**Optimalizálás**

Feladat

Módosítsd az elkészült űrlapot úgy, hogy a fizetendő díjba számítson bele 500 Ft házhoz szállítási díjat is, ha a rendelés értéke 5000 Ft alatt van!

Szervezd külön-külön függvényekbe a kódod a könnyebb áttekinthetőség végett! Legalább 3 függvényed legyen:

* Az első kommunikál a felhasználói felülettel: leolvassa az űrlapadatokat és kiírja a fizetendő összeget.
* A második validálja a kapott adatokat. Hibás vagy hiányzó adat esetén jelzi a hibát. (Például felugró ablakban.)
* A harmadik számítja ki a fizetendő összeget.

Ennél több függvényt is készíthetsz, ha szeretnél. Például az adatok validálása történhet akár űrlapelemenként is.

Függvények - innen nincs visszaút

Ha megismered a függvények korlátlan világát, végleg függő leszel ;-)

Mi a függvény?

A függvény egy egyszerű gép. Bedobáljuk a hozzávalókat, elvégzi a feladatát és a végén kidobja az eredményt.

Emlékszel még második osztályban? Bedobom a gépbe a hármat és a négyet, tizenkettőt dob ki. Mit csinál a gép? Nyilván összeszorozza a két kapott paramétert. Ugyanígy lehet elképzelni a függvényeket.

**Hivatalosan: azokat a kódrészleteket amelyeket gyakran használunk, kiemelünk függvényekbe.**

üggvények használata

Ahhoz, hogy használni tudj egy függvényt, legalább két lépésre van szükség:

* **Declaration:** megadom a függvény nevét és leírom a működését.
* **Invocation:** a kód megfelelő helyén meghívom (elindítom) a függvényt.

Deklarálás

Függvényt a function kulcszóval hozhatsz létre. Két lehetőséged van:

* function testFunction() {...} : ez a függvény deklaráció,
* let testFunction = function() {...} : ez a függvény expression (úgy adod meg, mint egy változót).

A két megadási mód között nincs működésbeli különbség. A teljes szintaxis:

function name(parameter1, parameter2, ..., parameterX) {

// a futtatható kód

}

Invocation - függvény meghívása

Az invocation, azaz a függvény meghívása azt jelenti, hogy elindítod a függvényt, az pedig végrehajtja a feladatát. Ezt a függvény neve után tett zárójelekkel teheted meg. Ezt hívják függvényhívásnak, azaz **call**-nak:

function myFunction(a, b) {

alert("Hello Dude!"); // megjelenít egy üzenetet amikor meghívják a függvényt

}

myFunction(); // az üzenet megjelenik

Függvény vagy eljárás?

Ha egy függvény nem ad vissza semmilyen adatot miután végzett a feladatával, azt eljárásnak szoktuk pontosan hívni. A függvények a futásuk végén visszaadnak valamilyen értéket. Ezt a return kulcsszóval tudod meghatározni:

function myFunction(a, b) {

return a \* b; // a függvény visszaadja az a és b szorzatát

}

var x = myFunction(4, 3); // tehát az eredmény 12 lesz, ami bekerül az x változóba

## Clean Code

Ahhoz, hogy a kódod később is ismerős legyen, illetve hogy mások is megértsék, fontos hogy tisztán és átláthatóan dolgozz. A változóknál már foglalkoztunk kicsit a jó nevekkel, itt most kibővítve vesszük át a dolgot.

**Fontos:** célunk az, hogy olyan kódot írjunk, ami úgy olvasható mintha mondatokat olvasnánk.

## Elnevezési szabályok

A változók és függvények nevei alapvető fontosságúak egy kódban. Nem kell sajnálni a karaktereket arra, hogy beszédes neveket válassz.

### **Beszédes nevek**

let n; // nem jó

let nrOfProducts; // nem jó

let numberOfProducts; // jó

Nincs szükség kommentekre, ha kifejező az elnevezés.

### **Ne legyen a név félrevezető**

function adder(a, b) {

return a \* b;

}

Nevezd át az osztályt, metódust, változót, stb. amikor a jelentése változik.

### **O és I kerülése**

var O;

var l;

A kisbetűs L és nagybetűs O használata is félreinformálhat, mert úgy néznek ki, mint a 0 és az 1.

### **Használj könnyen kiejthető neveket**

function getYYYY() { // rossz

return new Date().getYear();

}

function getYear() { // jó

return new Date().getYear();

}

## Kommentek

### **Ritkán írj kommenteket!**

* Ha sűrűn jelennek meg kommentek a kódban, elkezdünk nem törődni velük.
* Tartogasd a kommenteket olyan speciális esetekre, amikor a figyelem felhívására van szükség.
* A jó komment olvasója hálás lesz, hogy az a komment megíródott.

### **Informatív komment**

Valóban plusz inforációt közöl az olvasóval:

// matches any alphabetic character between m and n

var idPattern = /m[a-z]n/;

## Formázás

Fontos a formázás? Amikor a forráskódod olvasója ránéz a kódodra, akkor először akaratlanul is annak formázását figyeli meg (és nem a neveket vagy valami mást). A formázás a kommunikációról szól. Egy jól "kommunikáló" kódot könnyű változtatni és karban tartani, tehát a cégnek pénzt takarít meg.

### **Mi számít a formázásnál?**

* Fájl méret
* Függőleges formázás
* Vízszintes formázás
* Indentáció

### **Fájl méret**

* Rövid fájlméreteket tarts fenn!
* De mi számít rövidnek? A statisztikák arra mutatnak rá, hogy a modern szoftverek átlagosan 50-60 soros fájlokat tartalmaznak, míg más szoftverek 200-at.

### **Függőleges formázás - Üres sorok elválasztó eszközként**

class People {

constructor(name) {

this.name = name;

this.initPeople();

}

initPeople() {

//

}

}

A változók definícióit a konstruktorban elválasztottam a metódus hívásoktól.

### **Vízszintes formázás – sorok hossza**

Tartsd a sorok hosszát maximum 80 karakternél. Így nem kell soha vízszintesen görgetni amikor a kódot olvasod.

### **Indentáció**

Az indentáció azt jelenti, hogy egyes sorok beljebb kezdődnek a többinél.

* A szóköz használata jobb, mert minden szerkesztőban ugyanakkora lesz, függetlenül a tab méretétől.
* 2 vagy 4 szóköz is lehet.
* De: használja mindenki ugyanazt az indentációs stílust a csapatban.

## Függvények

Egy függvény egy és csakis egy dolgot csináljon! Akkor jó a függvény, ha már nem lehet értelmesen több másik függvénybe kiszervezni a működését.

* 4 - 6 sor az ideális hosszúság, 10 már sok.
* A függvények nevénél igét vagy ige szerkezetet válassz.
* A függvénynek maximum 3 argumentuma - paramétere legyen.

## Összefoglalás

A fenti szabályok a legtöbb esetben hasznosak amikor dolgozol. De sokszor előfordul, hogy a csapat más szabályokban egyezik meg praktikussági okokból, ebben az esetben természetesen azok az irányadóak.

Paraméterre fel!

Ahogy a metódusoknál láttad, a függvényeket paraméterekkel tudod ellátni.

**Paraméter - argumentum:** bemenő adat, amellyel a függvény műveleteket végez.

Paraméterek sorrendje

A paraméterek ugyanabban a sorrendben kerülnek feldolgozásra, ahogyan azt megadtad a függvény létrehozásánál. Ezért vigyázni kell, ne hagyj ki paramétereket, mert nem ugyanazt fogja érteni a függvény, mint amit megadtál:

var name = 'Joe';

var job = 'programmer';

function showPerson(name, job) {

document.querySelector('.card').innerHTML = 'Név: ' + name + ', munka: ' + job;

}

showPerson(name, job); showPerson(job, name);

Az első esetben azt írja ki: Név: Joe, munka: programmer  
Majd utána: Név: programmer, munka: Joe

Arguments - Paraméterek ellenőrzése

Az arguments tömb tartalmazza a függvény által kapott paramétereket. Ezzel le is tudod ellenőrizni, hogy megfelelő paramétereket kaptál-e.

Default paraméter

A JavaScript legújabb verziójában már vannak úgynevezett alapértelmezett vagy default értékei a paramétereknek, de ezt még nem minden böngésző támogatja. Ezért magunknak kell leellenőrizni a paramétereket, és ha nem megfelelő az értékük, akkor beállítani valami értelmesre:

var name = 'Joe';

var job = 'programmer';

var card = document.querySelector('.card');

function showPerson(name, job) {

name = name || 'Balázs';

job = job || 'admin';

card.innerHTML = 'Név: ' + name + ', munka: ' + job;

}

showPerson();

**Az új szabvány szerint:**

function showPerson(name = 'Balázs', job = 'admin') {

card.innerHTML = 'Név: ' + name + ', munka: ' + job;

}

showPerson();

Scope

Azt jelenti angolul, hogy terület, hatáskör, esetleg kiterjedés. A változók scope-ja az a része a programkódnak, ahol a változóval lehet dolgozni, el lehet érni. Azaz egy változó nem használható a teljes programban, csak a scope-jában. Kifejtem részletesebben, figyelj.

Egy alapvető szabály

**A scope-ból kifelé mindig lehet látni, de befelé nem.**

Mit jelent ez? Képzeld el, hogy egy olyan fólia van az ablakon, amitől látod, hogy mi van kint, de kintről nem látnak be. Pont így működik a js scope is. Egy scope -ban az összes változóval tudsz dolgozni, amit a sope -on kívül hoztak létre, de ha a scope -on belül van egy másik scope, abba nem látsz bele. Tehát a gyerek scope változóival nem tudsz dolgozni. A követkeőkben változó fajtánként is kifejtem neked a dolgot.

A var scope

A js 2015 előtti verziójában egyféleképpen lehetett változókat létrehozni a var, azaz variable kulcsszóval.

**function scope:** A var paranccsal létrehozott változó függvény hatáskörű volt, ez azt jelenti, hogy vagy a kód gyökerében, vagy abban a függvényben lehetett csak használni, ahol létrehozták. Ha máshonnan akartad elérni, akkor hibát kaptál. Tehát a függvényekben létrehozott változóid nem voltak elérhetőek (láthatóak) a függvényen kívül.

**local scope:** A függvény saját scope-ját lokális scope-nak is nevezzük.

**global scope:** A függvényen kívül eső területet pedig globális scope-nak hívjuk. Az, hogy egy scope globális vagy likális ugye attól függ, hogy honnan nézed? Erre hozok most néhány példát.

Lokális változó definiálása

// Global scope: a carName változót itt nem használhatod

function myFunction() {

var carName = "Volvo"; // Local scope: Itt használhatod a carName változót

}

myFunction();

console.log( carName ); // Uncaught ReferenceError: carName is not defined

**Mit láttál most?**

* Létrehoztam egy függvényt, amiben definiáltam a carName változót.
* Le is futtattam a függvényt, így a carName változó biztosan létre is jött a függvényben.
* Megpróbáltam kiíratni a console-ra a változót, de a kommentben látható hibát kaptam.
* Milyen hiba? Uncaught ReferenceError : váratlan vagy kezeletlen referenciahiba. Azaz olyan dologra - változóra hivatkoztam, olyan változóval akartam dolgozni, ami nem létezik. carName is not defined szép passzív angol szerkezettel meg van mondva: a carName változó nincs definiálva, nem lett létrehozva. Azt mondod, dehogynem, hát a függvényben ott van. Igen ám, de a függvénybe nem látunk bele a globális scope -bó, tehát számunkra nem létezik ott, ahol a log parancsot kiadtam.

Globális változó definiálása

A globális változó előnye, hogy a kódon belül bárhol eléred. Hátránya, hogy így a program több memóriát fogyaszt. Miért is? Hát azért mert amikor egy függvényt elindít, akkor át kell adnia neki az összes globális változót, mert nem tudhatja hogy használni akarod-e valamelyiket? Ezért ha lehet, mindig lokális változókkal dolgozz.

// Global scope: a carName változót itt használhatod, mert itt hoztad létre

var carName = "Volvo";

function myFunction() {

// Local scope: Itt is használhatod a carName változót, mert globális

console.log( carName ); // "Volvo"

}

myFunction();

console.log( carName ); // "Volvo"

**Fontos:** ugyanaz a példa, de a carName globális változó. Mind a két alkalommal el tudom érni, a függvényben és a globális scope-ban is.

Let

A 2015-ös js szabványban hozták be a let kulcsszót. A tanfolyamon ezt használjuk már a var helyett, mert hatékonyabb és takarékosabb mint a másik.

A blokk

A js kód blokkokból épül fel. Ezeket kapcsos zárójelekkel {} jelölik. Egy tipikus blokk ilyen:

{ var x = 2; }

let scope

A let kulcsszóval létrehozott változók block scope-úak, egy blokkon belül lehet őket elérni. Ez fontos különbség, mivel így a változó kisebb kódrészben érhető el. Viszont amikor az adott blokk lefut, hamarabb is törlődik a memóriából, így takarékosabb tud lenni a program.

Block scope

A {} zárójelekkel határolt programterület.

{ var x = 2; let y = 3; }

console.log( x ); // 2

console.log( y ); // Uncaught ReferenceError: y is not defined

**Lássuk csak!**

* Az x változót lazán eléred a blokkon kívül is, mivel az function scope-ú, és nem hoztál létre új függvényt, a hagyományos function scope alapján ugyanabban a scope-ban van.
* Viszont az y esetén hibát kapsz, mert az block scope-ú és egy blokkban hoztad létre, ahova a global scope-ból nem látsz bele.

let módosítása

A let kulcsszóval létrehozott változóknál nem lehet a változót kétszer létrehozni egy scope-ban.

**Blokkon belül nem lehet újradeklarálni:**

let name = "Péter";

let name = "Sándor"; // Uncaught SyntaxError: Identifier 'name' has already been declared

A gondja az, hogy már létre lett hozva azonos néven egy változó, azaz az azonosító már deklarálva van.

**Blokkon belül lokális változó jön létre:**

let name = "Péter";

{

let name = "Sándor";

console.log( name ); // "Sándor"

}

console.log( name ); // "Péter"

Először azt írja ki, hogy "Sándor", majd azt hogy "Péter". A blokkon belül létrehozott name változó nem azonos a globális name változóval. Tehát nem a globális változót hoztam létre újra, hanem a blokknak saját változója van.

Constant

Ez az angolban is a latinból jön (mint annyi minden más is). Azt jelenti, hogy állandó. Olyan változóról van szó, aminek egyszer lehet csak értéket adni, amikor létrehozod. Később már az értékét nem tudod megváltoztatni. Ilyen értelemben nem is változó, mert nem tudod változtatni. Azért neveztem mégis így, mert a js esetén bizonyos keretek között módosítható (erről majd a tömböknél és az objektumoknál lesz szó).

A const keyword

Ha egy változót a const kulcsszóval hozol létre, akkor nem adhatsz neki új értéket.

const PI = 3.141592653589793;

PI = 3.14; // Uncaught TypeError: Assignment to constant variable.

PI = PI + 10; // Uncaught TypeError: Assignment to constant variable.

Szabályok

* Mindig nagy betűvel írjuk: PI, START, BASE\_COLOR, stb...
* Mindig értéket is kell neki adni amikor létrehozzuk: const BASE\_COLOR = 255;

A const scope

**Block scope:** a const a let-hez hasonlóan szintán blokk szintű változó, azaz egy blokkon belül látható. Ha a kód gyökeréban hozod létre akkor mindenhol, ha nem akkor a saját blokkján belül.

const x = 10;

{

const x = 2;

console.log( x );

}

console.log( x );

A let-es példához hasonlóan itt is különböző számokat fogsz látni, a blokkon belül egy másik x van, aminek az értéke 2, a globális scope-ban pedig a 10 értékű.

const redeclare

Na ez az ami nem fog menni. Az alábbi példában azt láthatod, hogy ha egyszer egy blokkon belül létrehoztál egy const változót, akkor azt semmilyen formában nem tudod újra létrehozni, sem újra definiálni azaz új értéket adni neki (persze az adott blokkon belül, ha nyitsz új blokkot akkor igen).

const x = 2; // Rendben

const x = 3; // Hiba

x = 3; // Hiba

var x = 3; // Hiba

let x = 3; // Hiba

{

const x = 2; // Rendben

const x = 3; // Hiba

x = 3; // Hiba

var x = 3; // Hiba

let x = 3; // Hiba

}

Ezt jól jegyezd meg!