Seminarul 4

Object prototypes, Classes & Exceptions

❖ Object prototypes

Prototipul este un obiect asociat implicit fiecărei **funcții** și **obiect** în JavaScript, unde proprietatea prototype a unei funcții este accesibilă și modificabilă, iar proprietatea (cunoscută și sub numele de atribut) a unui obiect nu este vizibilă.

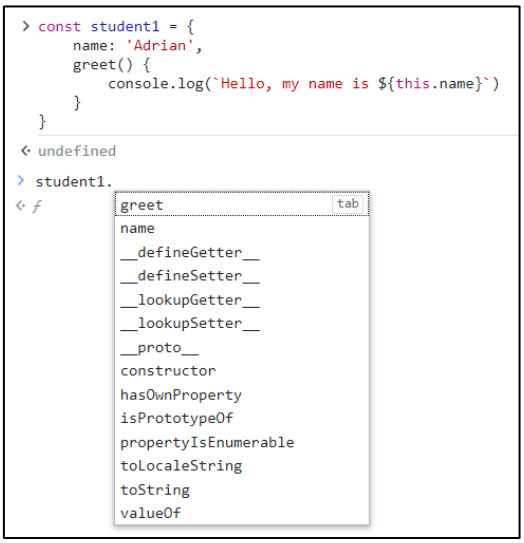
Fiecare funcție include în mod implicit un **obiect prototip**.

Prototipurile sunt o caracteristică puternică și foarte flexibilă a limbajului JavaScript, permițând **reutilizarea codului** și **combinarea obiectelor**. În special, ele susțin o formă de moștenire (inheritance).

Să încercăm să creăm acest obiect în consola de la browser:

```
const student1 = {
    name: 'Adrian',
    greet() {
        console.log(`Hello, my name is ${this.name}`)
    }
}
```

Acesta este un obiect cu o proprietate, "**name**", și o metodă, "**greet()**". Dacă tastați numele obiectului urmat de un punct în consolă, cum ar fi "student1.", atunci consola va afișa o listă a tuturor proprietăților disponibile pentru acest obiect.

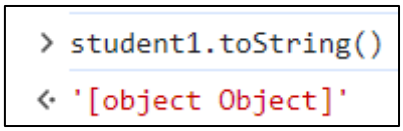


În JavaScript, fiecare obiect are o **proprietate** încorporată numită "**prototip**". Prototipul este, la rândul său, un obiect, astfel încât prototipul va avea propriul său prototip, creând ceea ce se numește o "**lanț de prototipuri**" (**prototype chain**). Lanțul se încheie atunci când ajungem la un prototip care are **null** ca prototip propriu.

Atunci când încercați să accesați o proprietate a unui obiect: dacă proprietatea nu poate fi găsită în obiectul însuși, se caută în prototip pentru proprietate. Dacă proprietatea nu poate fi găsită nici atunci, se caută în prototipul prototipului, și așa mai departe, până când fie proprietatea este găsită, fie se ajunge la capătul lanțului, caz în care se returnează **undefined**.

Dacă o sa încercăm de exemplu să tastăm în consolă comanda "**'student1.toString()**", browserul va executa următorii pași:

1. Va încerca să găsească metoda **toString** în obiectul **student1**
2. Deoarece nu o poate găsi aici, se va uita în **prototipul obiectului** pentru metoda **toString**
3. O găsește aici și o execută imediat



Pentru mai multe detalii legate despre prototipuri în JavaScript:
https://www.tutorialsteacher.com/javascript/prototype-in-javascript

❖ Classes

Clasele sunt șabloane pentru crearea obiectelor. Ele încapsulează date cu cod pentru a lucra cu acele date. Clasele în JavaScript sunt construite **pe baza prototipurilor**, dar au și unele sintaxe și semantici unice pentru clase.

Class declarations

Declarația de clasă este o modalitate tradițională de a defini clase în JavaScript, cu un bloc de cod care conține metode și proprietăți, **putând fi utilizată înainte de a fi definită în cod**.

```
class Student {
    constructor(name) {
        this.name = name;
        this.greet = function() {
            console.log(`Hello, my name is ${this.name}.`);
        }
    }
}
```

Class expressions

Expresia de clasă este o altă modalitate de a defini clase în JavaScript, fiind definită într-o expresie, care poate fi numită anonim sau atribuită unei variabile, **și nu este 'ridicată' (hoisted) la începutul domeniului său.**

```
const Student = class {
    constructor(name) {
        this.name = name;
        this.greet = function() {
            console.log(`Hello, my name is ${this.name}.`);
        }
    }
}
```

Hoisting

Hoistingul este comportamentul prin care **declarațiile** de **variabile (var)** și **funcții** sunt mutate (ridicate) în partea de sus a contextului lor. Cu alte cuvinte, puteți utiliza o variabilă sau o funcție înainte de a le declara explicit în cod. Cu toate acestea, este important să rețineți că doar **declarațiile în sine sunt mutate**, **nu și inițializările** (atribuirea de valori).

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

// La fel se aplică și pentru funcții
hoistedFunction(); // Va funcționa corect
function hoistedFunction() {
  console.log("Aceasta este o funcție ridicată.");
}

sayHello(); // Va arunca o eroare "TypeError: sayHello is not a function"

var sayHello = function() {
  console.log("Salut, lume!");
};

sayHello(); // Va afișa "Salut, lume!"

// Hoisting nu se aplică la inițializări
console.log(y); // Va arunca o eroare "ReferenceError: y is not defined"
let y = 10;
```

“Extends” keyword (Inheritance)

“Extends” în JavaScript este un cuvânt cheie folosit pentru a stabili o relație de moștenire între două clase, permițând clasei derivată (subclasă) să moștenească metodele și proprietățile clasei părinte (superclasă).

```
class Student {
    constructor(name, age, grade) {
        this.name = name;
        this.age = age;
        this.grade = grade;
    }
    greeting() {
        console.log(`Hello, my name is ${this.name} and I am ${this.age} years
old.`);
    }
}

class Teacher extends Student {
    constructor(name, age, grade, subject) {
        super(name, age, grade);
        this.subject = subject;
    }
    teach() {
        console.log(`I teach ${this.subject}.`);
    }
}
```

Din exemplul de mai sus: Metoda **super()** se referă la clasa părinte. Prin apelarea metodei **super()** în metoda constructor, apelăm metoda constructor a clasei părinte și obținem acces la proprietățile și metodele clasei părinte.

JavaScript call(), apply() & bind() methods

Scopul metodei **call()** în JavaScript este de a permite apelarea unei funcții cu un anumit context (this) și de a furniza argumentele funcției într-un mod explicit, separat. Aceasta este utilă în situațiile în care doriți să specificați în mod explicit valoarea this în interiorul funcției și să furnizați argumente individual.

```
const person = {
    fullName: function() {
        return this.firstName + " " + this.lastName;
    }
}
```

```
const person1 = {
    firstName: "John",
    lastName: "Doe"
}

person.fullName.call(person1);  // Will return "John Doe"
```

Scopul metodei **apply()** în JavaScript este similar cu cel al metodei call. Cu toate acestea, metoda apply permite apelarea unei funcții cu un anumit context (**this**) și furnizarea argumentelor funcției sub formă de array.

```
const person = {
    name: 'John Doe',
}

let greeting = function(a, b) {
    return `${a} ${this.name}. ${b}`
}

console.log(greeting.apply(person, ['Hello', 'How are you?'])); // Hello John
Doe. How are you?
```

Scopul metodei **bind()** în JavaScript este de a crea o nouă funcție care are un anumit context (this) permanent atașat și, opțional, anumite argumente fixate în mod permanent. Această funcție nouă, creată de bind, poate fi apoi apelată în orice moment fără a-și modifica contextul **this**.

```
const person = {
    name: 'John Doe',
}

let greeting = function(a, b) {
    return `${a} ${this.name}. ${b}`
}

let bound = greeting.bind(person);

console.log(bound('Hello', 'How are you?')); // Hello John Doe. How are you?
```

❖ Exceptions

Când apare o eroare, JavaScript se va opri în mod normal și va genera un **mesaj de eroare**. Termenul tehnic pentru aceasta este că JavaScript va arunca o **excepție** (va arunca o eroare). JavaScript va crea efectiv un obiect de eroare cu două proprietăți: **nume** și **mesaj**.

Proprietatea "name" a erorii poate returna șapte valori diferite:

1. **Eroare de sintaxă (Syntax error):**

```
const func = () =>
console.log(hello)
}
```

În exemplul de mai sus, lipsește o acoladă deschisă în cod, ceea ce declanșează constructorul de eroare de sintaxă.

2. **Eroare de referință (Refference error):**

```
console.log(x);
```

În exemplul de mai sus, lipsește definiția variabilei, ceea ce declanșează constructorul de eroare de referință.

3. **Eroare de tip (Type Error):**

```
let num = 15;
console.log(num.split("")); //converts a number to an array
```

În exemplul de mai sus, se încearcă o metodă specifică array-urilor pe un număr care nu suportă această metodă, ceea ce declanșează constructorul de eroare de tip.

4. **Eroare de evaluare (Evaluation error):**

```
try{
    throw new EvalError("'Throws an error'")
}catch(error){
    console.log(error.name, error.message)
}
```

Motoarele JavaScript actuale și specificațiile EcmaScript nu aruncă această eroare. Cu toate acestea, ea este încă disponibilă.

5. **Eroare de interval (Range error):**

```
const checkRange = (num)=>{
    if (num < 30) throw new RangeError("Wrong number");
    return true;}
checkRange(20);
```

6. **Eroare URI:**

```
console.log(decodeURI("%sdfk"));
```

În exemplul de mai sus, se va genera o eroare deoarece argumentul pasat funcției decodeURI nu este un URI valid, ceea ce declanșează constructorul de eroare de URI.

7. **Eroare internă (Internal error):**

```
switch(condition) {
    case 1:
    ...
    break
    case 2:
    ...
    break
    ... up to 500 cases
}
```

În motorul JavaScript, această eroare apare cel mai des atunci când există prea multe date și array-ul depășește dimensiunea critică. Atunci când există prea multe tipare de recursivitate, cazuri switch, etc., motorul JavaScript este supraîncărcat.

Try…Catch…Finally

Instrucțiunea **try...catch** este alcătuită dintr-un bloc **try** și fie un bloc **catch**, fie un bloc **finally**, sau ambele. Codul din blocul try este executat în primul rând, și dacă generează o excepție, codul din blocul catch va fi executat.

Sintaxă:

```
try {
    ...
} catch (exceptionVar) {
    ...
} finally {
    ...
}
```

De reținut:

1. Eroarea nu va fi gestionată automat, ceea ce înseamnă că programul se va opri și va afișa un mesaj de eroare în consolă. Acest lucru poate afecta funcționalitatea aplicației;
2. În absența unei structuri **try...catch**, erorile pot fi mai greu de depistat și de gestionat. Este important să fiți atenți la orice mesaje de eroare sau excepții care pot apărea în consolă.

Ex 1. Implementați funcția increaseSalary care primește ca parametri un array reprezentând o listă de salarii și un număr reprezentând procentul de creștere (de exemplu, 10). Funcția aruncă excepții dacă primul parametru nu este un array sau dacă al doilea parametru nu este un număr, altfel va genera un array cu noile salarii și îl va returna imediat, oprind execuția.

Ex 2. Implementați un tip obiectual care implementează un șir crescător având ca elemente toate numerele pare pornind de la o valoare dată. Constructorul primește valoarea inițială a secvenței. Singura metodă este 'next' care calculează următoarea valoare din șir.