Laborator 08 - JavaScript

Obiective: În urma parcurgerii acestui laborator studentul va:

- fi familiarizat cu conceptul scripturilor client-side
- înţelege conceptele generale ale limbajului JavaScript
- putea integra cod JavaScript într-o aplicaţie web

Introducere în client-side scripting

Scripturile interpretate şi executate de browser sunt denumite generic scripturi client-side. În prezent, ele reprezinta metoda principală prin care aplicaţiile web reuşesc să ofere interactivitate utilizatorilor. Caracteristicile cele mai importante ale lor sunt:

- executate de browser, pe maşina utilizatorului
- au acces la informaţiile şi setările din browser, însă nu şi în afara lui (rulează în cadrul procesului browserului şi nu au acces la sistemul de fişiere).
- implementările interpretoarelor de obicei sunt inconsistente între browsere (în special IE).

Cel mai folosit limbaj pentru scripturi client-side este de departe **Javascript**. Laboratorul prezent şi cel viitor se vor concentra pe acest limbaj.

Limbajul Javascript

Limbajul este bazat pe standardul ECMAScript (fel ca şi ActionScript, limbajul folosit de Adobe în Flash şi Flex). În ciuda denumirii neinspirate, Javascript **nu are nici o legătură cu Java**. Javascript este un limbaj cu **verificare dinamică a tipurilor**. De asemenea, el este **slab tipat** şi, deşi este bazat pe obiecte, foloseşte o paradigmă diferită faţă de majoritatea celorlalte limbaje OO: **moştenire prototipală**.

Sintaxa

Sintaxa este similară cu cea a familiei de limbaje C/Java. Exemplul de mai jos ilustrează acest lucru:

```
function helloBrowser() {
    var x = 1;

    if (x) {
        x = "Hello, Browser!";
    }
    console.log(x);
}

helloBrowser();
```

Variabile în Javascript. Variable scope.

Implicit, variabilele în Javascript sunt globale. Acesta este unul din cele mai mari capcane ale limbajului şi trebuie evitată folosirea variabilelor globale în orice situaţie, pentru că generează foarte multe erori. Folosirea operatorului **var** la declararea unei variabile previne aceste erori pentru că leagă variabila de contextul în care a fost creată. De asemenea, în Javascript nu există block-scope (ca în majoritatea limbajelor cu sintaxă C-like), ci doar functional-scope, ceea ce înseamnă că o variabilă declarată în interiorul unei funcţii este vizibilă oriunde în acea funcţie, dar nu şi în exterior.

```
var foo = function () {
    var a=3, b=5;
    var bar = function ( ) {

        var b = 7, c = 11;
        // a=3, b=7, c=11

        a += b + c;
        // a=21, b=7, c=11
    };

    //a = 3, b = 5, c = undefined
    bar();
    //a = 21, b=5
};
```

Obiecte în JavaScript

- Repartizare teme
- Catalog PW

Laboratoare

- Laborator 01 - Introducere
- Laborator 02 - BD in PHP
- Laborator 03 - Arrays, Magic Methods
- Laborator 04 - (X)HTML, CSS
- Laborator 05 - Formulare HTML, Persistența Datelor
- Laborator 06 - Securitate I
- Laborator 07 - Securitate II
- Laborator 08 - JavaScript
- Laborator 09 - DOM scripting
- Laborator 10 - AJAX
- Laborator 11 - Web Frameworks

Teme

- Tema 01 - API pentru BD
- Tema 02 - Template System & Controller
- Tema 03 - Generator de formulare
- Tema 04 - Tree Web UI

Login

Javascript este un limbaj obiectual. În afara câtorva tipuri de bază (number, string, boolean, null, undefined), **totul** în Javascript este un obiect. Şirurile sunt obiecte, expresiile regulate sunt obiecte, funcţiile sunt obiecte.

Marea diferenţă faţă de celelalte limbaje de programare obiectuale este faptul că în Javascript nu există clase. Acest lucru lasă obiectele libere de orice constrângeri pentru numele şi valorile membrilor şi metodelor. Obiectele pot fi astfel manipulate foarte uşor, chiar şi pentru a reprezenta structuri de date complicate (de ex. arbori). Observaţi că în Javascript nu există modificatori de acces pentru membri unui obiect (implicit, totul este public).

Object literals şi JSON

O particularitate interesantă a limbajului este notaţia prezentată în secvenţa de cod de mai jos:

```
var judet = {
 nume: "Brasov",
 resedinta: {
  nume: "Brasov",
  populatie: "200000"
 }
};
console.log(judet);
```

Obiectele pot fi create foarte usor folosind notaţia {}. Observaţi că numele proprietăţilor nu este obligatoriu să fie incadrate cu ghilimele, însă valorile da. Această notaţie este foarte utilă şi este întâlnită foarte frecvent în practică. Ea a stat la baza JSON (**J**ava**S**cript **O**bject **N**otation), care astăzi este folosit între multe aplicaţii şi între multe limbaje de programare ca format pentru mesajele transmise. JSON este foarte uşor de folosit în aplicaţiile web pentru că **este** Javascript.

În prezent, bibliotecile standard Javascript nu conţin parsere pentru stringuri în format JSON, iar folosirea funcţiei **eval()** este considerată nesigură. Creatorul JSON pune la dispoziţie un parser pentru a evita situaţia în care ajunge să fie executat cod maliţios din cauza folosirii eval(). De asemenea, multe frameworkuri de Javascript conţin parsere pentru JSON. În următorul exemplu este prezentat un mod de utilizare a JSON (reţineţi că nu este recomandat să folosiţi eval în aplicaţii din producţie).

```
var jsonText = '[{"first": "Cici", "last": "Pop"}, {"first": "Pandele", "last": "Georgescu"}]

var receivedData = eval( '(' + jsonText + ')' );

console.log(receivedData);
```

Referenţierea dinamică a membrilor obiectelor

Pentru a referenția dinamic un membru al unui obiect puteți folosi o notație foarte asemănătoare cu cea de la array-uri:

```
var sky = { color: "blue" },
    property = "color";

// sky['color'] === sky.color === sky[property] === "blue"
```

Funcţii in Javascript

Funcţiile în Javascript sunt obiecte. Ca orice alt obiect, sunt colecţii de perechi nume-valoare, conţinând un link ascuns la un obiect prototip (Function.prototype, care este legat la rândul lui la Object.prototype). De asemenea, fiecare funcţie conţine alte două proprietăţi ascunse: contextul său şi codul care implementează comportarea sa. Ţinând cont de aceste proprietăţi, ele pot fi folosite ca orice altă valoare - pot fi stocate în variabile, obiecte şi array-uri, pot fi transmise ca argumente la o altă funcţie sau pot fi returnate ca rezultat al apelului unei funcţii. Particularitatea lor, care le diferenţiază de celelalte constructe ale limbajului este că funcţiile pot fi **invocate**.

Declarare şi invocare

Se disting patru tipare de invocare pentru o funcţie în Javascript.

Primul, invocarea unei funcţii ca **metodă** a unui obiect se aplică acelor funcţii care au fost declarate ca proprietăţi ale obiectelor. Cel de-al doilea tratează cazul în care funcţia nu a fost declarată ca membru al unui obiect şi este invocată ca funcţie "pură". Diferenţa majoră între cele două tipare este valoarea pe care o ia **`this`**. În primul caz, this reprezintă obiectul de care aparţine metoda. În cel de-al doilea, însă, this se referă la **obiectul global** (`document`, atunci când codul este rulat în browser) şi nu la obiectul în al cărui context se află funcţia. Acesta este o altă sursă frecventă de confuzii şi erori a limbajului. Analizaţi secvenţa de cod de mai jos pentru a vă face o idee mai bună asupra acestor concepte:

```
var myObject = {
    value: 0,
    increment: function (inc) {
        this.value += typeof inc === 'number' ? inc : 1;
    }
```

```
};

myObject.increment();
console.log(myObject.value);    // 1

myObject.increment(2);
console.log(myObject.value);    // 3

myObject.double = function () {
    var wrongHelper = function () {
        this.value = this.value + this.value;
    };

    wrongHelper();    // Invoke helper as a function.
};

// Invoke double as a method.
myObject.double();
console.log(myObject.value);    // 3
```

Celelalte două tipare sunt invocarea ca şi constructor şi invocarea prin metoda `apply`. După cum am discutat şi în secţiunea dedicată moştenirii, nu este recomandată folosirea operatorului `new` (implicit şi a invocării ca şi constructor), pentru că pot apărea erori destul de greu de depistat. Ultimul tipar de invocare este cel folosind metoda `apply`. Javascript permite funcţiilor să aibă metode la rândul lor. `apply` este o metodă, prezentă la toate funcţiile, ce primeşte doi parametri: primul reprezintă valoarea ce se va substitui prin `this` în corpul funcţiei, iar cel de-al doilea este un şir de argumente cu care se va apela funcţia.

```
var Quo = function (string) {
    this.status = string;
};

// Give all instances of Quo a public method
// called get_status.

Quo.prototype.get_status = function () {
    return this.status;
};

// Make an instance of Quo.

var myQuo = new Quo("confused");

console.log(myQuo.get_status());  // confused

var statusObject = {
    status: 'A-OK'
};

// we can invoke the get_status method on
// statusObject even though statusObject does not have
// a get_status method.

var status = Quo.prototype.get_status.apply(statusObject);
// status is 'A-OK'
```

Closures

Un alt aspect interesant al lucrului cu funcţii în Javascript este conceptul de închidere funcţională. Acest lucru înseamnă că o funcţie are acces la variabilele definite în contextul în care este declarată (mai puţin la `this` şi la `arguments`). `()` de pe ultima linie reprezintă faptul că lui `myObject` îi este asignată valoarea returnată de funcţia anonimă, şi nu funcţia efectivă. Metodele interne beneficiază de acces la variabila `value`, datorită închiderii funcţionale.

```
var myObject = function ( ) {
    var value = 0;

    return {
        increment: function (inc) {
            value += typeof inc === 'number' ? inc : 1;
        },
        getValue: function () {
            return value;
        }
    }
}();
```

Moştenire

Există mai multe moduri pentru a instanţia un obiect şi a defini o relaţie de moştenire:

- pseudoclasic (folosind operatorul **new**)
- prototipal (folosind proprietatea **prototype**, prezentă la toate obiectele)
- funcţional (mai complicat; permite variable hiding - o simulare de private -).

Moştenirea prototipală este metoda recomandată de reprezentare a moştenirii în Javascript. În exemplul de mai jos, este creată o metodă `beget`, ce va fi prezentă la toate obiectele şi care ascunde folosirea constructorului faţă de programator. Folosirea ulterioară a `beget` are ca rezultat "copierea" obiectului părinte în obiectul copil, urmând ca apoi să fie specificate doar diferenţele dintre cele două.

```
if (typeof Object.beget !== 'function') {
     Object.beget = function (o) {
         var F = function () {};
         F.prototype = o;
         return new F();
     };
}

var myMammal = {
    name : 'Herb the Mammal',
    get_name : function ( ) {
        return this.name;
    },
    says : function ( ) {
        return this.saying || '';
    }
};


var myCat = Object.beget(myMammal);
myCat.name = 'Henrietta';
myCat.saying = 'meow';
myCat.purr = function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
        if (s) {
            s += '-';
        }
        s += 'r';
    }
    return s;
};
myCat.get_name = function () {
    return this.says + ' ' + this.name + ' ' + this.says;
};
```

Moştenirea pseudoclasică oferă o sintaxă apropiată de cea a limbajelor familiare, însă reprezintă doar un nivel suplimentar de indirectare faţă de natura reală a limbajului (în loc ca obiectele să moştenească direct de la alte obiecte, se introduc constructorii).

```
var Mammal = function (name) {
    this.name = name;
};

Mammal.prototype.get_name = function ( ) {
    return this.name;
};

Mammal.prototype.says = function ( ) {
    return this.saying || '';
};

var myMammal = new Mammal('Herb the Mammal');
var name = myMammal.get_name(); // 'Herb the Mammal'

var Cat = function (name) {
    this.name = name;
    this.saying = 'meow';
};

// Replace Cat.prototype with a new instance of Mammal

Cat.prototype = new Mammal();

// Augment the new prototype with
// purr and get_name methods.
```

```
Cat.prototype.purr = function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
        if (s) {
            s += '-';
        }
        s += 'r';
    }
    return s;
};
Cat.prototype.get_name = function ( ) {
    return this.says( ) + ' ' + this.name +
            ' ' + this.says( );
};

var myCat = new Cat('Henrietta');
var says = myCat.says( ); // 'meow'
var purr = myCat.purr(5); // 'r-r-r-r-r'
var name = myCat.get_name( );
// 'meow Henrietta meow'
```

Javascript in aplicatii Web

Rolul Javascript. DOM.

Rolul Javascript în aplicaţiile Web este de a oferi programatorului un mod de a interacţiona cu browserul. Tot ce se întâmplă în pagina web **încărcată în browser**, se întâmplă prin execuţia de cod Javascript.

DOM-ul este o convenţie cross-platform şi cross-language de reprezentare a unui document XML, HTML sau XHTML ca obiect şi de manipulare a elementelor sale constitutive. Toate browserele oferă un API în Javascript de manipulare a DOM-ului. Acest lucru înseamnă că puteţi folosi Javascript pentru a face orice operaţie asupra conţinutului şi a modului de reprezentare al informaţiilor dintr-o pagină web **fără a mai fi nevoie să faceţi vreo cerere suplimentară la server**. Laboratorul următor va prezenta moduri diferite de utilizare al DOM-ului.

Deployment şi debugging

Codul Javascript se include într-o pagină web prin folosirea tag-ului `<script>`. El poate primi ca parametru (prin atributul `src`) un fişier, pe care îl încarcă printr-o cerere HTTP. Alternativ, codul Javascript se poate adăuga direct, ca şi conţinut al tag-ului.

Pentru debugging, recomandăm călduros folosirea Firebug. Firebug permite urmărirea valorilor variabilelor, setarea de breakpoint-uri, executarea de cod în consola sa, afişarea unor mesaje user-friendly în consolă, inspectarea DOM-ului, inspectarea HTML-ului (inclusiv comportamentul elementelor determinat de Javascript) dintr-o anumită pagină şi multe altele.

Există multe utilitare care fac o pseudo-compilare a codului Javascript pentru a prinde erorile frecvente, a impune restricţii asupra căror constructe ale limbajului este bine să fie folosite şi/sau pentru a optimiza codul. Cele mai cunoscute sunt JSLint şi Google Closure Compiler.

Task-uri

Descărcaţi această arhivă şi plasaţi folderul `lab08` pe serverul local.

1. Scrieţi o funcţie care primeşte ca parametru o variabilă şi returnează tipul său, dacă acea variabilă este definită şi are o valoare asignată. Luaţi în considerare şi cazul în care variabila reprezintă un array. Dacă valoarea sa e "falsy", returnaţi valoarea şi nu tipul. **(2p)**
   Tip: valorile "falsy" sunt: `0`, `NaN` (not a number), `undefined`, `null` şi `false`. Folosiţi operatorul `typeof` pentru a determina tipul unei variabile. Tipurile posibile sunt: `boolean`, `string`, `numeric`, `object`, `function`, `xml`.
2. Creaţi o structură arborescentă a cursurilor de la care aţi primit noţiuni care v-au folosit la cursul de PW şi stocaţi-o într-un obiect. Afişaţi-l. **(2p)**
   Tip: Pentru afişare, puteţi folosi cu încredere console.log (afişează în consola din Firebug).
3. Consideraţi secvenţa de cod din laborator:

```
var myObject = {
    value: 0,
    increment: function (inc) {
        this.value += typeof inc === 'number' ? inc : 1;
    }
};

myObject.increment();
console.log(myObject.value);    // 1

myObject.increment(2);
console.log(myObject.value);    // 3

myObject.double = function () {
    var wrongHelper = function () {
```

```
        this.value = this.value + this.value;
    };

    wrongHelper();    // Invoke helper as a function.
};

// Invoke double as a method.
myObject.double();
console.log(myObject.value);    // 3
```

Adăugaţi o funcţie `correctHelper` în corpul metodei `double` care să nu aibă aceleaşi probleme ca `wrongHelper`. **(2p)**

4. Este corectă următoarea funcţie dacă se vrea afişarea la click a indicelui din şirul de noduri transmis funcţiei? De ce? Corectaţi dacă e cazul. **(2p)**

```
var add_the_handlers = function (nodes) {
    var i;
    for (i = 0; i < nodes.length; i += 1) {
        nodes[i].onclick = function (e) {
            alert(i);
        }
    }
};
```

5. Scrieţi o funcţie care poate parsa un JSON simplu (o singură pereche cheie-valoare), fără a folosi `eval`. **(2p)**

```
// exemplu de input:
var simpleJSON = '{ "task": "6" }';
```

Tip: la acest link găsiţi informaţii despre lucrul cu stringuri în Javascript.

Bonus:

6. Implementaţi un algortim de sortare, la alegere între merge sort, insertion sort şi quick sort în Javascript. **(3p)**
7. Folosirea JSLint sau Closure compiler pe task-ul de la punctul 6. **(1p)**

Resurse

- Javascript: The Good Parts, Douglas Crockford

- Douglas Crockford: "The Javascript Programming Language", video 1 of 4
- Douglas Crockford: "The Javascript Programming Language", video 2 of 4
- Douglas Crockford: "The Javascript Programming Language", video 3 of 4
- Douglas Crockford: "The Javascript Programming Language", video 4 of 4
- Slide-uri prezentare

- John Resig: Learn Advanced Javascript: ~90 de exemple de cod

- wtfjs :)