JavaScript alapok helyesen

II. rész

Az előző részben megismertük a JavaScript primitív és összetett típusait, illetve néhány nyelvi sajátosságot és az esetleges problémákat, amik ezen nyelv alapjainak hiányos ismeretéből, illetve helytelen használatából eredhetnek. Ebben a részben néhány szót ejtünk az objektumokról JavaScriptben. Mivel a nyelv nagyon engedékeny és az objektumok sem a megszokottak, a helyes alapok elsajátítása hiányában sok negatív vélemény születik, illetve a későbbiekben nehézséget okoz ezek elsajátítása.

A felsorolt példákat továbbra is bármelyik böngésző JavaScript konzoljában ki lehet próbálni, amit az F12 (IE, FF, Chrome) vagy CTRL+SHIFT+I (Opera) billentyű lenyomásával lehet előhozni, vagy a JConsole [1.] internetes oldalon.

Objektumok általánosan

A továbbiakban leírtakhoz némi előzetes, alapvető objektumorientált paradigma (OOP) ismeret szükséges. Röviden tekintsünk át néhány alapfogalmat. Egy osztály (másképpen tekintve egy modell vagy makett) összetartozó adatok és az ezekkel elvégzendő műveletek egybezárt halmaza. Fő szerepük a programozásban az egyszerűsítés és átláthatóság mellett az adatok elrejtése és védelme a „külvilágtól", amiket meghatározott, az osztályhoz tartozó függvények segítségével lehet lekérdezni, illetve módosítani. Ezeket az adatokat adattagoknak nevezzük, míg a hozzátartozó műveleteket függvények írnak le, amelyeket metódusoknak nevezünk. A metódusokban levő kódrészletek segítségével tudjuk az osztályba tartozó adatok helyességét biztosítani, illetve megőrizni. Mivel egy osztály csak egy modell, azaz leír egy absztrakt dolgot, ezek használata a példányosítás segítségével történik, tehát az objektum egy adott osztály egy példánya. Következésképpen egy osztálynak több példánya lehet, ahol a példányokban szereplő adatok egymástól függetlenek. Példaként tekintsünk egy autó osztályra, amely leírja egy gépjármű néhány tulajdonságát, mint például márka, modell, évjárat és kilométer. Ez az osztály lesz minden gépjármű modellje, azaz minden gépjármű rendelkezik az adott adattagokkal, más néven tulajdonságokkal. Példányosításkor a létrejött objektumokban ezek a tulajdonságok konkrét értékeket kapnak, és természetesen példányokként az értékek nem függnek egymástól.

JavaScript objektumok

A megszokott programozási nyelvektől eltérően JavaScriptben a függvények egyben objektumok is, egy objektum létrehozásához viszont nem szükséges osztályt definiálni. Tekintsük a következő példát:

```
var pistike = {
 nev: 'Pistike',
 eletkor: 20,
 koszon: function(){ console.log('Szia!'); }
}
```

A fenti JSON (JavaScript Object Notation) [2.] objektumábrázolás egy, a JavaScriptből kiterjesztett, emberi szemmel olvasható és nyílt standardú adatformátum.

Egyszerűségének köszönhetően nagy népszerűségnek örvend, napjainkban már a legelterjedtebb adatformátummá vált, megelőzve az XML ábrázolást is. A programozási nyelvek nagy része közvetlen vagy közvetett (kiterjesztések segítségével) módon támogatja ezt a formátumot. Mint a legtöbb programozási nyelvben, az objektum adattagjainak elérése itt is a "." jelöléssel történik, viszont a tömbféle elérés is használható. Példa:

```
console.log(pistike.nev); // Pistike
pistike.koszon(); // Szia!

console.log( pistike['nev'] ); //Pistike
pistike['koszon'](); // Szia!
```

Eltérően a megszokott programozási nyelvek objektumaitól, a JavaScriptben lehetőségünk van egy objektum dinamikus (futási időben való) bővítésére tetszőleges adattagokkal illetve metódusokkal. Példa:

```
pistike.magassag = '170cm';
console.log(pistike.magassag); // 170cm

pistike.szorakozik = function() { console.log('Szorakozok...'); }
pistike.Szorakozik(); // Szorakozok...
```

Ugyanakkor egy objektum üres is lehet, azaz egyetlen adattaggal vagy függvénnyel sem rendelkezik, amit a későbbiekben természetesen bővíteni lehet.

JavaScript "osztalyok"

A következőkben áttekintjük a sokak számára legtöbb gondot és bonyodalmat okozó osztályokat JavaScriptben. Tekintsük a következő példát:

```
var Auto = function() { }
```

amivel létre is hoztunk egy osztályt. De miért nem függvény? A JavaScript konvenció szerint a függvénynév első betűmérete dönti el, hogy ez egy általános függvény vagy egy „osztály". Valójában JavaScriptben nincsenek a megszokott definíció szerinti osztályok, tehát a nagybetűvel kezdődő függvénynevek nem osztályok, se nem átlagos függvények, hanem ezek úgynevezett „gyártó" függvények (angolul: factory functions). A klasszikus osztályokhoz hasonlítva a gyártófüggvényeket ezen osztályok konstruktorainak lehet tekinteni, azaz olyan függvények, amelyek előállítanak egy objektumot. Más szóval sosem osztályokat írunk, hanem konstruktorokat. Ez sokak számára összezavaró, főleg ha figyelembe vesszük, hogy a typeof(Auto), az függvény típust mond. Az EcmaScript 2015-ben behozott „class" kulcsszó ennek tisztázására szolgál, viszont ez továbbra is a fent említett gyártófüggvényt takarja.

Mint a konstruktoroknak, a gyártófüggvényeknek is lehetnek paraméterei. Tekintsük a következő példát:

```
var Auto = function(evjarat){
 this.evjarat = evjarat;
 console.log('Auto letrejott');
}
```

A fenti példában a „this" kulcsszó a kontextust jelöli, amelyikben létrehozzuk az „evjarat" tulajdonságot, éspedig az éppen aktuális objektumot jelöli. A példányosítás a következőképpen történik:

```
var auto1 = new Auto(2015);
var auto2 = new Auto(2016);
```

```
var auto3 = new Auto(2017);
```

Az „auto1” változó típusát megvizsgálva a typeof(auto1) most már „object” típust mond, a console.log(auto1) esetén látjuk is, hogy ez egy JSON. Más szóval a gyártófüggvény létrehozott egy JSON objektumot. Bár a konvenció megköveteli, hogy a gyártófüggvény első betűje az nagybetű legyen, a legtöbb JavaScript motor esetén ez kisbetűvel is kezdődhet és hiba nélkül működik, viszont semmi garancia nincs arra, hogy ez mindig is fog, tehát egy esetleges későbbi hiba elkerülése érdekében ajánlott ezt a konvenciót betartani.

JavaScript engedékenysége miatt „new” kulcsszó elhagyása sem jár hibával, ezért a következő példa hiba nélkül lefut, viszont böngésző motorban futtatva egy nem várt eredményhez vezet:

```
var auto4 = Auto(2018);
console.log(auto4); // undefined. Hová lett az objektumunk?
console.log(window.evjarat); // 2018.
```

A „new” kulcsszó elhagyása miatt nem jön létre új objektum, ezért a „this” kontextus a böngésző „window” szuper-objektum lesz, és ebben jön létre az „evjarat” tulajdonság.

A továbbiakban nézzük meg, mi is történik egy új példány létrejöttekor. A „var auto1 = new Auto(2015)” sor mögött a következő három lépés rejtőzik:

1. memória lesz foglalva az új objektumnak, nevezzük ezt az instanciát „inst”-nek.
2. Auto.call(inst, 2015);
3. inst.__proto__ = Auto.prototype
4. visszaadja az új objektumot HA a függvénynek nincsen más visszatérési értéke

A második lépésben meghívódik a gyártófüggvény, és kontextusként átadódik az új memória terület. Ezen belül fognak létrejönni az adattagok és lesznek az értékek beállítva. A harmadik lépést a lentiekben, a JavaScript öröklődésnél tárgyaljuk részletesebben.

Öröklődés

Az OOP-nek az egyik fő pillére az öröklődés, ami egy osztályból egy másik osztály származtatását jelenti, azaz a tulajdonságok átöröklését egy szülő osztályból a gyerek osztályba. Ellentétben ezzel a megszokott osztálybeli öröklődéssel, JavaScriptben ez prototípus alapú. A prototípus is egy JSON objektum, amit minden példány örököl, ezzel örökölve minden adattagot és metódust ebből. Az osztályalapú öröklődéssel szemben, ami statikus, és emiatt korlátozott a prototípus alapú öröklődés JavaScriptben igencsak dinamikus. Mint minden objektumot, a prototípust is dinamikusan lehet módosítani.

A fenti példában a harmadik lépésben az újonnan létrejött objektum megkapja a gyártófüggvény („osztály”) prototípusát. Az objektumon ezt, *a __proto__* rövidített névvel érjük el, míg a konstruktoron, azaz a gyártófüggvényen ennek neve a teljesen leírt „prototype”. Könnyen észre lehet venni, hogy több létrehozott példány esetén a prototípus közös. Ezt egy nagy előnyként lehet értelmezni, főleg több ezer objektum esetén igencsak memóriatakarékos megoldás, viszont hátránya, hogy ennek módosítása az összes objektumot érinti.

Mit is jelent gyakorlatban a prototípus és hogyan lehet dinamikusan módosítani? Folytassuk a fenti példát:

```
Auto.prototype.km = 0;
Auto.prototype.megtesz = function(tavolsag) {
 this.km += tavolsag;
}
var auto5 = new Auto(2018);

console.log(auto5); // Auto {evjarat: 2018}
console.log(auto5.__proto__); // {km: 0, megtesz: ƒ, constructor: ƒ}
auto5.megtesz(100);
console.log(auto5); // Auto {evjarat: 2018, km: 100}
console.log( auto5.__proto__ === auto2.__proto__ ); // true (közös
prototípusok)
```

A fenti példa esetén az auto5 példány létrehozása után megvizsgálva azt láthatjuk, hogy csak az évjárat értéket tartalmazza, a „km" és a „megtesz" viszont csak a prototípusban szerepel. Viszont a továbbiakban a „megtesz" metódus meghívása után a „km" adattag már szerepel az objektumban is. Az adattagok és metódusok elérésekor először az osztályban lesznek keresve és utána a prototípusban. Egy adattag értékadása esetén viszont ez bemásolódik az aktuális osztályba, így megőrizve az objektumpéldányok közötti, adattagjainak egymástól való függetlenségét. Ugyanez érvényes a metódusokra is:

```
auto5.megtesz = function(){ console.log('mukodeskeptelen'); }
auto5.megtesz(100);
```

A fenti példában a megtesz függvény már nem a prototípusból hívódik, hanem a saját példányból. Ezt visszaállítani a prototípusbeli függvényre a helyi definíció törlését igényeli, ami a "delete" kulcsszó segítségével lehet:

```
delete auto5.megtesz; // true
```

Előfordul, hogy szükséges eldönteni egy adattagról vagy metódusról, hogy az illető osztály rendelkezik-e vele vagy a prototípusból örökli. Ezt a *hasOwnProperty(név)* segítségével lehet, ami igazat térít vissza ha az illető tulajdonság az osztályban van, és hamisat különben. Példa:

```
var auto6  = new Auto(2010);
auto6.hasOwnProperty('km'); // false
auto6.megtesz(10);
auto6.hasOwnProperty('km'); // true
```

Prototípus lánc

JavaScriptben minden objektum rendelkezik prototípussal, ami alapértelmezetten az *Object.prototype* lesz. Egymásba ágyazott objektumok esetén úgynevezett prototípus lánc alakul ki. Ilyen esetben egy adattag vagy metódus keresése az aktuális osztálytól kiindulva a legutolsó szülő prototípusa felé halad a láncon amíg nem lesz eredménye a keresésnek vagy el nem éri a legutolsó szülőt aminek a prototípusa *null*.

Memóriakezelés

Amennyiben szó esett a memóriafoglalásról, objektumok esetén érdemes megemlíteni ennek felszabadítását is. A legtöbb objektumorientált programozási nyelvhez hasonlóan a JavaScript is automata módon szabadítja fel a már nem használt memóriaterületeket. Egy terület akkor tekinthető nem használtnak, ha már egyetlen változó sem

tartozik hozzá. Objektum esetén a „delete” kulcsszó nem működik, és hamis érteket térít vissza, helyette elegendő, ha a hozzá tartozó változó értékét *null*-ra állítjuk (vagy egy más értékre). Például:

```
delete auto5; // false
auto5 = null;
```

Függvénykontextus

A következő rész már a programozásban jártas és a JavaScriptet jobban megismerni akaró személyeknek szól. A fentiekben láttuk, hogy objektum létrehozáskor a gyártófüggvény mint kontextus lesz hozzárendelve az új objektumhoz. A JavaScript közönséges függvények esetén is engedi a kontextus hozzárendelést, mint a következő egyszerű példában:

```
var jatekszer1 = { jatek: 'baba', eletkor: 2 };
var jatekszer2 = { jatek: 'szamitogep', eletkor: 18 };
var jatszik = function(gyerek) {
 console.log(gyerek + ' ' + this.eletkor + ' eves es kedvenc
jatekszere: '
 + this.jatek );
}
jatszik('Marika'); // Nem lesz jo
jatszik.call(jatekszer1, 'Marika'); // Marika 2 eves es kedvenc
jatekszere: baba
jatszik.call(jatekszer2, 'Pistike'); // Pistike 18 eves es kedvenc
jatekszere: szamitogep
```

Ha a megszokott módon hívjuk a „jatszik” függvényt, akkor a kontextus, azaz a „this”, a window objektum lesz, ezért a példában használt „this.eletkor” és „this.jatek” változók helyett „undefined”-t fog hozzáfűzni a kimenetben. A kontextus megváltoztatása a fent említett „.call(kontextus, paraméterek)” függvény segítségével történik.

Végkövetkeztetések

Manapság a JavaScript az egyik legelterjedtebb és legnépszerűbb programozási nyelv. A helyes alapok ismerete nélkül, főleg az interneten található mások példájából könnyen és rossz alapelveket lehet elsajátítani, amik nagy bonyodalmakat okozhatnak. Ezt a JavaScript engedékenysége, valamint a népszerű nyelvekben használt osztály alapú öröklődéstől eltérő prototípus alapú öröklődés is nagyban elősegíti. Egy programozási nyelv elsajátításához nem elég a szintaxist, valamint egyszerű példákat tudni, hanem ismerni kell az alapokat és az ezek mögött rejlő működési elveket is.

Referenciák

[1.] JavaScript online konzol, http://jsconsole.com
[2.] JSON specifikáció, https://www.json.org/

Filep Levente