JavaScript alapok helyesen

Bevezető

A JavaScript, gyakran csak JS-nek rövidítve, korunk egyik legnépszerűbb programozási nyelve [1, 2]. A HTML, CSS mellett a webfejlesztésnek egy alapköve, az elmúlt években számos más iparágban is elterjedt, például szerver oldali programozásban (NodeJS), web applikációkban és a Google Big-Query-ban is jelen van. Mára már a JavaScript egy megérett nyelvvé fejlődött, viszont az alapok helyes ismerete nélkül igencsak érthetetlenné tud válni. Ez mind az interpreterek engedékenysége, mind a nyelv fejlődése során a kompatibilitás szempontjából visszamaradt problematikus dolgoknak köszönhető.

A JavaScript története 1995-ben kezdődik, az akkor igen népszerű Netscape Navigator internetes böngészővel. A Microsoft-tal való versengésben szükség volt egy böngésző támogatta nyelvre. Fejlesztő cége, a Netscape Communications, először a Sun nagyvállalattal folytatott együttműködést, a Sun által fejlesztett Java nyelv beépítése érdekében a Netscape böngészőbe. Közben a Netscape nézőpontot változtatott, és úgy döntött, hogy a kívánt nyelv direkt a Java nyelvvel versengjen, ezért a szintaxisa hasonló kell legyen. Ezzel a feladattal Brendan Eicht bízta meg egy prototípus megalkotásával, ami 10 nap alatt el is készült az EcmaScript specifikációra épülve. Hivatalosan a nyelv először LiveScript néven jelent meg, majd később kapta a végleges, ma is ismert nevét, a JavaScriptet.

Régebbi fejlesztőknél előfordulhatott az a probléma, hogy nem tanulták meg a Javascript alapjait helyesen, egyrészt mert a nyelv nem volt még egy nagyon kifejlett nyelv, másrészt mert a webfejlesztés iparágban csak egy kiegészítő kellékként tekintettek rá; ezért soha nem vették a fáradtságot, hogy megtanulják a nyelv alapjait, hanem az interneten böngészve másolgatták a mások által esetleg hibásan megírt kódokat, így sajátítva el az „alapokat". Amikor viszont problémába ütköztek a rosszul elsajátított alapok miatt, mindig magát a nyelvet hibáztatták, ezért is alakult ki a webfejlesztők körében ilyen negatív vélemény a JavaScriptről. Vicces megjegyzésként sokszor emlegetett jellemző, hogy „ne használd a JavaScript és logika szavakat egy mondatban", amit szintén a rossz alapismeretek szültek. A JavaScriptben is vannak hibák, amik a visszafelé kompatibilitási szempontból megmaradtak, viszont ezeket ismerve el lehet kerülni az általuk okozott esetleges problémákat.

A továbbiakban leírtakhoz szükségesek alapvető programozási és némi objektum orientált ismeretek. Minden nyelvi alapot nem fejtek ki részletesen, csupán a nyelv furcsaságairól és sajátosságaiból eredő problematikus esetekről lesz szó.

A felsorolt példákat bármelyik böngésző JavaScript konzoljában ki lehet próbálni, amit az F12 (IE, FF, Chrome) vagy CTRL+SHIFT+I (Opera) billentyű lenyomásával lehet előhozni, vagy a következő internetes oldalon: http://jsconsole.com

2. Nyelvi alapok

A JavaScript szintaxisa nagyban hasonlít a C, Java szintaxisához. Több C nyelvbeli strukturális programozási szintaxist is támogat, mint például: if és switch kifejezések, for és while ciklusok, stb. A JavaScript egy nagyon flexibilis, dinamikus és gyengén típusos nyelv. Ez azt jelenti, hogy a változók típusát dinamikusan, az értékadásnál használt típus határozza meg.

Semmi akadálya nincs annak, hogy egy változó típusát, függvényt vagy objektumot futási időben lecseréljük. A JavaScript nyelv típusait két kategóriába lehet sorolni: primitív és összetett. A következőkben röviden tekintsük át ezeket.

2.1. Primitív típusok: logikai, numerikus, karakterlánc (String), undefined (speciális típus). Mivel dinamikus változókról van szó, nem kell semmilyen típust megadni deklaráláskor, hanem egyszerűen használjuk őket:

```
var x = true;
console.log(x); // konzolra való írás
console.log( typeof(x) ); // boolean
x = 10;
console.log(x); // 10
console.log( typeof(x) ); // number
console.log(x / "tiz"); // NaN (jelentése: Not a Number)
console.log(x / 0); // Infinity
x = "ez egy szam: ";
console.log( typeof(x) ); // string
console.log(x + 10); // ez egy szam: 10
var y;
console.log(y); // undefined
console.log( typeof(y) ); // undefined
var szoveg = "ez egy szoveg";
szoveg = 'ez is egy szoveg';
szoveg = 'es "ez" is egy szoveg';
```

Amint észrevehető, a JavaScript egy igen engedékeny és kényelmes nyelv. A nullával való osztás sem okoz hibát, az eredmény egyszerűen „végtelen". Az „undefined" különleges típus, ami azt jelzi, hogy a változó értéke még nem volt megadva, definiálva. A console.log(…) a JavaScript konzolra való írást valósítja meg, a typeof(…) operátor pedig egy változó aktuális típusát mondja meg.

2.2. Komplex típusok: függvény (function), objektumok (Object). A typeof „object"-et ad vissza mind az objektumokra, mind a tömbökre, és a „null" értékre is. Ez utóbbit nyelvi hibának lehet tekinteni, mivel a „null" megszokott jelentése a „semmi", „nem létezik". Ezt kompatibilitási okokból nem lehet javítani, ugyanis számos régebb írt kód nem működne többet helyesen, ha ez nem objektum típus volna.

```
typeof (null) // object
```

2.2.1. Függvények: többféle képpen lehet definiálni:

A

```
function proba() { console.log("proba fuggveny"); }
proba(); // proba
```

B

```
var proba = function() { console.log("proba fuggveny"); }
proba(); // proba
```

C

```
var proba = function proba() { console.log("proba fuggveny"); }
proba();
```

A függvényeket változóként is lehet használni. Legelterjedtebb használat a „B" példában feltüntetett; a „C" verzió az előző kettő kombinációja, ami redundáns, viszont az értelmező megengedi. Érdemes megemlíteni a névtelen függvényeket is. Ezeket névtelennek hívjuk, ha nem adjuk át őket egyetlen változónak sem, hanem definíció után rögtön meg is hívjuk. Hívás után meg is szűnnek létezni, többet nem lehet rájuk hivatkozni.

```
// „proba"  függvény definíció, ha elhagyjuk az értékadást akkor névtelen
lesz
var proba = (function(…) { … });
// Az utolso par zarojel a hivast biztositja
(function(){ console.log("Ez egy nevtelen fuggveny!"); }) ();
  // Paraméter átadás
(function(szam){ console.log("Ez egy szam: " + szam); }) (10);
```

2.2.2. Tömbök: jelölése [], tetszőleges típust tartalmazhatnak. Indexei mindig egész számok és nullától kezdődnek.

```
var tomb = [1, 2, 3]
console.log(tomb[0]); // 1
typeof(tomb) // object
var tomb = [1,2, function(){ console.log("ez egy tomb"); }]
```

2.2.3. Objektumok: jelölése {}, tulajdonságok és függvények csoportja. A közismert JSON adattípus megnevezés innen ered (JavaScript Object Notation).

```
var szemely = {
    eletkor: 20,
    nev: "Pista",
      olvas: function() { console.log("Olvas..."); }
  }
       console.log(szemely); // {eletkor: 20, nev: "Pista"}
       szemely.olvas();
       typeof(szemely.eletkor); // number
       typeof(szemely.olvas); // function
```

JavaScriptben a függvény és a tömbök is összetett típusok, ezért a következő műveletek megengedettek:

```
tomb.koszon = function() { console.log("Szia!"); }
tomb.koszon(); // Szia!
```

A tomb.koszon a tömbnek nem egy eleme lesz, hanem egy tulajdonsága!

```
proba.szam = 10;
proba(); // Fuggvenyhivas
typeof(proba); // function
typeof(proba.szam); // number
console.log(proba.szam); // 10
```

3. Nyelvi sajátosságok és ezekből eredhető problémák

3.1. Szintaxis

A következő példa egy olyan esetet ábrázol ahol a kódrészlet szintaktikailag helyes, semmi hibajelzést nem ad az értelmező, és mégsem úgy megy ahogy azt elképzeltük:

```
function proba(n) {
    if(n < 5) {
        return
            n * 2
    }
    else
        return n
}
proba(10); // 10
proba(2); // undefined
```

Miért nem megy a „2" paraméterrel? Hogy a problémát megértsük, és válaszolni tudjunk a kérdésre, először meg kell érteni mi történik. Amint említettük, a nyelv nagyon engedékeny és teljesen rendben van, hogy ne tegyünk pontosvesszőt a sor végére. Azonban ilyen esetben az értelmező úgy tudja megállapítani, hogy hol az utasítássor vége, hogy megnézi a következő sorban levő utasításokat. Amennyiben lehetséges, hogy ezek önálló utasításssorként létezzenek, akkor úgy tekinti, hogy az előző sornak vége is van. A mi esetünkben az n*2 lehet egy önálló utasítás, így az értelmező úgy veszi, hogy a „return" az egy egyedülálló utasítás, és ezt így hajtja végre. Mivel a visszatérési érték nincs megadva, ezért ez „undefined" lesz és ezt téríti vissza nekünk. Ha a „return" részt átírjuk a következő formára:

```
return n
    * 2;
```

akkor máris jól fog működni, mivel most a következő sorban a „* 2" értelmetlen mint önálló utasítás, ezért úgy fogja értelmezni, hogy az előző sornak a folytatása. Bár az interneten számos kódban előfordul a pontosvessző elhagyás, az ilyen fajta programozási stílus nem ajánlott. Egy több száz soros kódban a pontosvessző elhagyása által okozott hibát megtalálni nagyon nehéz és időigényes lehet.

3.2. Változók összehasonlítása

Adott a következő összehasonlítási példa:

```
var a = "1";
var b = 1;
var c = "1.0";

console.log(a == b); // true
console.log(b == c); // true
```

Logikából ha még emlékszünk, akkor ha A egyenlő B és B egyenlő C akkor kötelező módon A is egyenlő C-vel. Ha ezt kiprobáljuk, akkor:

```
console.log(a == c); // false
```

„hamis" értéket kapunk. Miért? Mivel a JavaScript egy gyengén típusos nyelv, ezért két változó összehasonlítása az értékük szerint történik. Először egy típuskonverzió történik, amely folyamán az egyik változót átalakítja a másik típusára és csak ezután tudja összehasonlítani az értéküket. Az első összehasonlításkor a „b" változó alakul át karakterlánc típusúvá, így az "1"="1" igazként lesz kiértékelve. Második összehasonlításkor a szöveges "1.0"-ból lesz numerikus 1-es, így az 1=1 is igaz. A harmadik összehasonlításkor viszont mind az „a" változó mind a „c" változó szöveges, így az "1"="1.0" hamis lesz. Ez más gyengén típusos nyelvekre is igaz, mint például a PHP. Helyesen az „==" és „!=" operátorok helyett az „===" és „!==" operátorok használata javasolt. Ezek a változók típusait is figyelembe véve hasonlítják össze őket. Az interneten ritkának számít ezen operátorok használata.

3.3. Változók láthatósága

Bár az eddigi példákban a „var" kulcsszóval deklaráltunk változókat, ennek a kulcsszónak a használata nem kötelező. Viszont, ha nem használjuk, akkor a nyelv úgy tekinti ezt mint globális változót. A használata azt jelenti, hogy az illető változó csak a helyi kontextusban lesz látható, mint például egy függvényen belül. Tekintsük a következő eseteket:

```
function proba() {
    var x = 10;
    console.log(x); // 10
}
console.log(x); // Hiba, a változó nem létezik

function proba(n) {
    for(i = 0; i < n; i++) - hianyzik a var kulcsszó
        proba2();
}
function proba2()
{
    i = 0;
}
proba(); // Végtelen ciklus
```

Miért lesz végtelen ciklus? Mivel a „for” ciklusban nem volt ott a „var” kulcsszó, ezért ezt globális változóként kezeli az értelmező. Ebben az esetben a proba2 függvényben ezt felül lehet írni, így mindig 0-os értéket kap itt, tehát a ciklus soha nem éri el az n változó értékét. Ami hibakeresés szempontjából rosszabb, hogy a nyelv bármilyen globális változót enged átírni. Az ilyen esetek okozta probléma javítása pedig egy igencsak megterhelő feladat, főleg terjedelmes kód esetén, mivel nagyon könnyű elsiklani fölötte, ugyanis ez nem hiba. Ezen eseteket kétféle képpen lehet elkerülni. Az első, hogy mindig használjuk a „var” kulcsszót. A második a "use strict"; direktíva használata. Ezt a szkript vagy függvény elején lehet megadni, és ennek hatására az értelmező korlátoz bizonyos lehetőségeket, mint például a nem deklarált változók használatát.

```
"use strict";
var x = 3.14; // Rendben van
y = 3.14; // Hiba, mert y nem volt deklarálva
```

3.4. Függvény paraméterek

A nyelv egy másik jellegzetessége vagy lehet szokatlanságnak is nevezni, a függvény paraméterek. Tekintsük a következőt:

```
var proba = function(a, b) { console.log(a + " , " + b); }
proba(1, 2);  // 1, 2
proba(1, 2, 3);  // 1, 2
proba(1); // 1, undefined
proba(); // undefined undefined
```

Mi is történik ilyenkor, ugyanis sem hibajelzést nem kapunk, sem figyelmeztetést. Eltérően a megszokott programozási nyelvektől, JavaScriptben a paraméterek száma nem definiáláskor rögzül, hanem híváskor dől el. Ez azt jelenti, hogy annyi paramétere van egy függvények amennyivel éppen meghívódik. Felmerül a kérdés, hogy hány paramétere lehet egy függvénynek? A válasz, hogy akármennyi. Az összes paramétert egy „arguments” tömbben kapjuk meg, míg a függvény paraméterei viszont csak megnevezései a tömb egyes elemeinek. Az „arguments” egyben kulcsszó is a nyelvben.

```
var proba = function(a, b) {
    console.log(arguments);
}

proba(4,5,6); // { `1`: 4, `2`: 5, `3`: 6 }

var proba = function() {
    for(var param in arguments)
        console.log(param);
}
proba(4,5,6); // Egyenkent irja ki a paramétereket
```

```
// Maximum szamot kiválasztó függvény
var maximum = function() {

    // Paraméterek számának ellenőrzése
    if(arguments.length == 0) {
        console.error("Nincs parameter!");
        return;
    }
      var max = arguments[0];
      for(var i = 0; i < arguments.length; i++)
          if(arguments[i] > max)
              max = arguments[i];

      return max;
}

maximum(); // Hiba
maximum(1); // 1
maximum(1, 2, 3, 2, 1); // 3
```

A fent megírt maximum függvény bármennyi paraméter esetén működik. Ez első ránézésre igencsak furcsának tűnik, hogy a függvény definiálásakor egyetlen paraméter sincs megadva neki és mégis helyes a kód és jól működik.

4. Tanulságok

A JavaScript egy igencsak engedékeny nyelv, sőt azt is lehet mondani, hogy túlságosan. Ebből kifolyólag, ha nem ismerjük jól az alapjait illetve bökkenőit, nagyon meg tudjuk nehezíteni a saját dolgunkat. A legfontosabb amit be kell tartani, az a „szép" és „tiszta" kódírás, mert ez mindig meghálálja magát.

Referenciák

[1] 2017 IEEE Spectrum felmérés, https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

[2] 2017 Stackoverflow felmérés, https://insights.stackoverflow.com/survey/2017

Filep Levente