peste lista completă) sunt următoarele:

- `push(lista_elemente)` – adaugă un nou element (sau elementele) la sfârșitul unui tablou și returnează noua lungime a acestuia;
- `pop()` – șterge ultimul element dintr-un tablou și returnează elementul șters;
- `shift()` – șterge primul element al unui tablou și returnează acest element;
- `unshift(lista_elemente)` – inserează elementul (sau elementele) la începutul tabloului și returnează noua lungime a acestuia;
- `slice(poz, nrElemente)` – extrage un subsir de `nrElemente` începând cu poziția `poz`. Tablou pe care este apelată nu se modifică;
- `splice(poz, nrElemente, lista_elemente)` – șterge începând de la poziția `poz` un număr de `nrElemente` din tablou, returnând elementele șterse. Inserează pe poziția `poz` elemente din `lista_elemente`. Dacă este apelată doar cu doi parametrii (`listă_elemente` lipsește), șterge doar elementele din tablou (atenție!, spre deosebire de `slice` modifică tabloul pe care este apelată). Daca `nrElemente` este 0 (adică nu se dorește ștergerea niciunui element), `splice` este practic folosită pentru a insera elemente într-un tablou.
- `indexOf(elem)` – returnează indexul primei apariții a elementului in tablou sau -1 dacă acesta nu este găsit;
- `lastIndexOf(elem)` – returnează indexul ultimei apariții a elementului in tablou sau -1 dacă acesta nu este găsit;
- `isArray()` – returnează dacă un obiect este sau nu tablou (`true` sau `false`);
- `forEach(f)` – iterează elementele unui tablou și execută funcția `f` pentru fiecare element al acestuia

- `sort()` – ordonează un tablou. Funcției `sort` i se poate da ca parametru inclusiv o funcție care specifică cum ar trebui să fie comparate 2 elemente ale tabloului (utilă mai ales în situația în care elementele tabloului ce se dorește a fi sortat nu sunt elemente de tip `numeric` sau `string` ci obiecte mai complexe);
- `reverse()` – inversează ordinea elementelor unui tablou.

Obiecte JavaScript

Un obiect in JavaScript poate fi văzut ca o colecție neordonată de date (valori) ce pot avea tipuri diferite, dar împreună au o anumită semantică – spre exemplu "datele" despre o persoană și acțiunile întreprinse de persoana respectivă. Pentru a accesa fiecare dată / valoare din cadrul unui obiect este nevoie și de o cheie, astfel putem privi un obiect și ca o colecție de perechi (cheie, valoare). Este mai natural să ne referim la cheile cu care se accesează datele unui obiect cu numele de atribut-ul obiectului sau proprietatea obiectului, valorile asociate acestora putând fi primitive numerice, boolean-e, string-uri, dar și referințe la alte obiecte, tablouri sau funcții (acestea două din urmă fiind tot obiecte).

Exemplu:

```
var person = {
    name: 'Chuck Norris',
    strength : Infinity,
}
```

Proprietățile obiectului de mai sus sunt `name` și `strength` iar valorile asociate acestora sunt de tip `string`, și `number`. Unui obiect pot să îi fie atribuite proprietăți și mai târziu, astfel putem să-l facem pe Chuck Norris oricând nemuritor:

```
person.immortal = true;
```

Proprietățile unui obiect pot să primească ca și valori funcții, astfel adăugăm metode obiectului respectiv:

```
person.kick = function() {
    this.opponents = null;
}
```

In acest moment, obiectul dat ca exemplu, are un nume (`'Chuck Norris'`), putere (`Infinity`), este nemuritor (are proprietatea `immortal` setată la valoarea `true`) și prezintă o metodă numită `kick` care însă nu a fost apelată. În cadrul acestei metode, `this` (cuvânt rezervat) indică spre obiectul pe care se va apela funcția, proprietatea `opponents` adăugându-i-se acestuia (obiectul încă nu are aceasta proprietate, ea va fi adăugată la apelul metodei).

Dacă Chuck Norris dă cu piciorul, adică dacă se invocă metoda `kick` pe acest obiect:

```
person.kick();
```

obiectul dat ca exemplu va avea o proprietate nouă numită `opponents` cu valoarea `null` (Chuck Norris anihilându-și toți adversarii cum e și normal).

Proprietățile unui obiect pot să fie accesate și folosind o notație de forma (a se observa asemănarea dintre obiecte și Array-uri):

```
person["name"] // va returna 'Chuck Norris'
```

O proprietate pe un obiect poate fi și ștearsă cu (dacă Chuck Norris „o ia in freză”):

```
delete person.immortal
```

Revenind la tablouri, am specificat anterior că acestea sunt tot obiecte. Valorile memorate în cadrul unui obiect de tip tablou putând fi accesate prin intermediul indecșilor numerici (`x[0], x[1], x[2]…`), chiar și o expresie de forma `x["1"]` fiind corectă (nu și una de forma `x.1`).

Puțin mai târziu în cadrul acestui document vom relua discuția despre obiectele JavaScript după ce discutăm de scop global.

Instrucțiuni de control

Instrucțiunile de control `while`, `for`, `if` sunt identice cu cele din limbajele C/C++/Java. Insistăm însă pe o variantă de `for` care permite iterarea elementelor unui tablou sau a proprietăților unui obiect. Spre exemplu, pentru a vedea care sunt proprietățile obiectului `person` declarat mai sus, le putem itera cu (a se rula codul în consola JavaScript din *Developer Tools*):

```
for (i in person)
    console.log(i + ' are valoarea ' + person[i]);
```

Observație: In exemplu de mai sus valorile proprietăților iterate pot fi accesate cu o expresie de forma `person[i]` dar nu cu o expresie de forma `person.i` (aceasta din urmă s-ar referi la o proprietate `i` pe care are avea-o obiectul `person`, proprietate pe care obiectul nu o are).

In același, mod pot fi iterate elementele unui tablou:

```
x = [5, 6, 7];
x[10] = 0;
for (i in x)
    console.log('x[' + i + '] = ' + x[i]);
```

Mai târziu în acest material vom itera proprietățile a două obiecte importante JavaScript: `window` și `document`.

In contextul folosirii dese a unui obiect, se poate folosi instrucțiunea `with` pentru a simplifica codul și a nu repeta folosirea numelui obiectului. Spre exemplu, dacă dorim să vedem câți inamici are Chuck Norris după ce lovește fulgerător, putem folosi:

```
with(person) {
```

```
    kick();
    console.log(opponents);
}
```

In secțiunea de față a prezentului material mai insistăm pe folosirea operatorului de comparație `===` (față de folosirea operatorului `==`). Ambii operatori se folosesc pentru testarea egalității a două valori, dar `===` verifică în plus (pe lângă faptul că cele două expresii sunt evaluate la aceeași valoare) și faptul că cele două valori sunt de același tip. Un astfel de operator este in general specific limbajelor *weakly-typed* (mai este prezent de exemplu în PHP) și nu se regăsește in limbajele *strongly-typed*. Exemplu:

```
if (1 == true) alert('Sunt egale'); else alert ('Nu sunt egale');
// se afișează Sunt egale, pentru că true ca valoare de adevăr este evaluată
// la valoarea numerică 1
if (1 === true) alert('Sunt egale'); else alert ('Nu sunt egale');
// se afișează Nu sunt egale pentru că cele doua expresii au tipuri diferite,
// number, respectiv boolean
```

Obiectul window, scop global, DOM, manipularea DOM-ului

În exemplele date până în prezent în materialul de față am folosit unele funcții precum `alert()` sau obiecte precum `document` care par predefinite. Acestea nu sunt predefinite, ele de fapt există ca funcții membre, respectiv date (proprietăți) membre în cadrul unui obiect global numit `window` care abstractizează fereastra browser-ului. Nu este greșit de exemplu să folosim expresii de forma `window.alert()` sau `window.document`, dar specificarea explicită a obiectului `window` este redundantă.

Un exercițiu interesant este iterarea (folosind forma instrucțiunii `for` prezentată anterior) datelor și funcțiilor membre (proprietățile) ale obiectelor `window` și `document`. Astfel, pentru a realiza un fel de introspecție pe obiectul `window`, puteți încerca în consola JavaScript a browser-ului:

```
for (i in window) console.log(i + '=' + window[i]);
```

Puteți observa pe obiectul `window` existența unor date membre/proprietăți precum `document` sau `outerWidth` precum și a unor funcții membre precum `alert` sau `setTimeout`. Identic, se poate face introspecție pe `document`:

```
for (i in document) console.log(i + '=' + document[i]);
```

Aruncați in mare o privire peste tot ce „are” documentul ca proprietăți. Exemple de proprietăți ale obiectului `document` care ar trebui să vă fie intuitive: `title` sau `location`.

Toate variabilele și funcțiile care se declară in interiorul unui tag `<script>` spunem că sunt declarate in scopul global (ele sunt accesibile de oriunde din JavaScript). De fapt, ele ajung să fie definite ca date membre și funcții membre (proprietăți) ale obiectului `window`. Nu este greșit nici să afirmăm că scopul global in JavaScript (pentru codul care rulează intr-un browser) este reprezentat de obiectul `window`.

Pentru următorul exemplu de cod:

```
<script>
var x = 7;
function f() {
    // do something here
    console.log('Hello');
}
</script>
```

variabila `x` și funcția `f` ajung proprietăți ale obiectului `window`. Acest lucru se poate verifica ușor făcând din nou introspecția la proprietățile acestui obiect:

```
for (i in window) console.log(i + '=' + window[i]);
```

De altfel, aceste proprietăți/metode pot fi referite și cu `window.x` sau `window.f()`.

La nivelul scopului global, cuvântul rezervat `this` va fi referință chiar la obiectul `window`. Acest lucru se poate verifica ușor cu:

```
if (this === window) console.log('True');
```

DOM (Document Object Model), manipularea DOM-ului

După cum am văzut la iterarea proprietăților obiectului `document`, acesta are proprietăți precum `title`, `location`, `head`, `body`. DOM-ul (abreviere de la Document Object Model) reprezintă o structură ierarhică de obiecte construită de către browser pentru a facilita manipularea documentului (a paginii web) din JavaScript.

Exercițiu: care credeți ca este efectul rulării codului de mai jos în consola JavaScript din *Developer Tools*?

```
window.document.body.innerHTML='';
```

După cum am spus și la începutul acestui material, JavaScript permite (prin intermediul DOM-ului) modificarea dinamică a conținutului documentului, crearea de noi elemente (tag-uri) in cadrul pagini, ștergerea unor elemente (tag-uri), modificarea atributelor HTML (adăugarea, ștergerea, schimbarea valorilor) a unor anumite tag-uri, adăugarea/ștergerea de atribute (proprietăți CSS) sau modificarea valorilor atributelor CSS existente, sau permite execuția anumitor funcții la apariția anumitor evenimente. Vom da mai jos câteva exemple pentru toate scenariile înșirate mai sus.

O operație frecventă în JavaScript este obținerea referinței la un element (tag) din pagină pe baza id-ului său, acest lucru realizându-se cu funcția `document.getElementById()`. O astfel de operație este necesară pentru a manipularea din JavaScript a elementului respectiv. Exemplu:

```
<div id="somediv"></div>
<script>
    var mydiv = document.getElementById("somediv");
    mydiv.innerHTML = 'Ana are mere';
</script>
```

Este important în codul de mai sus ca tagul `script` să succeadă tag-ului `div`. Dacă tag-ul `script` ar fi plasat în secțiunea `head` a documentului HTML, e posibil ca efectul să nu fie cel așteptat și în consola JavaScript să aveți un mesaj de eroare legat de faptul ca variabila `mydiv` este `null`. Acest lucru se datorează faptului că in momentul execuției codului JavaScript browserul nu termină de construit DOM-ul (de parsat și încărcat pagina) și `div`-ul `somediv` nu este disponibil încă în DOM.

Va recomand să „vă jucați" și cu funcțiile `document.getElementsByTagName` și `document.getElementsByClassName`.

Proprietatea `innerHTML` este folosită pe un element container (nu se poate folosi pe elemente/tag-uri fără corp, doar pe tag-urile cu marcaj de început și sfârșit de tag) pentru a accesa / modifica conținutul din interiorul tag-ului. Folosind această proprietate se poate seta pe un element container ca și conținut inclusiv cod HTML. Exemplu:

```
<div id='somediv'></div>
<script>
    var mydiv = document.getElementById('somediv');
    mydiv.innerHTML = '<a href="http://www.google.com" id="somelink">Click
here</a>';
    var mylink = document.getElementById('somelink');
    mylink.style.color = '#00FF00';
    mylink.style.backgroundColor = 'red';
</script>
```

In exemplu de mai sus, in `div`-ul `somediv` s-a creat dinamic un tag ancora. Acesta este imediat disponibil in DOM – am dat și exemple de accesare a acestuia și de modificare a stilurilor CSS folosind JavaScript (am făcut la cursul de CSS observația că un atribut CSS ce conține liniuța în denumirea sa, precum `text-align` se transforma in JavaScript in proprietatea `textAlign`).

Crearea unui nou element HTML și integrarea sa in pagina/DOM

Pe lângă exemplul de mai sus în care am creat un nou element în pagina setând ca valoare pentru atributul `innerHTML` a unui container conținut HTML, un element in DOM mai poate fi creat și adăugat folosind metodele `document.createElement()` și `appendChild()`. `document.createElement()` primește ca parametru tag-ul (elementul) care se dorește a fi creat, iar `appendChild()` se apelează pe un container - spre exemplu `container.appendChild(elementnou)`.

Pentru exemplu de mai sus, linia de cod:

```
mydiv.innerHTML = '<a href="http://www.google.com" id="somelink">Click
here</a>';
```

poate fi rescrisă astfel:

```
var mylink = document.createElement('a');
mylink.setAttribute('href', 'http://www.google.com');
mylink.setAttribute('id', 'somelink');
mylink.innerHTML = 'Click here';
```

```
mydiv.appendChild(mylink);
```

Observați în liniile de cod anterioare folosire metodei `setAttribute` pe un element HTML pentru setarea atributelor (și valorilor asociate acestor atribute) elementului. Cele două apeluri `setAttribute` de mai sus pot fi rescrise și:

```
mylink.href = 'http://www.google.com';
mylink.id = 'somelink';
```

Problemă rezolvată pentru fixarea cunoștintelor: Să se creeze dinamic folosind JavaScript un `select` cu 100 de `option`-uri (listă), elementul curent selectat fiind al 50-lea (cel cu valoarea 50). Mai jos dăm trei variante (oarecum distincte) de rezolvare:

Exemplul 1 (folosește proprietatea `innerHTML`):

```
<div id="container">
</div>
<script type="text/javascript">
    var container = document.getElementById("container");
    var string = '<select name="numar">';
    for (var i = 1; i <= 100; i++)
        if (i == 50)
            string += '<option selected value="' + i + '">' + i +
'</option>';
        else
            string += '<option value="' + i + '">' + i + '</option>';
    string += '</select>';
    container.innerHTML = string;
</script>
```

Exemplul 2 (creează dinamic `select`-ul și fiecare `option`, folosește metodele `container.appendChild()` și `element.setAttribute()`):

```
<div id='container'></div>
<script>
var container = document.getElementById('container');
select = document.createElement('select');
select.setAttribute('name', 'numar');
container.appendChild(select);
for (var i = 1; i <= 100; i++) {
    var option = document.createElement('option');
    option.setAttribute('value', i);
    option.text = i;
    select.appendChild(option);
    if (i == 50)
      option.setAttribute('selected', 'selected');
}
</script>
```

Exemplul 3 (creează dinamic selectul și fiecare `option`, nu folosește metoda `element.setAttribute()` ci accesează direct atributele elementului HTML ca proprietăți JavaScript ale acestuia, și adaugă `option`-urile la `select` folosind metoda `add` specifică doar unui `select`.

Diferența dintre `appendChild` și `add` este că prima poate fi apelată pentru orice container, `add` este specifică unui `select` pentru a permite adăugarea de elemente `option`.

```
<div id='container'></div>
<script>
var container = document.getElementById('container');
select = document.createElement('select');
select.name='numar';
container.appendChild(select);
for (var i = 1; i <= 100; i++) {
    var option = document.createElement('option');
    option.value = i;
    option.text = i;
    if (i == 50)
        option.selected = 'selected';
    select.add(option);
    oldOption = option;
}
</script>
```

Pe toate elementele (tag-urile) HTML din cadrul unui document (DOM), API-ul JavaScript prezintă o serie de funcții și proprietăți și comune tuturor acestor elemente. Câteva exemple intuitive in acest sens sunt metode precum `setAttribute()`, `click()`, `remove()` sau proprietăți precum `tagName` și `style`. API-ul JavaScript oferă însă pentru o serie de elemente punctuale unele proprietăți și metode specifice ce pot fi apelate doar pe elementele respective. Un exemplu in acest sens este metoda `add()` din exemplul anterior prezentă pe un container de tip `select` – `appendChild()` fiind comună tuturor containerelor. In acest context este important să prezentam un pic modul de manipulare/accesare a unui tabel (element `table`) din JavaScript. Dacă `myTable` reprezintă referința către un element tabel obținută spre exemplu cu `myTable = document.getElementById('someTable')`, atunci putem efectua pe acest tabel următoarele operații

- adăugarea (sau ștergerea) de linii (rânduri) noi in tabel: `myTable.insertRow()`, `myTable.deleteRow(index);`
- obținerea tuturor rândurilor din tabel: `myTable.rows` – tablou ce conține rândurile tabelului și ce poate fi iterat. `myTable.rows.length` reprezintă lungimea acestui tablou;
- daca `row` reprezintă un rând din tabel, obținut spre exemplu cu `row = myTable.rows[i]`, pe acest rând se pot adăuga (sau șterge) celule cu `row.insertCell()`, respectiv `row.deleteCell(index);`
- `row.rowIndex` reprezintă indexul rândului în cadrul tabelului;
- celulele de pe un rând se pot accesa prin intermediul proprietății `row.cells`, ce returnează un tablou ce poate fi iterat, `row.cells.length` reprezentând lungimea acestui tablou.

Exemplu. Următoarea secvență de cod populează celulele un tabel HTML cu numere de ordine de la 1 la `n x m` unde `n` reprezintă numărul de linii și `m` numărul de coloane:

```
<script>
    var myTable = document.getElementById("someTable");
    var rows = myTable.rows;
```

```
    var k = 1;
    for (i = 0; i < rows.length; i++) {
       var cells = rows[i].cells;
       for (j = 0; j < cells.length; j++)
          cells[j].innerHTML = k++;
    }
</script>
```

Evenimente

Unul dintre scopurile inițiale ale limbajului JavaScript a fost să faciliteze interacțiunea paginii Web cu utilizatorul și să permită posibilitatea de a executa anumite secvențe de cod la apariția anumitor evenimente. Evenimentele JavaScript se pot identifica ca proprietăți (metode membre) pe un element, metode prefixate de obicei cu prefixul `"on"` și la a căror apariție se pot executa secvențe de cod sau asocia funcții care să se execute. Puteți relua exemplele anterioare de iterare a proprietăților / metodelor prezente pe obiectele `window` și `document` pentru a revedea metodele ce pot fi invocate pe aceste obiecte.

Dăm în continuare câteva exemple de folosire a evenimentelor:

Exemplul 1:

```
<button onclick='alert("Vei fi redirectat!"); window.location =
"https://www.youtube.com/watch?v=Rtog93HyR3k";'>
Apasă-l...
</button>
```

La tratarea unui eveniment, cuvântul rezervat `this` indică spre obiectul pe care a apărut evenimentul. Exemplul 2:

```
<button onclick='alert("Ati dat click pe un " + this.tagName + " pe care
scrie " + this.innerHTML);'>
Apasă-l...
</button>
```

Dacă secvența de cod care trebuie executată la apariție evenimentului este mai complexă, aceasta poate fi plasată și intr-o funcție separată. Exemplul 3:

```
<script>
function faCeva(element) {
    alert("Ati dat click pe un " + element.tagName + " pe care scrie " +
element.innerHTML);
}
</script>
<button onclick='faCeva(this)'>Apasă-l...</button>
```

În exemplu de mai sus, pentru a avea acces la elementul pe care a apărut evenimentul, referința la acest element, `this`, a fost trimisă ca parametrul actual funcției `faCeva()`. La apelul acesteia, parametrul formal `element` va indica spre elementul HTML pe care a apărut evenimentul. Nu este însă obligatorie

trimiterea acestui parametru pentru a avea acces in cadrul funcției care tratează un eveniment la obiectul care a generat apariția evenimentului. La nivelul unui *handler* de eveniment (funcție de tratare a evenimentului), obiectul global `window` prezintă prin intermediul unei proprietăți (dată membră) numită `event` informații despre evenimentul care tocmai se tratează. La rândul său, obiectul `event` prezintă o proprietate numită `target` care indică spre elementul pa care s-a apelat evenimentul. Astfel, exemplu 3 de mai sus, poate fi rescris in exemplul 4:

```
<script>
function faCeva() {
    var element = window.event.target; // sau simplu event.target
    alert("Ati dat click pe un " + element.tagName + " pe care scrie " +
element.innerHTML);
}
</script>
<button onclick='faCeva()'>Apasă-l...</button>
```

Unui element din cadrul paginii (încărcat în DOM), i poate asocia un eveniment și dinamic (la runtime). Spre exemplu, odată obținută referința `myElem` la un element (fie obținută cu `document.getElementById()` sau că este vorba de un element proaspăt creat cu `document.createElement()`), putem preciza ce se întâmplă la click pe acest element in modul următor (exemplul 5):

```
function faCeva() {
    alert('S-a dat click!');
}
myElem.onclick = faCeva;
```

sau și mai simplu:

```
myElem.onclick = function () {
    alert('S-a dat click!');
}
```

In exemplu de mai jos, asociem in acest fel un *handler* de eveniment pentru tratarea click-urilor pe toate celulele unui tabel. La click pe o celulă de tabel, dorm să afișam linia și coloana pe care se regăsește acea celulă. Exemplul 6:

```
<table id="someTable" border="1">
<tr><td>a</td><td>b</td><td>c</td></tr>
<tr><td>d</td><td>e</td><td>f</td></tr>
<tr><td>g</td><td>h</td><td>i</td></tr>
</table>
<script>
    var myTable = document.getElementById("someTable");
    var rows = myTable.rows;
    for (i = 0; i < rows.length; i++) {
       var cells = rows[i].cells;
       for (j = 0; j < cells.length; j++) {
          let l = i;
          let c = j;
          cells[j].onclick = function() {
              alert('S-a dat click pe linia ' + l + ' si coloana ' + c);
```

```
            }
        }
    }
</script>
```

Vă recomand cu căldura sa rulați / testați codul de mai sus. Observați variabilele declarate cu cuvântul rezervat `let`. Am amintit anterior în acest material că variabilele pot fi declarate și folosind `let` și am promis că revenim asupra folosirii acestuia. Pe scurt, `let` permite declararea unor variabile al căror scop este la nivel de bloc. O implementare naivă a codului de mai sus nu ar fi folosit variabilele auxiliare `l` și `c` declarate cu `let`, iar funcția de tratare a evenimentului click ar fi arătat astfel:

```
            // incorect !!!
            cells[j].onclick = function() {
                alert('S-a dat click pe linia ' + i + ' si coloana ' + j);
            }
```

Ce nu este corect în acest caz? Testați exemplul 6, folosind liniile de cod de mai sus pentru tratarea evenimentului click.

În unele situații este util / se dorește ca pe un element să fie adăugate pentru același eveniment mai multe funcții de tratare a evenimentul. In acest sens se poate folosi metoda `element.addEventListener`. Exemplu:

```
<button id="myElem">Apasă-l...</button>
<script>
function faCeva() {
    alert('Salut');
}

function faAltceva() {
    console.log('Salut din nou');
}

var myElem = document.getElementById("myElem");
myElem.addEventListener('click', faCeva);
myElem.addEventListener('click', faAltceva);
</script>
```

Funcția de tratare a unui eveniment dată ca parametru la metoda `addEventListener`, poate să fie și o funcție anonimă. Exemplu:

```
myElem.addEventListener('click', function() {
    alert('Salut');
});
```

Observații:

- Numele evenimentului (`'click'` in cazul de fata) nu mai trebuie prefixat cu 'on';
- Uneori se dorește ca o funcție de tratarea a unui eveniment să nu se mai apeleze la declanșarea acestuia. In acest caz se poate folosi metoda element.removeEventListener. Important in acest

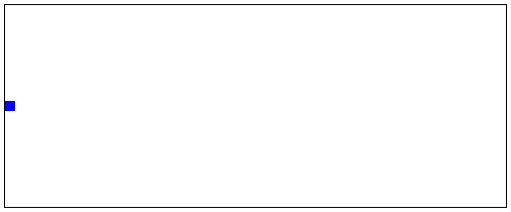
scenariu, este ca funcția de tratarea a evenimentului adăugată / care se dorește a fi înlăturată să nu fie una anonimă. Exemplu:

```
<button id="myElem">Apasă-l...</button>
<script>
function faCeva() {
    alert('Aceasta functie se apeleaza doar la primul click');
    event.target.removeEventListener('click', faCeva);
}

var myElem = document.getElementById("myElem");
myElem.addEventListener('click', faCeva);
</script>
```

setTimeout, clearTimeout, setInterval, clearInterval

La începutul acestui material am făcut o observație deosebit de importantă despre modul de execuție a codului JavaScript (scrisă cu roșu) și anume că în timp ce browser-ul execută cod JavaScript, acesta nu „face nimic altceva" – nu (re)randează pagina, nu interacționează cu utilizatorul (nu răspunde la comenzi/click-uri de mouse etc). Pentru a demonstra și a face mai ușor de înțeles acest lucru, dăm mai jos următoarea problemă rezolvată:



Fie un pătrățel colorat cu albastru plasat in stânga unui container mai mare conform figurii alăturate. Folosind cod JavaScript să se de deplaseze liniar acest pătrățel într-un anumit interval de timp de la stânga la dreapta containerului părinte. Prezentăm pentru această problemă, două variante de rezolvare, una „naivă" și incorectă și una corectă. În ambele rezolvări, pătrățelul albastru este reprezentat de un div cu poziționare absolută, plasat intr-un div mai mare (containerul) ce are poziționare relativă (tocmai pentru a putea plasa absolut un element in cadrul său) – pentru cine nu înțelege aceste poziționări, rog să recapituleze cursul de CSS. Pătrățelul poate fi mutat, modificând proprietatea CSS left. Varianta 1 de rezolvare este prezentată mai jos, codul acesteia fiind disponibil aici.

```
<body>
Click anywhere for start...
<div style="position: relative; width: 500px; height: 200px; border: 1px
solid black">
<div id="patratel" style="position: absolute; left: 0px; top: 95px; width:
10px; height: 10px; background-color: blue">
</div>
</div>
Coodonate patratel: <span id="label"></span>
</body>
<script>
document.body.onclick = function () {
    var patratel = document.getElementById('patratel');
    for (i = 0; i <= 500; i+=0.001) {
```

```
        patratel.style.left = i + 'px';
        document.getElementById('label').innerHTML = patratel.style.left;
    }
}
</script>
```

In rezolvarea „naivă" de mai sus, modificăm proprietatea `patratel.style.left` într-o iterație `for`. Dacă rulați acest exemplu, veți observa că pătrățelul nu se deplasează liniar, el fiind afișat de către browser doar în pozițiile inițială și finală. El nu este randat în nicio poziție intermediară pentru că, codul JavaScipt nu se termină (se execută dintr-o bucată). Mai mult, încercați după ce ați dat click și ați pornit mutarea pătrățelului să dați un F12 pentru a deschide *Developer Tools*. Funcționează?

Varianta corectă de rezolvare presupune deplasarea pătrățelului cu un anumit număr de pixeli mai la dreapta, după care, peste un anumit număr de milisecunde repetarea acestei operații. Din păcate (sau din fericire ☺) JavaScript nu oferă o funcție `sleep` clasică care să permită „așteptarea" unui număr de milisecunde (și dacă ar exista o astfel de funcție, cat timp s-ar executa funcția `sleep`, tot cod JavaScript s-ar executa, problema nerandării pătrățelului s-ar păstra). In schimb, JavaScript permite prin folosirea funcției `setTimeout` (funcție membră pe obiectul `window`) apelarea unui anumite funcții peste un număr precizat de milisecunde:

```
<script>
function salut() {
    alert('Salut cu o intarziere de 3 secunde');
}
setTimeout(salut, 3000);
</script>
```

Observații:

- Aveți grija să dați ca parametru numele funcției - `salut` în cazul de fată, nu să apelați funcția `salut()` – cu paranteze după. O expresie de forma `setTimeout(salut(), 3000)` este greșită pentru că duce la execuția instantă a funcției `salut`, nu la apelarea ei peste 3000 de milisecunde;
- Dacă funcția care se dorește a fi apelată are parametrii, aceștia se specifică la funcția `setTimeout` după numărul de milisecunde, spre exemplu: `setTimeout(salut, 3000, mesaj)`. Total greșit: `setTimeout(salut(mesaj), 3000)` pentru că ar duce din nou la apelul instant al funcției `salut`;
- Funcția dată ca parametru funcției `setTimeout` și care trebuie executată peste un anumit număr de milisecunde, poate fi și o funcție anonimă. Astfel, exemplu de mai sus poate fi scris mai scurt astfel:

```
<script>
setTimeout(function () {
    alert('Salut cu o intarziere de 3 secunde');
}, 3000);
</script>
```

Dacă se dorește reapelarea periodică a funcției din 3000 in 3000 de milisecunde, funcția apelată se poate termina cu un nou `setTimeout`:

```
<script>
function salut() {
    alert('Salut din 3 in 3 secunde');
    setTimeout(salut, 3000);
}
setTimeout(salut, 3000);
</script>
```

Apelarea automată peste o anumită perioada de timp stabilită cu funcția `setTimeout` poate fi anulată prin intermediul funcției `clearTimeout`. Funcția `clearTimeout` primește ca parametru o variabilă întoarsă de `setTimout`-ul pe care doriți să-l anulați. Exemplu:

```
var t = setTimeout(f, 3000);
if (whatever_happens)
    clearTimout(t);
```

Puteți privi variabila `t` ca pe un fel de identificator de thread – deși exprimarea nu e tocmai corectă.

Pe lângă `setTimeout/clearTimeout`, JavaScript mai oferă tandemul de funcții `setInterval/clearInterval`. Funcționalitatea și utilizarea acestora este asemănătoare, cu observația că `setInterval` apelează periodic funcția specificată (în timp ce `setTimeout` apela funcția o singura dată și era nevoie de un nou `setTimeout` la sfârșitul funcției apelate pentru un nou apel al acesteia). Exemplul anterior care ne salută periodic din 3 in 3 secunde, poate fi rescris mai simplu folosind `setInterval` astfel:

```
<script>
setInterval(function() {
    alert('Salut din 3 in 3 secunde');
}, 3000);
</script>
```

Revenind la problema cu pătrățelul, varianta 2 (corectă) a rezolvării este disponibilă mai jos, iar codul aici.

```
<body>
Click anywhere for start...
<div style="position: relative; width: 500px; height: 200px; border: 1px
solid black">
<div id="patratel" style="position: absolute; left: 0px; top: 95px; width:
10px; height: 10px; background-color: blue">
</div>
</div>
Coodonate patratel: <span id="label"></span>
</body>
<script>
var i = 0;
document.body.onclick = function () {
    var patratel = document.getElementById("patratel");
    var t = setInterval(function() {
```

```
        if (i < 500) {
            i+=0.5;
            patratel.style.left = i + 'px';
            document.getElementById('label').innerHTML = patratel.style.left;
        } else
            clearInterval(t);
    }, 1);
}
</script>
```

Ce se întâmplă dacă pe varianta corectă dați de mai multe ori click? Puteți explica „fenomenul”?

Încărcarea dinamică a unui fișier JavaScript

Uneori se dorește ca execuția unui secvențe de cod JavaScript sau încărcarea unui fișier ce conține cod JavaScript să se facă ulterior încărcării paginii (în principal pentru a reduce timpul de încărcare și de randare al acesteia). Un tag `script` se poate crea dinamic din cod JavaScript la fel ca orice alt element HTML (precum `select`-urile și `option`-urile din exemplele anterioare prezentare în acest material. Dăm mai jos un exemplu in acest sens.

In fișierul `extern.js`:

```
alert('Hello World!');
```

Intr-un fișier HTML (exemplul în acțiune poate fi vizualizat aici):

```
<body>
<script>
  var newScript = document.createElement('script');
  newScript.src = 'extern.js';
  document.body.appendChild(newScript);
</script>
</body>
```

Greșeli frecvente des întâlnite…

1. Se dorește executarea unei funcții `faCeva` la încărcarea paginii (documentului HTML) și în acest sens se apelează funcția `faCeva` pe evenimentul `onload` al elementului `body`. Exemplu:

```
<script>
function faCeva() {
   document.getElementById("demo").innerHTML = "Hello World!";
}
</script>
<body onload="faCeva()">
<div id="demo">
</div>
</body>
```

Deși e posibil ca acest exemplu să funcționeze, există pericolul ca la momentul execuției funcției `faCeva` (adică la apariția evenimentului `onload` pe `body` = încărcarea `body`-ului in DOM), `div`-ul cu id-ul `demo` să nu fie încărcat încă în DOM. Acest lucru face ca `document.getElementById("demo")` să returneze `null`. Soluția corectă este apelarea funcției spre execuție într-un tag `<script>` plasat la sfârșitul documentului:

```
<script>
function faCeva() {
   document.getElementById("demo").innerHTML = "Hello World!";
}
</script>
<body>
<div id="demo">
</div>
</body>
<script>
    faCeva();
</script>
```

Vom vedea cursul următor că jQuery oferă o modalitate mult mai elegantă de a executa ceva la încărcarea documentului.

2. Încărcarea / construirea unei noi imagini și accesarea dimensiunilor acesteia

Uneori este necesară construirea dinamică a unui element imagine și încărcarea dinamică a acestuia în DOM. Problemele apar când se dorește accesarea dimensiunilor imaginii proaspăt create. În exemplul de mai jos, se construiește dinamic un element imagine, se stabilește URL-ul (fișierul sursă) pentru acest element și se dorește afișare imaginii la 50% din rezoluția imaginii originale (înjumătățim lățimea imaginii, înălțimea se va autoscala automat):

```
<body></body>
<script>
var img = new Image();
// sau var img = document.createElement('img');
img.src = "poza.jpg";
img.width = img.width / 2;
// dorim sa afisam poza la jumatatea rezolutiei acesteia
document.body.appendChild(img);
</script>
```

Exemplul de mai sus (disponibil on-line aici) este posibil să nu ofere rezultatul dorit, pentru că la momentul folosirii expresiei `img.width` ca și right value, imaginea nu se termină de încărcat iar lățimea acesteia este necunoscută (`img.width` va avea valoarea 0), `img.width` folosit că și left value primind valoare 0 (fapt ce va duce la neafișarea imaginii nici când aceasta este încărcată complet în browser). Dacă exemplu vă funcționează puteți să-l încercați cu o imagine de rezoluție mai mare sau să dați un refresh (CTRL-F5). Varianta corectă disponibilă aici presupune scalarea imaginii doar când aceasta se încarcă in DOM (browser-ul a terminat încărcarea acesteia) și se cunoaște lățimea acesteia:

```
<body></body>
<script>
```

```
var img = new Image();
// sau var img = document.createElement('img');
img.src = "poza.jpg";
img.onload = function() {
    img.width = img.width / 2;
}
document.body.appendChild(img);
</script>
```

Încă un pic despre obiecte în JavaScript

Redăm mai jos câteva exemple pentru a face mai ușor de înțeles atât unele concepte OOP din JavaScript cât și unele aspecte „de finețe” ale acestui limbaj.

Exemplul 1 (cod disponibil aici)

Fie secvența JavaScript de mai jos care dorește să calculeze media notelor unui student:

```
<script>
    function Student(name, grades) {
        this.name = name;
        this.grades = grades;
    }

    function displayStudentAverage() {
        var sum = 0;
        for (var i = 0; i < this.grades.length; i++)
            sum += this.grades[i];
        var average = sum / this.grades.length;
        alert(this.name + " - media " + average);
    }

    Student("Pop Ionel", new Array(6, 8, 10));
    displayStudentAverage();
</script>
```

Am amintit anterior în acest material că funcțiile și variabilele declarate în scopul global ajung proprietăți membre ale obiectului `window`. Ambele funcții de mai sus fiind definite în scopul global, ajung funcții membre ale acestui obiect. După apelul funcției `Student`, obiectul `window` va deține două proprietăți noi, `name` și `grades` care memorează numele și notele studentului (pentru a vă convinge de acest lucru puteți itera proprietățile obiectului `window`, veți observa pe acest obiect patru proprietăți noi: `Student`, `displayStudentAverage`, `name` și `grades`). Funcția `displayStudentAverage()` în exemplu de cod se mai sus se apelează tot pe obiectul `window`, calculând și afișând media pentru array-ul de note memorate pe obiectul `window`.

Observație: în exemplul de mai sus, `this` în ambele funcții va fi referință spre obiectul `window` pe care se apelează cele doua funcții. `this.name` și `this.grades` vor indica spre proprietăți ale obiectului `window`. Variabilele `sum` și `average` sunt variabile locale în cadrul funcției `displayStudentAverage`, ele neaparținând obiectului `window`.

Exemplul 2 (cod disponibil aici)

Ne dorim evident să calculăm și afișăm media pentru mai mulți studenți. Funcția `Student` poate fi folosită și ca și constructor pentru a construi un obiect de tipul `Student`, precum în exemplul de mai jos. `s1` și `s2` vor fi două obiecte noi, ambele proprietăți ale obiectului `window` (sunt declarate cu `var` în scopul global), fiecare student având propriul nume și tablou cu note. In cadrul funcției `Student`, `this` de această dată va indica spre obiectul nou construit. Pe exemplul anterior (exemplul 1), funcția `displayStudentAverage` nu poate fi apelată pe un student, `displayStudentAverage` fiind funcție membră (proprietate) a obiectului `window`, nu a unui obiect `Student`. Pentru a putea afișa media unui student, în cadrul funcției constructor `Student` putem adăuga o proprietate membră funcție numită `show` care să indice spre funcția de afișare a mediei `displayStudentAverage`.

```
<script>
    function Student(name, grades) {
        this.name = name;
        this.grades = grades;
        this.show = displayStudentAverage;
    }

    function displayStudentAverage() {
        var sum = 0;
        for (var i = 0; i < this.grades.length; i++)
            sum += this.grades[i];
        var average = sum / this.grades.length;
        alert(this.name + " - media " + average);
    }

    var s1 = new Student("Pop Ionel", new Array(6, 8, 10));
    var s2 = new Student("Ionescu Maria", new Array(8, 9, 10));
    s1.show();
    s2.show();
</script>
```

Exemplul 3 (cod disponibil aici)

În exemplul anterior, funcția `displayStudentAverage` rămâne cumva "de izbeliște", membră pe obiectul `window`. Pentru a nu fi disponibilă în scopul global (nu are nici un sens acolo), putem să o definim ca funcție imbricată în cadrul funcției `Student`:

```
<script>
    function Student(name, grades) {
        this.name = name;
        this.grades = grades;
        this.show = displayStudentAverage;

        function displayStudentAverage() {
            var sum = 0;
            for (var i = 0; i < this.grades.length; i++)
                sum += this.grades[i];
```

```
            var average = sum / this.grades.length;
            alert(this.name + " - media " + average);
        }
    }

    var s1 = new Student("Pop Ionel", new Array(6, 8, 10));
    s1.show(); // show se comporta ca o metoda publica
    alert(s1.name); // name se comporta ca o data membra publica
</script>
```

În acest exemplu, oriunde în cadrul funcției `Student` (puteți sa va gândiți la funcția `Student` ca la o clasa, începe să semene cu o clasa din C++/Java...), funcția `displayStudentAverage` se comportă ca o funcție privată, în timp ce `show` se comportă ca o funcție publică (putând fi apelată din exterior pe un obiect de tipul `Student`). De altfel, toate proprietățile membre (indiferent că sunt date membre sau funcții membre) stocate cu "`this.`" în cadrul constructorului sunt publice (pot fi accesate din exterior pe un obiect de tipul `Student`) - a se vedea ultimul alert din exemplul de mai sus.

Exemplul 4 (cod disponibil aici)

Funcția care afișează media unui student poate fi declarată și ca funcție anonimă:

```
<script>
    function Student(name, grades) {
        this.name = name;
        this.grades = grades;
        this.show = function() {
            var sum = 0;
            for (var i = 0; i < this.grades.length; i++)
                sum += this.grades[i];
            var average = sum / this.grades.length;
            alert(this.name + " - media " + average);
        }
    }

    var s1 = new Student("Pop Ionel", new Array(6, 8, 10));
    s1.show(); // show se comporta ca o metoda publica
</script>
```

Exemplul 5 (cod disponibil aici)

Studentul s1 din exemplul anterior ar mai putea fi definit și in modul următor, însă o astfel de definire nu este utilă în momentul în care dorim să declaram mai multe obiecte de același tip:

```
<script>
var s1 = {
    name: "Pop Ionel",
    grades: [6, 8, 10],
    show: function() {
        var sum = 0;
        for (var i = 0; i < this.grades.length; i++)
```

```
            sum += this.grades[i];
        var average = sum / this.grades.length;
        alert(this.name + " - media " + average);
    }
};
s1.show();
</script>
```

Exemplul 6 (cod disponibil aici)

Uneori se dorește ascunderea detaliilor de implementare și a datelor membre / proprietăților unui obiect. În acest sens, putem să declarăm eventualele sale proprietăți ca variabile locale, ele comportându-se ca niște date membre private. În exemplu de mai jos `_average`, `_name` și `_grades` au un astfel de comportament fiind variabile locale, la fel cum `computeAverage` este o funcție privată. Atât `computeAverage` cât și `_average`, `_name` și `_grades` pot fi apelate/accesate doar din interiorul funcției `Student` (interiorul clasei), nefiind vizibile din exterior. Singurul lucra văzut în exterior este metoda `show()`.

```
<script>
    function Student(name, grades) {
        var _average;
        var _name = name;
        var _grades = grades;

        this.show = function() {
            computeAverage();
            alert(_name + " - media " + _average);
        }

        function computeAverage() {
            var sum = 0;
            for (var i = 0; i < _grades.length; i++)
                sum += _grades[i];
            _average = sum / _grades.length;
        }
    }

    var s1 = new Student("Pop Ionel", new Array(6, 8, 10));
    var s2 = new Student("Ionescu Maria", new Array(8, 9, 10));
    s1.show(); // show se comporta ca o metoda publica
    s2.show();
</script>
```

Exemplul 7 (cod disponibil aici)

Pentru a simplifica codul, în exemplul anterior putem renunța la variabilele locale `_name` și `_grades` și să folosim în interiorul "clasei" direct parametrii formali `name` și `grades` ai funcției `Student`:

```
<script>
```

```
    function Student(name, grades) {
        var _average;

        this.show = function() {
            computeAverage();
            alert(name + " - media " + _average);
        }

        function computeAverage() {
            var sum = 0;
            for (var i = 0; i < grades.length; i++)
                sum += grades[i];
            _average = sum / grades.length;
        }
    }

    var s1 = new Student("Pop Ionel", new Array(6, 8, 10));
    var s2 = new Student("Ionescu Maria", new Array(8, 9, 10));
    s1.show(); // show se comporta ca o metoda publica
    s2.show();
</script>
```

Exemplul 8 (cod disponibil aici)

Funcționalitatea unui obiect `Student` poate fi extinsă cu noi metode. Astfel, la fel cum l-am înzestrat pe Chuck Norris cu metoda `kick` într-un exemplu anterior, putem să-l înzestrăm pe studentul `s1` cu o metoda `hasToRepeatAnExam` care va returna `true` daca acest student are cel puțin o materie restantă și `false` în caz contrar:

```
<script>
    function Student(name, grades) {
        this.name = name;
        this.grades = grades;
        this.show = function() {
            var sum = 0;
            for (var i = 0; i < this.grades.length; i++)
                sum += this.grades[i];
            var average = sum / this.grades.length;
            alert(this.name + " - media " + average);
        }
    }

    var s1 = new Student("Pop Ionel", new Array(3, 8, 10));
    var s2 = new Student("Ionescu Maria", new Array(8, 9, 10));

    s1.show();
    s2.show();

    s1.hasToRepeatAnExam = function() {
        for (var i = 0; i < this.grades.length; i++)
           if (this.grades[i] < 5)
              return true;
        return false;
```

```
    }

    alert(s1.name + " are restante: " + s1.hasToRepeatAnExam());
    alert(s2.name + " are restante: " + s2.hasToRepeatAnExam()); // eroare in
consola JavaScript

</script>
```

Observații importante:

- Doar studentul `s1` are o proprietate/metodă `hasToRepeatAnExam` ce poate fi invocată pe acest obiect. `s2` nu are această proprietate/metodă.
- Daca `grades` nu ar fi fost proprietate publică a obiectului (memorată în funcția constructor `Student` cu `this.grades`) ci doar variabilă locală (declarată doar local cu `var` in cadrul funcției constructor `Student` sau parametru formal al acestui constructor) nu ar mai fi fost accesibilă.

JavaScript prototype

Exemplul 9 (cod disponibil aici)

Evident este de dorit să extindem uneori funcționalitatea tuturor obiectelor de același tip. Adică, să înzestrăm toate obiectele de tip `Student` (gata instanțiate sau instanțiate pe viitor) cu o metodă `hasToRepeatAnExam` care să poate fi apelata pe aceste obiecte. Acest lucru se poate face în JavaScript prin intermediul unei proprietăți speciale a funcției constructor `Student` denumita `prototype`.

```
<script>
    function Student(name, grades) {
        this.name = name;
        this.grades = grades;
        this.show = function() {
            var sum = 0;
            for (var i = 0; i < this.grades.length; i++)
                sum += this.grades[i];
            var average = sum / this.grades.length;
            alert(this.name + " - media " + average);
        }
    }

    var s1 = new Student("Pop Ionel", new Array(3, 8, 10));
    var s2 = new Student("Ionescu Maria", new Array(8, 9, 10));

    s1.show();
    s2.show();

    Student.prototype.hasToRepeatAnExam = function() {
        for (var i = 0; i < this.grades.length; i++)
           if (this.grades[i] < 5)
              return true;
        return false;
```

```
    }

    alert(s1.name + " are restante: " + s1.hasToRepeatAnExam());
    alert(s2.name + " are restante: " + s2.hasToRepeatAnExam());
</script>
```

Extindere funcționalității unor tipuri predefinite folosind prototype

Folosind proprietatea `prototype` poate fi extinsă inclusiv funcționalitatea unor tipuri predefinite din JavaScript. Spre exemplu, am enumerat anterior diverse metode care pot fi apelate pe un `Array`, dar din păcate pe un astfel de obiect lipsește o metoda `shuffle` care ar fi utilă când se dorește "amestecarea" (randomizarea ordinii) elementelor din tablou. Folosind `prototype` este ușor de implementat o astfel de metodă ce poate fi ulterior apelată pe orice `Array`. Codul sursa de mai jos este disponibil on-line si aici:

```
<script>
Array.prototype.shuffle = function() {
   this.sort(function (x, y) {
      return (Math.random() * 2 - 1);
   });
}

var x = new Array(1, 2, 3, 4, 5);
x.shuffle();
console.log(x);
</script>
```

Metoda `shuffle` apelează metoda `sort` pe tablou ce trebuie sortat (`this`), `sort` având ca parametru o funcție anonima custom de comparație a două elemente ale tabloului (această funcție returnează aleator dacă doua elemente `x` si `y` comparate trebuie interschimbate sau nu).

Cu *Developer Tools* pornit (F12), puteți da refresh pe pagina (F5). Observați valorile din tabloul `x` afișate la consolă într-o ordine aleatoare în urma execuției metodei `shuffle` pe acest tablou.

Este posibil să fi promis pe parcursul materialului că vom detalia mai târziu anumite lucruri și să îmi fi scăpat să revin la ele. Sunt deschis la orice sugestii de îmbunătățire a acestui material și observații privind eventuale scăpări (acord bonusuri recompensă ☺). Mulțumesc.