JavaScript
I. KÖNYV: ALAPOK

I. JavaScript – Bevezetés

A JavaScript (a továbbiakban JS) a legelterjedtebben használt internetes programozási nyelv, melyet a legtöbb böngésző (pl. IE, Firefox, Chrome, Opera, Safari) támogat.

Előismeretek

A JS elsajátításához szükséges a HTML/XHTML nyelv legalább alapfokú ismerete.

Mi a JavaScript?

A JavaScript egy Internetes felhasználáshoz idomított ún. scripting language (=parancsleíró nyelv), vagyis voltaképpen egy (leegyszerűsített) programozási nyelv, mellyel interaktívvá tehetjük weboldalainkat. A JavaScriptet a (bonyolultabb) C programozási nyelvből alakították ki. A JS ún. interpretált nyelv, azaz a parancsait a program az erre kifejlesztett program gépi kódra való lefordítás nélkül hajtja végre.

A JS-et általában közvetlenül a HTML-fájlba írjuk bele.

A JS-et bárki ingyenesen használhatja.

Java = JavaScript?

Nem! A Java és a JavaScript két, alapgondolataiban és megjelenésében teljesen eltérő programozási nyelv. A Sun Microsystems által kifejlesztette Java a JS-nél sokkal hatékonyabb és bonyolultabb nyelv, s így inkább a C ill. C++ -hoz hasonló.

Mire jó a JavaScript?

- **A JS a webes fejlesztők programozási eszköze.**

A HTML-oldalak készítői gyakran nem programozók; így számukra ideális programozási lehetőséget nyújt az igen egyszerű mondattannal rendelkező JavaScript. A JS segítségével tehát a programozáshoz nem értők is bővíthetik néhány „program-töredékkel" weboldalukat.

- **A JS-tel dinamikus (szöveg)tartalommal tölthetjük fel a weboldalakat.**

Az alábbihoz hasonló JavaScript-utasításokkal (=statements) az oldal bizonyos (pl. szöveges) elemeinek tartalmát a weboldal aktuális állapotához igazíthatjuk:

```
document.write("<h1>" + name + "</h1>)
```

- **A JS-tel az oldal reagálhat bizonyos eseményekre**

A JS megfelelő beállításával bizonyos történésekhez (pl. az oldal letésének befejeződése, vagy a felhasználó elemre-kattintása) az oldal tartalmának megváltozásában álló választ rendelhetünk.

- **A JS kezeli (olvassa és felülírja) a HTML-tartalmat**

A JS működés közben beolvashatja/felhasználhatja ill. módosíthatja egyes HTML-elemek tartalmát.

- **A JS adatok validálására is használható.**

A JS segítségvel leellenőrizhetjük a felhasználó által (pl. űrlapon) küldendő adatok helyességét a továbbításuk előtt. A kiszűrt hibás üzenetek nem terhelik feleslegesen a szervert.

- **A JS felismeri a felhasználó böngészőprogramját.**

A JS segítségével felismerhetjük a felhasználó böngészőjének típusát, és a szerver az azonos tartalmú, de eltérő böngészők számára készült weboldal-variánsok közül a legmegfelelőbbet továbbíthatja neki.

- **A JS alkalmas cookie-k létrehozására**

A JS-segítségével ún. cookie-k formájában adatokat tárolhatunk és kereshetünk vissza a weboldal látogatójának számítógépén/számítógépéről. A (HTTP-/web-/browser-)cookie (=keksz) nem más, mint a webbönglsző által a felhasználó gépén eltárolt szöveg. Ezt a későbbiekben bejelentkezéseknél, keresési és böngészési javaslatokhoz, bevásárlókosár-tartalom megőrzésére egyaránt felhasználhatjuk, de az éppen látogatott weboldalak tartalmának megőrzésére és a gép váratlan leállását követő visszaállítására is

alkalmas.

A JavaScript pontos elnevezése ECMAScript
A JavaScript az ECMASCript nyelvi normára épülő alkalmazás (=implementáció). Az ECMAScriptet az ECMA International fejleszti és működteti, amely egy nemzetközi kommunikációtechnológiai és szórakoztató elektronikai szabványügyi szervezet. A hivatalos JavaScript szabvány jele ECMA-262.

Az ECMAScript nyelvet Brendan Eich, a Netscape munkatársa fejlesztette ki (a Navigator 2.0-ra), majd 1996 óta minden Netscape és Microsoft böngészővel használhatóvá vált.

Az ECMA-262 kifejlesztése 1996-ban kezdődött, az első változatát pedig az ECMA 1997. júniusi Általános Közgyűlésén fogadták el.

A szabványt 1998-ban az ISO is bejegyezte, ISO/IEC 16262 szám alatt. Fejlesztése jelenleg is folyik.

II. JavaScript – módszertan

A JavaScriptet a **<script>** HTML tag segítségével írhatjuk bele a weboldalba.

JavaScript írása weboldalba

Az alábbi példában egy szöveget íratunk ki a weboldalra a JS segítégével:

```
<html>
<body>

<script type="text/javascript">
document.write("Hello World!");
</script>

</body>
</html>
```

A fenti HTML-oldalon csupn a **Hello World!** szavak olvashatók (idézőjel nélkül). Ha az idézőjeleknek akár csak egyikét is kitöröljük, vagy az utasításban szereplő pontot kettőspontra változtatjuk, a böngésző nem ír ki semmit.

Második példánkban látjuk, hogyan adhatunk HTML-tageket a megjelenítendő tartalomhoz:

```
<html>
<body>

<script type="text/javascript">
document.write("<h1>Hello World!</h1>");
</script>

</body>
</html>
```

Ebben az esetben a szöveg elsőrendű címsorként jelenik meg.

A példát átalakítva linket is kiírathatunk:

```
<html>
<body>

<script type="text/javascript">
document.write("<h2><a href=http://www.lap.hu target=_blank>Hello
World!</a></h2>");
</script>

</body>
</html>
```

Mint látszik, az **<a>** tag attribútum-értékeit (**http://www.lap.hu** ill. **_blank**) nem tettük idézőjelbe, mert ellenkező esetben a program megzavarodik, és nem ír ki semmit (ti. maga a JS utasítás értéke is idézőlejben van).

Magyarázat

Mint első példánkban láttuk, a JS-nek a dokumentumba ágyazására a **<script>** tag szolgál, melyben a **type** attribútummal a parancsnyelv fajtáját is meg kell adnunk. Így a **<script type=”text/javascript”>** és a **</script>** tagek jelzik a JS kezdetét és végét a dokumentumban:

```
<html>
<body>
<script type="text/javascript">
...
</script>
</body>
```

```
</html>
```

A **document.write** egy standard JS-parancs, mellyel szöveget íratunk ki az oldalra. Hogyha ezt az előbb bemutatott **script** elembe írjuk, a böngésző JS-parancsként fogja értelmezni, és végrehajtja a parancssort, azaz ebben az esetben kiírja a **Hello World!** szöveget:

```
<html>
<body>
<script type="text/javascript">
document.write("Hello World!");
</script>
</body>
</html>
```

Hogyha a parancssort nem **<script>** tagbe írjuk, akkor azt a böngésző egyszerű szöveges tartalomként kezeli, és az egészet kiírja az oldalra. Tehát pl. a következő esetben:

```
<html>
<body>
document.write("Hello World!");
</body>
</html>
```

a **document.write(”Hello World!”);** felirat jelenik meg a képernyőn.

Megoldás az egyszerű böngészők számára

A JavaScriptet nem támogató böngészők a JS rendelkezéseket (=statements) az oldal-tartalom részeként jelenítik meg, ami zavarólag hat. Ennek kiküszöbölésére, a JS szabvány értelmében a HTML comment (**<!--->**) taget alkalmazzuk az elrejtésükre.

Azaz egyszerűen írjunk egy **<!--** taget a JS rendelkezések elé, és egy comment-záró **-->** taget utánuk, így:

```
<html>
<body>
<script type="text/javascript">
<!--
document.write("Hello World!");
//-->
</script>
</body>
</html>
```

A két, jobbra dőlő perjel a comment-záró tag előtt a JS-megjegyzések jelölése, azaz az ezen jelek közé írt rendelkezéseket a program nem veszi figyelembe. A HTML comment-lezáró tag elé írt kettős perjel értelmében a JavaScript az oldal teljes további tartalmát megjegyzésként kezeli, és nem hajtja végre.

JavaScript – módszertan:

A JS-et tartalmazó <script> tag:

```
<script type="text/javascript">
        JavaScript – rendelkezések
</script>
```

Szöveg-kiíratő parancs: document.write(”*Szöveg*”);

JS elrejtése nem támogató böngésző esetére (HTML comment taggel):

```
<script type="text/javascript">
        <!--
        JavaScript – rendelkezések
        //-->
</script>
```

III. A JavaScript elhelyezése a weboldalban

A JS-et a dokumentum fejrészébe ill. szövegtestébe egyaránt beleírhatjuk.

A JS elhelyezése a weboldalban

A HTML-oldalba írt JavaScripteket a böngésző az oldal betöltésekor azonnal végrehajtja.

Előfordulhat azonban, hogy máskorra kell időzítenünk őket, pl. amikor a felhasználó megnyom egy gombot. Az utóbbi esetben egy ún. függvénybe (=function) kell beírnunk a JS-rendelkezéseket, amiről egy későbbi fejezetben lesz szó.

JS-rendelkezések a <head> részben

Azon JS-rendelkezéseket, melyeknek csak megfelelő felhasználó-oldali akcióra, „hívásra" szabad végbemenniük, az előbbiek szerint függvényekbe írjuk; amelyket a dokumentum fejrészébe írva elkülöníthetünk a szövegtestbe írt HTML, CSS és JS elemektől és rendelkezésektől.

Következő példa-oldalunk betöltésekor egy párbeszédpanelben a **This alert vox was called with the onload event** felirat jelenik meg:

```
<html>
<head>
<script type="text/javascript">
function message()
{
alert("This alert box was called with the onload event");
}
</script>
</head>

<body onload="message()">

<p>We usually use the head section for functions (to be sure that the
functions are loaded before they are called).</p>

</body>
</html>
```

Mint látjuk, a **<body>** elem betöltésekor a böngésző a **message** függvényben szereplő **alert** parancs tartalmát iratja ki egy párbeszédpanelben. Természetesen a függvényt **message** helyett másként is elnevezhetjük (pl. **sanyi**), elegendő az elnevezést a **<body>** elemben ill. a fejrészben átírni.
Mint látjuk, a függvényet azért is a **<head>** részbe írjuk, hogy a böngésző még a szövegtest betöltődése, vagyis az ottani akció nyomán történő esetleges „meghívásuk" előtt beolvassa őket.

JS-rendelkezések a <body> részben

Hogyha a parancsunkat nem szükséges függvényként végrehajtatnunk, vagy pl. csupán szöveget akarunk vele kiíratni, akkor a dokumentum **<body>** részében célszerű elhelyeznünk, pl.:

```
<html>
<head>
</head>

<body>
<script type="text/javascript">
document.write("This message is written by JavaScript");
</script>
</body>
</html>
```

Az oldalon tehát a **This message is written by JavaScript** felirat jelenik meg.

JS-rendelkezések a <head> és <body> részben
A dokumentumban tetszőleges számú scriptet heylezhetünk el, a fejrészben és a szövegtestben egyaránt, így pl. az alábbi oldalon mind a szövegtestbe, mind a fejrészbe írt JS-ek hibátlanul működnek, nem zavarják egymást:

```
<html>
<head>
<script type="text/javascript">
function message()
{
alert("This alert box was called with the onload event");
}
</script>
</head>

<body onload="message()">
<script type="text/javascript">
document.write("This message is written by JavaScript");
</script>
</body>
</html>
```

A szövegtest betöltésekor tehát figyelmeztető üzenet jelenik meg (párbeszédpanelben), a megjelenített tartalom pedig megfelel a szövegtestbe írt scriptnek.
Akkor is ugyanez történik, hogyha csak a fejrész ill. szövegtest tartalmazza mindkét scriptet, vagy azok elhelyezését felcseréljük. Tehát voltaképpen bárhol elhelyezhetjük őket, a lényeg, hogy a program képes leyen őket a megfelelő időben a megfelelő elemre vonatkoztatva végrehajtani, és hogy a kód áttekinthető legyen.

Külső JavaScript használata
Hogyha számos oldalon kívánjuk ugyanazokat a JavaScripteket futtatni, nem kell a rendelkezéseket minden egyes oldalba beleírnunk elgendő egy külső fájlként összeállított JS-re hivatkoznunk. E fájl **.js** kiterjesztéssel rendelkezzen!

A külső JS fájl csak JS-rednelkezéseket tartalmazhat, tageket és egyéb elemeket vagy szöveget nem. Ezért a **<script></script>** tageket se írjuk bele!

A külső JS-fájlra a **<script>** elem **src** (=source) attribútumával utalunk, pl.:

```
<html>
<head>
</head>

<body>

<script type="text/javascript" src="xxx.js">
</script>

<p>
The actual script is in an external script file called "xxx.js".
</p>

</body>
</html>
```

Mint látjuk, az oldal a **http://www.w3schools.com/js/xxx.js** JS-fájlra hivatkozik, aminek tartalma:

```
document.write("This text was written by an external script!")
```

A <script> taget, habár tartalmat nem írunk bele, továbbra is ugyanott kell elhelyezni, ahová belső

JS-ként beírnánk!

A JavaScript elhelyezése a weboldalban:

Belő JavaScriptek:

A HTML-oldalba írt JavaScripteket a böngésző az oldal betöltésekor azonnal végrehajtja; a függvényekbe írt kódot viszont csak a függvény meghívásakor.

JS-rendelkezések a <head> részben

A csak meghívásra indítandó függvény-parancssorokat a **<head>** részbe írjuk; hogy elkülönítsü a szövegtestbe írt, azonnal végrehajtandó JS-től, ill. hogy már meghívásuk előtt betöltse őket a böngésző..

JS-rendelkezések a <body> részben

Hogyha a parancsunkat nem szükséges függvényként (késleltetve) végrehajtatnunk, akkor a dokumentum **<body>** részében célszerű elhelyeznünk

A dokumentumban tetszőleges számú scriptet heylezhetünk el, a fejrészben és a szövegtestben egyaránt, azaz mind a szövegtestbe, mind a fejrészbe írt JS-ek hibátlanul működnek, nem zavarják egymást!

Külső JavaScriptek:

Adott esetben (ha annak tartalma megfelelő), a JS-et egy külső, **.js** kiterjesztésű fájlként is hozzáadhatjuk a weboldalhoz. A külső JS fájl csak JS-rendelkezéseket tartalmazhat, tageket és egyéb elemeket vagy szöveget nem. Ezért a **<script></script>** tageket se írjuk bele!

Hivatkozás külső JS-re:

<script type="text/javascript" src="*URL*.js"></script>

A <script> taget, habár tartalmat nem írunk bele, továbbra is ugyanott kell elhelyezni, ahová belső JS-ként beírnánk!

Scriptek:

Oldalbetöltés (=onload) esemény/attribútum:

<*elem* onload=”*függvény-név*()” />

Függvény:

function *függvény-név* (*paraméter1*,*paraméter2*)
{
végrehajtandó_kód;
}

Figyelmeztető-ablak:

alert(”*Üzenet*”);

IV. JavaScript – rendelkezések

A JS a böngésző által végrehajtandó rendelkezések (azaz parancsoknak és azok értékeinek) sorozata.

A JavaScript esetfüggő

A HTML-lel szemben a JS esetfüggő, ezért fokozottan ügyelnünk kell a nagy-és kisbetűk következetes használatára a JS-et alkotó kijelentések, (kijelölendő vagy meghívandó) változók, objektumok és függvények írásakor.

JavaScript rendelkezések (statements)

A JS-rendelkezések nem egyebek, mint a böngészőnek szóló utasítások, azaz a teendők meghatározásai.

Pl. az alábbi rendelkezés értelmében a böngészőnek meg kell jelenítenie a **Hello Dolly** feliratot a weboldalon:

```
document.write("Hello Dolly");
```

A rendelkezések végére általában pontosvesszőt írunk. Ez bevett és elismert gyakorlat a programozók között, az Interneten számos helyen talákozhatunk vele.

A JS szabványnak a rednelkezések pontosvesszővel való elválasztása csak kiegészítő eleme, mivel a böngészők e szabvány szerint a sortöréseket (entereket) tekintik az egyes rendelkezések végének. Így minden egyes rendelkezést külön sorba kellene írni.

A pontosvesszők használatával nemcsak könnyebben olbvashatóvá, elkülöníthetőbbé válnak az egyes rendelkezések, hanem egy sorba többet is írhatunk belőlük.

JavaScript kód

A JavaScript-kód vagy egyszerűbben JavaScript JS-rendelkezések sorozata. Ezeket a böngésző az oldalan elfoglalt helyük sorrendjében hajtja végre. Következő példánkban egy címsort és két bekezdést íratunk ki a weboldalra:

```
<html>
<body>
<script type="text/javascript">
document.write("<p>This is a paragraph.</p>");
</script>

<script type="text/javascript">
document.write("<h1>This is a heading</h1>");
document.write("<p>This is another paragraph.</p>");
</script>
</body>
</html>
```

Mint látjuk, itt a címsor a két bekezdés között található. Hogyha a pontosvesszőket kitöröljük, a kód továbbra is működik, de hogyha ezután a sortöréseket megszüntetjük és az egyes rendelkezéseket enterek helyett csak szóközök választják el, a böngésző nem képes kiírni a szöveget.

JavaScript – tömbök

A JS-rendelkezéseket ún. tömbökbe csoportosíthatjuk, melyeket kapcsos zárójelekkel jelölünk. A tömbbe tartozó rendelkezéseket a böngésző egyszerre hatja végre.

A következő példában ugyancsak címsort és bekezdéseket íratunk ki a weboldalra:

```
<html>
<body>

<script type="text/javascript">
{
```

```
document.write("<h1>This is a heading</h1>");
document.write("<p>This is a paragraph.</p>");
document.write("<p>This is another paragraph.</p>");
}
</script>

</body>
</html>
```

Példánk nem túl tanulságos, csupán a tömbbe zárás mondattanát mutatja be, mely a CSS-meghatározásaok csoportosításával analóg.
A rendelkezéseket leggyakrabban akkor csoportosítjuk, ha egy függvénybe (=function) vagy feltételbe (=condition) szeretnénk őket csoportosan beírni.

A függvényekről és feltételekről (mely utóbbi teljesülése esetén a rendelkezések adott csoportját a program végrehajtja) a későbbi fejezetekben lesz szó.

JavaScript – rendelkezések:

A JS-rendelkezések a böngészőnek szóló utasítások. Sortörések vagy pontosvesszők választhatják el őket a kódban.

A rendelkezéseket tömbökbe csoportosíthatjuk, amelyek elején ill végén kapcsos zárójelek állnak. A tömbbe zárt kódot (pl. egy függvény vagy feltételes rendelkezés értékeként) a böngésző a többitől elkülönítve, egységként hajtja végre.

A JS esetfüggő, ezért fokozottan ügyelnünk kell a nagy-és kisbetűk következetes használatára!

V. JavaScript – megjegyzések

A JS-megjegyzéseket a kód olvashatóbbá tételére használjuk.

JavaScript – megjegyzések

A JS-megjegyzések a JS-kód magyarázatára vagy áttekinthetőségének növelésére szolgálnak. Az egysoros megjegyzések kettős perjellel (//) kezdődnek

Az alábbi példában egysoros megjegyzésekkel magyarázzuk a kódot:

```
<html>
<body>

<script type="text/javascript">
// Write a heading
document.write("<h1>This is a heading</h1>");
// Write two paragraphs:
document.write("<p>This is a paragraph.</p>");
document.write("<p>This is another paragraph.</p>");
</script>

</body>
</html>
```

Mint látjuk, az egysoros megjegyzések használatának kulcsa, hogy a böngésző a JavaScript rendelkezéseit soronként értelmezi, így a megjegyéseink nem szorulnak külön lezáró jelre, hiszen a kód új sorban folytatódik.

Többsoros JavaScript-megjegyzések

A többsoros JS-megjegyzések /*-gal kezdődnek és */-re végződnek, a CSS-megjegyzésekhez hasonlóan.

Alábbi példánk rendelkezéseihez többsoros magyarázatot fűztünk:

```
<html>
<body>

<script type="text/javascript">
/*
The code below will write
one heading and two paragraphs
*/
document.write("<h1>This is a heading</h1>");
document.write("<p>This is a paragraph.</p>");
document.write("<p>This is another paragraph.</p>");
</script>

</body>
</html>
```

Végrehajtás-gátló megjegyzés-jelek

Az alábbi példában a megjegyzés-jelet egy rendelkezés végrehajtásának meggátlására, azaz kvázi a rendelzekés „kikapcsolására” használjuk. A megjegyzés-jelek ilyetén használata az oldalak hibáinak kijavításakor (=debugging) szokásos.

```
<html>
<body>

<script type="text/javascript">
//document.write("<h1>This is a heading</h1>");
```

```
document.write("<p>This is a paragraph.</p>");
document.write("<p>This is another paragraph.</p>");
</script>

</body>
</html>
```
Ebben az esetben tehát a címsor nem látszik, mert a megjelenítésére vonatkozó rendelkezést a böngésző egy egysoros megjegyzésnek tekinti.

Többsoros rendelkezések vagy rendelkezés-tömbök kikapcsolására szabatosan a többsoros megjegyzés-jelet alkalmazzuk:
```
<html>
<body>

<script type="text/javascript">
/*
document.write("<h1>This is a heading</h1>");
document.write("<p>This is a paragraph.</p>");*/
document.write("<p>This is another paragraph.</p>");
</script>

</body>
</html>
```
Itt csak az utolsó bekezdés kerül megjelenítésre.

Sorvégi megjegyzések
Következő példánk megjegyzéseit a rendelkezések után, a sorvégekre írtuk, így azok nem foglalnak el külön sorokat:
```
<html>
<body>

<script type="text/javascript">
document.write("Hello"); // Write "Hello"
document.write(" Dolly!"); // Write " Dolly!"
</script>

</body>
</html>
```
Az oladl tartalma itt is **Hello Dolly!**

JavaScript – megjegyzések:
A JS-megjegzéseket jegyzeteknek a kódba írásához vagy a kód egyes részeinek „kikapcsolásához”/érvénytelenítéséhez használjuk a szerkesztés vagy javítás során.

Egysoros megjegyzés:
//Megjegyzés
Többsoros megjegyzés:
*/*Megjegyzés*
Megjegyzés/*
Sorvégi megjegyzés (nem foglal el külön sort):
kód; //Megjegyzés

VI. JavaScript – változók

A változók (=variables) a JS információ-tároló részei.

Algebrai változók

Az iskolai algebrából emlékezhetünk a következőhöz hasonló összefüggésekre:

```
x=5, y=6, z=x+y
```

Ezekben a betűkhöz (pl. az **x**) bizonyos értéket (pl. **5**) társítva a művelet végeredménye (**z = 11**) kiszámíthatóvá válik.

Az említett betűket változóknak (=variables) nevezik, melyek bizonyos értékekre (pl. **x = 5**) illetve összefüggésekre (pl. **z=x+y**), azaz végső soron információkra utalhatnak.

JavaScript – változók

Az algebrai változókhoz hasonlóan a JS változói is értékeket ill. összefüggéseket takarnak.

Tetszőleges elnevezésekkel illethetjük őket, mint amilyen az **x**, illetve jellegzetesebbekkel is, mint pl. a **carname**.

A JS-változók elnevezésekor két szabályt kell betartanunk:

1. A JS-változó – nevek, mint minden JS-elem, esetfüggők, azaz pl. az **y** és az **Y** két külön változót jelöl.
2. A változó-neveknek betűvel vagy aláhúzás-karakterrel (_) kell kezdődniük.

Egy gyors példa

A változók értéke a parancssor végrehajtása közben változhat. Az aktuális érték behívása ill. megváltoztatása egyaránt az érték-névre való hivatkozással történik, mint azt példánk mutatja:

```
<html>
<body>

<script type="text/javascript">
var firstname;
firstname="Hege";
document.write(firstname);
document.write("<br />");
firstname="Tove";
document.write(firstname);
</script>

<p>The script above declares a variable,
assigns a value to it, displays the value, changes the value,
and displays the value again.</p>

</body>
</html>
```

A példánkban szereplő parancssor első sorában definiáljuk a **firstname** nevű változót, majd a második sorban hozzárendeljük a **Hege** értéket.
A harmadik sor kiíratja a böngészővel a **firstname** változó éppen aktuális értékét (**Hege**).
A negyedik sor egy sortörést jelent a megjelenő szövegben.
Az ötödik sorban a **firstname** változóhoz új értéket rendelünk (**Tove)**, majd a hatodik sorban ismét kiíratjuk a **firstname** változó értékét, ami ekkor már **Tove.**

A parancssor futtatásának eredménye tehát a
Hege
Tove

felirat lesz.

JavaScript – változók létrehozása
A JS – változók létrehozása (=creation) ún. meghatározással (=declaration) történik, amit a **var** kulcsszóval végzünk, pl.:

```
var x;
var carname;
```

Az így létrehozott két válzotó üres (=empty), azaz értékkel nem rendelkezik.

A változókhoz, mint fentebb láttuk, a ***változónév=szám*;** ill. a ***változónév="szöveg"*;** rendelkezésekkel redndelhetünk szám- vagy szöveges értéket. Ezt (azaz az értékek megadását) a definiálással össze is vonhatjuk, pl.:

```
var x=5;
var carname="Volvo";
```

E rendelkezések végrehajtásával az **x** változó értéke **5**, a **carname**-é pedig **Volvo** lesz.

Amint láttuk, a szöveges változó-értékeket idézőjelbe kell tenni.

Természesen a számokat is tehetjük idézőjelbe, de akkor azt a böngésző szövegnek fogja tekinteni és akként kezeli. Pl. a böngésző a

```
document.write(<br />);
```

parancsot nem hajtja végre, mert a **
** taget nem tudja számként értelmezni. Hogyha ehelyett a

```
document.write("<br />");
```

rendelkezéssel élünk, akkor sortörést kapunk.
Ugyanígy, a

```
document.write("x");
```

rendelkezéssel a böngésző nem az **x** függvény értékét (**5**) írja ki, hanem az **x** betűt. Ugyanakkor az

```
document.write(x);
```

rendelkezést már nem betűként, hanem számként kezeli, és az **x** változónak (azaz számnak) megfelelő, **5**-ös értéket írja ki.

JavaScript – változók létrehozása az értékek megadásával
Ahogy fentebb láttuk, a JS-változó – értékek megadásakor mindig le kell írnunk a változó nevét is. Ezért úgy vehetjük, hogy az érték-megadással egyúttal az érték-nevet is definiáljuk.

Azaz a JS-ben a meg nem határozott változók értékének megadásával automatikusan definiáljuk a változókat is. Ennek értelmelmében a

```
x=5;
carname="Volvo";
```

parancssor jelentése megegyezik az előbb ismertetett, közös meghatározási módszer eredményével:

```
var x=5;
var carname="Volvo";
```

illetve annak hosszabb megfelelőjével:

```
var x;
x=5;
var carname;
carname="Volvo";
```

A JavasScript – változók újra-meghatározása
Hogyha – pl. mint az alábbi parancssorban kátható – egy JS-változót újra-definiálunk, annak értéke továbbra is megmarad, pl.: a következő parancssor értelmében

```
<html>
<body>
<script type=”text/javascript”>
var x=5;
document.write(x);
document.write(”<br />”);
var x;
document.write(x);
</script>
</body>
</html>
```

A böngésző egymás alá két **5**-öst ír ki, jelezve, hogy az **x** változó újradefiniálása az értékét önmagában nem változtatta meg ill. nem törölte.

Következő példánkban azonban kétszer is megváltoztatjuk az **x** értékét, s eközben mindhárom definiálási eljárást alkalmazzuk:

```
<html>
<body>
<script type=”text/javascript”>
var color;
color=”red”;
document.write(color);
document.write(”<br />”);
var color=”green”;
document.write(color);
document.write(”<br />”);
color=”blue”;
document.write(color);
</script>
</body>
</html>
```

Ebben az esetben a **red**, **green** és **blue** szavakat látjuk egymás alatt.

JavaScript – aritmetika
Az algebrához (betűszámtanhoz) hasonlóan a JS-változókkal is végezhetünk aritmetikai (számtani) műveleteket, pl.:

```
y=x-5;
z=y+5;
```

Az ilyenkor végeztethető műveletekről (=operators) a következő fejezetben lesz szó.

JavaScript – változók:
A változók a JS információ-tároló részei. Az algebrai változókhoz hasonlóan értékeket ill. összefüggéseket takarnak.

Tetszőleges elnevezésekkel illethetjük őket, de két szabályt be kell tartanunk:

1. A JS-változó – nevek esetfüggők, azaz pl. **y** és **Y** két különbözőt jelöl.
2. A változó-neveknek betűvel vagy aláhúzás-karakterrel (_) kell kezdődniük.

Változók definiálása:

1. **Külön sorokban:** var x;
   var x=1;
2. **Egy sorban:** var x=1;

3. **Az érték megadásával:** x=1;

Érték-típusok:

Szám, pl.: 123
Szöveg, pl.: ”*Szöveg*”
Logikai érték: *true/false/1/0*
Művelet, pl.: *a+b*

Üres (empty): ””/null
Nem szám (not a number): NaN

Változók újradefiniálása, hierarchiája:

1. A változók nevének (pl. var „telefon”;) újradefiniálása az értékre nincs hatással.
2. Az érték újbóli megadása felülírja a korábbit (az oldalban előbb szereplőt).
3. A függvényeken ill. tömbökön belüli (helyi), és az egész dokumentumra vonatkozó (külső) változók típusa ért értéke között nincs összefüggés.

VII. JavaScript – műveletek

Alapismeretek

Az egyenlőségjelnek (=) megfelelő hozzárendelési műveletet a JS-válltozó(-neve)k és értékük egymáshoz rendelésére alkalmazzuk.

A számtani összeadásjelnek (+) megfelelő művelet pedig a változók értékeinek összeadását jelenti.

Pl. az alábbi parancssorban lévő rendelkezések –

```
y=5;
z=2;
x=y+z;
```

– végrehajtásának eredménye a képernyőn: **7**.

JavaScript – számtani műveletek

A számtani (=arithmetic) műveletekel a változókat és/vagy értéküket kezelhetjük.

A következő táblázatban az **y=5** esetre vonatkozó példákon keresztül mutatjuk be a JS-ben használt számtani műveleteket:

Művelet (=operator)	Leírás	Példa	Eredmény
+	Összeadás (=addition)	x=y+2	x=7
-	Kivonás (=subtraction)	x=y-2	x=3
*	Szorzás (=multiplication)	x=y*2	x=10
/	Osztás (=division)	x=y/2	x=2.5
%	Modulus (osztási maradék)	x=y%2	x=1
++	Növekmény (=increment)	x=++y	x=6
--	Csökkenés (=decrement)	x=--y	x=4

JavaScript – hozzárendelő-jellemzők

A hozzárendelő-jellemzőkkel értéket rendelhetünk a JS-változókhoz.

Az alábbi táblázatban összefoglaljuk őket, példákon is bemutatva használatukat (**x=10** és **y=5** esetén):

Művelet	Példa	Hagyományos alak	Érték
=	x=y		x=5
+=	x+=y	x=x+y	x=15
-=	x-=y	x=x-y	x=5
*=	x*=y	x=x*y	x=50
/=	x/=y	x=x/y	x=2
%=	x%=y	x=x%y	x=0

Többtagú vagy szöveges értékek összeadása (a + művelettel)

A + műveletet több számból vagy szóból álló változó-értékek összeadására is használhatjuk; pl. a következő parancssor végrehajtása után a **txt3** változó értéke **What a verínice day** lesz:

```
txt1="What a very";
txt2="nice day";
txt3=txt1+txt2;
```

Tehát a fenti rendelkezések végrehajtása után a **txt3** tartalma (azaz értéke): **What a verynice day**.

Hogyha a **very** ill. **nice** szavak között szóközre is szükség van, akkor azt vagy az egyik változó érték-sorozatába kell beírnunk –

```
txt1="What a very ";
```

```
txt2="nice day";
txt3=txt1+txt2;
```
vagy
```
txt1="What a very";
txt2=" nice day";
txt3=txt1+txt2;
```
– vagy a **txt3** értékét megadó kifejezésbe kell közvetlen értékként vagy egy újabb, külső változó értékeként belefoglalnunk, azaz vagy
```
txt1="What a very";
txt2="nice day";
txt3=txt1+” ”+txt2;
```
vagy
```
txt1="What a very";
txt1b=” ”;
txt2="nice day";
txt3=txt1+txt1b+txt2;
```
rendelkezésekkel élnünk. Ezek végrehajtása után tehát a **txt3** tartalma: **What a very nice day**.

Megjegyezzük, hogy a JS-be írt szövegek többszörös szóközei is eggyé olvadnak!

Érték-sorok (többszörös számértékek vagy szavak) és számok összeadása
Esetén az eredmény minig érték-sor lesz.

Ezt a szabályt illusztrálja alábbi példánk is:
```
<html>
<body>

<script type="text/javascript">
x=5+5;
document.write(x);
document.write("<br />");
x="5"+"5";
document.write(x);
document.write("<br />");
x=5+"5";
document.write(x);
document.write("<br />");
x="5"+5;
document.write(x);
document.write("<br />");
</script>

<p>The rule is: If you add a number and a string, the result will be a
string.</p>

</body>
</html>
```
ahol az első művelet eredménye **10**, az összes többié viszont **55**, ami tehát nem számnak, hanem számsorozatnak/szövegnek számít, tehát pl. ha a parancssorhoz hozzáírjuk az
```
y=x+6;
document.write(y);
```
meghatározásokat is, annak eredménye 61 helyett **556** lesz; míg ha rögtön az első **x**-meghatározás után írjuk, akkor **y** értéke **16** lesz.

JavaScript – műveletek:

Számtani műveletek jelei:

- **Összeadás:** +
- **Kivonás:** -
- **Szorzás:** *
- **Osztás:** /
- **Modulus (osztási maradék):** %
- **Növekmény:** ++
- **Csökkenés:** --

Hozzárendelések jelei:

- **Egyenlőség:** x=y
- **Növelés:** x+=y [x=x+y]
- **Csökkentés:** x-=y [x=x-y]
- **Többszörözés:** x*=y [x=x*y]
- **Felosztás:** x/=y [x=x/y]
- **Maradék-képzés:** x%=y [x=x%y]

Szöveg-értékű változók összeadása:

var x=”*Szöveg1*”;
var y=”*Szöveg2*”;
var z=x+” ”+y;
az eredmény: *Szöveg1 Szöveg2*.

Számot szöveggel összeadva az eredmény szöveg (string); pl. x=5+5=10, de y=5+”5”=55!

VIII. JavaScript – összehasonlító és logikai műveletek

Az összehasonlító és logikai műveleteket igaz/hamis megkülönböztetésekre használjuk a JS-ben.

Összehasonlító műveletek

Az összehasonlító műveleteket (=comparison operators) logikai rendelkezésekben használjuk, változók vagy azok értékei közti azonosság vagy eltérés felismerésére.

Az alábbi táblázatból (ahol **x=5**) megismerhetjük a JS összehasonlító műveleteit:

Művelet	Leírás	Példa	Műveleti érték
==	(x) **egyenlő** (8-cal)	x==8	HAMIS
===	(x) **azonosan** (azaz mind értékében, mind szöveg/szám jellegében) **egyenlő** (5-tel ill. "5"-tel)	x===5 x==="5"	IGAZ HAMIS
!=	(x) **nem egyenlő** (8-cal)	x!=8	IGAZ
>	(x) **nagyobb, mint** (8)	x>8	HAMIS
<	(x) **kisebb, mint** (8)	x<8	IGAZ
>=	(x) **nagyobb vagy egyenlő, mint** (8)	x>=8	HAMIS
<=	(x) **kisebb vagy egyenlő, mint** (8)	x<=8	IGAZ

Összehasonlító műveletek használata

Az összehasonlító műveleteket feltételes rendelkezések (=conditional statements) alapértékeinek összehasonlítására, azaz végső soron a rendelkezés érvényességének eldöntésére és az annak megfelelő akció elindítására használhatók, pl.:

```
if (age<18) document.write("Too young");
```

Itt, hogyha az **age** változó értéke 18-nál kisebb, a böngésző a **Too young** feliratot írja ki az oldalra.

A feltételes rendelkezésekkel (=conditional statements) a következő fejezetben foglalkozunk.

Logikai műveletek

A logikai műveletek (=logical operators) segítségével változók vagy azok értékei közti logikai (ÉS/NEM/VAGY) kapcsolatokat írhatunk le.

Táblázatunk **x=6** és **y=3** esetére felírt példákon mutatja be a három logikai művelet JS jelölését és alkalmazását:

Művelet	Leírás	Példa	Műveleti érték
&&	ÉS	(x<10 && y>1)	IGAZ
||	VAGY	(x==5 || y==5)	HAMIS
!	NEM	!(x==y)	IGAZ

Feltételes hozzárendelés

A JS egy úgynevezett feltételes hozzárendelés (=conditional operator) is tartalmaz. A feltételes hozzárendelés két lehetséges érték közül a fetétel teljesülésének vagy meghiúsulásának megfelelően az elsőt vagy a másodikat rendeli a változóhoz.

A feltételes művelet mondattana:
***változó*=(*feltétel*)?*érték1*:*érték2*;**

Egy példa feltételes műveletre:

```
greeting=(visitor=="PRES")?"Dear President ":"Dear ";
```

Mint látjuk, ez a **greeting** változóra vonatkozik. A feltétel szerint, hogyha a **visitor** változó értéke egyenlő **PRES**-sel, akkor az IGAZ műveleti értéknek megfelelő **Dear President** lesz a **greeting** értéke (melyet pl. egy **document.write(greeting);** paranccsal kiíratunk valahová); ellenkező esetben pedig csak **Dear**.

JavaScript – összehasonlító és logikai műveletek:

Összehasonlító műveletek:

Egyenlő:	5= =”5”
Azonosan (értékében és típusában) egyenlő:	”5”= = =”5”
Nem egyenlő:	5!=6
Nagyobb, mint:	6>5
Kisebb, mint:	5<6
Nagyobb vagy egyenlő, mint:	6>=5
Kisebb vagy egyenlő, mint:	5<=6

Logikai műveletek:

ÉS:	&&
VAGY:	||
NEM:	!

Feltételes hozzárendelés:

A feltételes hozzárendelés két lehetséges érték közül a fetétel teljesülésének vagy meghiúsulásának megfelelően az elsőt vagy a másodikat rendeli a változóhoz.

Mondattan:

***változó*=(*feltétel*)?*érték1*:*érték2*;**

Példa:

greeting=(visitor=="PRES")?"Dear President ":"Dear ";

[Ha visitor==”PRES”, akkor greeting=”Dear President ”, mádként geeting=”Dear ”.]

IX. JavaScript – feltételes rendelkezések

A feltételes rendelkezésekkel (=conditional statements) megadhatjuk, hogy az egyes feltételek teljesülését milyen akció kísérje.

Feltételes rendelkezések

A weboldalak írásakor igen gyakran felmerülő igény, hogy a felhasználó döntéseinek megfelelő dolog történjen meg (pl. megfelelő szöveg vagy dokumentum-részlet jelenjen meg). Ezt a kódba írt feltételes rendelkezésekkel érhetjük el.

A JavaScriptben négyféle feltételes rendelkezés szerepel:

- **if** a böngésző akkor hajtja végre a kérdéses kód-részletet, ha a feltétel IGAZ;
- **if…else** ha a feltétel IGAZ/HAMIS voltánakmegfelelően a böngésző egyik vagy másik kód-részletet hajtja végre.
- **if…else if…else** feltétetelt használunk, hogyha még több lehetséges kód-variáns közül kell a megfelelőt kijelölni.
- **switch** szintén akkor használjuk, hogyha számos lehetséges kód-variáns közül kell egyet kijelölni.

Az if feltételes rendelkezés

Az **if** feltételes rendelkezés tartalma csak akkor kerül végrehajtásra, ha a feltétel teljesül, vagyis a feltételül szabott művelet értéke IGAZ.

A feltételes rendelkezés mondattana:
if (*feltétel-művelet*)
{
végrehajtandó kód
}

Az **if** nevet mindig kisbetűkkel írjuk, másképp a rendelkezés nem működik!

Következő példánkban a böngésző belső órájának aktuális idő-értékéhez képest, reggel 10 óra előtt a **Good morning** felirat jelenik meg (egyébként nem jelenik meg semmi):

```
<html>
<body>

<script type="text/javascript">
var d = new Date();
var time = d.getHours();

if (time < 10)
  {
  document.write("<b>Good morning</b>");
  }
</script>

<p>This example demonstrates the If statement.</p>
<p>If the time on your browser is less than 10, you will get a "Good
morning" greeting.</p>

</body>
</html>
```

Mint látjuk, először meghatározzuk a **d** változót, melynek értékéül a böngésző belső órájának (az oldal betöltésekor) aktuális állását vesszük. Ezután a **time** változót definiáljuk, ami nem más, mint az előbbi időpontot jelölő számsorból az órák számértéke.

Majd következik a feltételes rendelkezés: hogyha a **time**, azaz az előbbi, egész órákra vonatkozó adat 10-nél kisebb, akkor a böngésző kiírja a **Good morning** szöveget, félkövér szedéssel.
Hogyha az idő tíz óránál több, nincs felirat. Hogyha már későre jár, átírhatjuk a feltételt pl. 22-re, ekkor este 10 óra előtt még jó reggelt fog kívánni a böngésző.

Az if else feltételes rendelkezés
Az if else feltételes rendelkezést akkor használjuk, hogyha a feltétel betöltetlensége (vagyis a benne foglalt művelet NEM értéke) esetén is szeretnénk valamilyen rendelkezés teljesülését (következésképpen egy, az előzőtől eltérőét, hiszen egyébként nem kéne feltételhez kötni a megvalósulását).

Az if else feltételes rendelkezés mondattana:
if (*feltétel-művelet*)
{
végrehajtandó kód no_1
}
else
{
végrehajtandó kód no_2
}

Hogyha tehát a **feltétel-művelet** teljesül, a **no_1**, ellenkező esetben a **no_2** kód lép érvénybe.

Előző példánkat tehát most már egy alternatív (10 óra utánra vonatkozó) üdvözléssel is kiegészíthetjük:

```
<html>
<body>

<script type="text/javascript">
var d = new Date();
var time = d.getHours();

if (time < 10)
{
document.write("<b>Good morning</b>");
}
else
{
document.write("<b>Good day</b>");
}
</script>

<p>
This example demonstrates the If...Else statement.
</p>

<p>
If the time on your browser is less than 10,
you will get a "Good morning" greeting.
Otherwise you will get a "Good day" greeting.
</p>

</body>
</html>
```

10 óránál korábban a böngésző tehát **Good morning**-ot ír ki, ellenkező esetben pedig **Good day**-t.

Az if else if else feltételes rendelkezés

Természetesen a második felirat megjelenését is köthetjük valamiféle feltételhez, melynek betöltetlensége esetére újabb, immár harmadik alternatívát adhatunk meg; s így a kezdeti **if** feltétel után korlátlan számú **else**, azaz további feltételt és azokhoz tartozó kód-variánsokat fűzhetünk.

Az if else if else feltételes renedlkezése mondattana tehát (három lehetőséggel):
if (*feltétel-művelet no_1*)
{
végrehajtandó kód no_1
}
else (*feltétel-művelet no_2*)
{
végrehajtandó kód no_2
}
else
{
végrehajtandó kód no_3
}

Hogyha a **feltétel-művelet no_1** teljesül, akkor a **végrehajtandó kód no_1** lép érvénybe, ellenkező esetben pedig, a **feltétel-művelet no_2** teljesülése esetén **végrehajtandó kód no_2**-nek megfelelő második alternatíva; ha pedig a második feltétel-művelet értéke is HAMIS, akkor a **no_3** kód kerül végrehajtásra.

A következőképpen bővíthetjük ki tehát előző példánkat:

```
<html>
<body>

<script type="text/javascript">
var d = new Date();
var time = d.getHours();
if (time<10)
{
document.write("<b>Good morning</b>");
}
else if (time>=10 && time<16)
{
document.write("<b>Good day</b>");
}
else
{
document.write("<b>Hello World!</b>");
}
</script>

<p>
This example demonstrates the if..else if...else statement.
</p>

</body>
</html>
```

Itt 10 óra előtt **Good morning**, 10 és 16 óra közt **Good day**, azután pedig **Hello World!** feliratot látunk. Mint látjuk, a második feltétel egy ÉS logikai művelet.

Egy további példa: véletlen link
Következő példánkban egy újabb gépi függvényt, a **Math.random()** véletlenfüggvényt használjuk fel,

mely 0 és 1 közötti számot generál, tehát annak esélye, hogy értéke 0,5-nél nagyobb, 50%.
Ennek megfelelően az alábbi példában szereplő JS hol a **Learn Web Development!**, hol pedig a **Visit Refsnes Data!** linket írja ki, egyenlő esetszámban:

```
<html>
<body>

<script type="text/javascript">
var r=Math.random();
if (r>0.5)
{
document.write("<a href='http://www.w3schools.com'>Learn Web
Development!</a>");
}
else
{
document.write("<a href='http://www.refsnesdata.no'>Visit Refsnes
Data!</a>");
}
</script>

</body>
</html>
```

JavaScript – feltételes rendelkezések:

Az if feltételes rendelkezés

Az if feltételes rendelkezés tartalma csak akkor kerül végrehajtásra, ha a feltétel teljesül, vagyis logikai értéke IGAZ.

Modattana:

if (*feltétel-művelet*)
{
végrehajtandó kód
}

Példa:

```
var d = new Date();
var time = d.getHours();

if (time < 10)
{
document.write("<b>Good morning</b>");
}
```

Az if else feltételes rendelkezés

Az if else feltételes rendelkezést akkor használjuk, hogyha a feltétel NEM értéke esetére is rendelkezni akarunk.

Mondattana:

if (*feltétel-művelet*)
{
végrehajtandó kód no_1
}
else
{
végrehajtandó kód no_2

```
}
```

Példa:

```
var d = new Date();
var time = d.getHours();

if (time < 10)
{
document.write("<b>Good morning</b>");
}
else
{
document.write("<b>Good day</b>");
}
</script>
```

Az if else if else feltételes rendelkezés

Az if-re következő alternatíva végrehajtását is feltételhez köthetjük, így korlátlan számú **else**, azaz további feltételt és azokhoz tartozó kód-variánst fűzhetünk utána.

Mondattana (három lehetőséggel):

```
if (feltétel-művelet no_1)
{
végrehajtandó kód no_1
}
else (feltétel-művelet no_2)
{
végrehajtandó kód no_2
}
else
{
végrehajtandó kód no_3
}
```

Példa:

```
var d = new Date();
var time = d.getHours();

if (time<10)
{
document.write("<b>Good morning</b>");
}
else if (time>=10 && time<16)
{
document.write("<b>Good day</b>");
}
else
{
document.write("<b>Hello World!</b>");
}
```

Véletlen link készítése:

```
<html>
<body>
```

```
<script type="text/javascript">
var r=Math.random();
if (r>0.5)
{
document.write("<a href='URL1'>Link-szöveg1</a>");
}
else
{
document.write("<a href='URL2'>Link-szöveg2</a>");
}
</script>

</body>
</html>
```

X. JavaScript – a switch feltételes rendelkezés

Mint láttuk, a JS feltételes rendelkezései eltérő feltételek mellett eltérő kód végrehaktásához vezetnek.

A switch feltételes rendelkezés

A **switch** feltétel számos (a többszörös **if else** feltételekkel szemben nem hierarchikus) rendelkezési lehetőség közül választja ki az előírt értéknek leginkább megfelelőt.

A switch feltételes rendelkezés mondattana (három lehetőséggel):

switch(*érték*)
{
case *érték_1*:
végrehajtandó kód no_1
break;
case *érték_2*:
végrehajtandó kód no_2
break;
default:
végrehajtandó kód no_3
}

A **switch** rendelkezés feltételét (mely leggyakrabban egy változó) az **érték** szóval jelöltük. A böngésző ezen érték beolvasása után végignézi az összes esethez tartozó értéket (itt **érték_1** és **érték_2**). Hogyha ezek közül valamelyik megegyezik a fő-értékkel, akkor az ahhoz tartozó rendelkezéseket végrehajtja. A **break;** szó beírásával meggátolhatjuk, hogy a kód végrehajtásakor a program túlfusson a következő eset szövegére. Enélkül a program az összes lehetőségnek megfelelő parancssort végrehajtja!

A **break;** parancs tehát a parancssor futását leállítja, azaz amikor

Hogyha a feltételnek egy eset-érték sem felel meg, akkor a program nem ír ki semmit. Ezért a **switch** lehetséges értékeinek felsorolása végére beírhatunk egy **default**, azaz alapértelmezett kódot. Hogyha a program egyezés-találat nélkül eljut idáig, akkor bármilyen feltétel esetén végrehajtja e rendelkezést.

Következő példánkban a böngésző eltérő üdvözletet ír ki, attól függően, hogy épp a hét mely napján járunk:

```
<html>
<body>
<script type="text/javascript">
var d = new Date();
theDay=d.getDay();
switch (theDay)
{
case 5:
  document.write("<b>Finally Friday</b>");
  break;
case 6:
  document.write("<b>Super Saturday</b>");
  break;
case 0:
  document.write("<b>Sleepy Sunday</b>");
  break;
default:
  document.write("<b>I'm really looking forward to this weekend!</b>");
}
</script>
```

```
<p>This JavaScript will generate a different greeting based on what day
it is. Note that Sunday=0, Monday=1, Tuesday=2, etc.</p>

</body>
</html>
```

Mint látjuk, a **d** változó értéke itt is a böngésző-óra (letöltéskor) aktuális értéke, melyből a **theDay** változó a napra vonatkozó, 0 és 6 közötti számot választja ki.

E változót a switch rendelkezés feltételéül választva, a pénteknek megfelelő **5**-ös szám esetén félkövér **Finally Friday** felirat jelenik meg az oldalon, **6**=szombat esetén **Super Saturday**, **0**=vasárnapnál pedig **Sleepy Sunday**. Hogyha a nap-szám 1 és 4 közötti (hétfő-csürtörtök), akkor az alapértelmezett **I'm really looking forward to this weekend!** szöveg jelenik meg – hogyha pedig az utolsó (**default**) lehetőséget kitöröljük, akkor semmi.

JavaScript – a switch feltételes rendelkezés:

A **switch** feltétel számos (a többszörös **if else** feltételekkel szemben nem hierarchikus) rendelkezési lehetőség közül választja ki az előírt értéknek leginkább megfelelőt.

Mondattana (három lehetőséggel):

switch(*érték, pl. változó*)
{
case *érték_1 pl. egy szám*:
    végrehajtandó kód no_1
    break;
case *érték_2 pl. másik szám*:
    végrehajtandó kód no_2
    break;
default:
    végrehajtandó kód, ha az előző két case nem teljesült
}

XI. JavaScript – felugró ablakok

A JS-ben háromféle felugró ablakot alkalmazunk: figyelmeztető, megerősítő és beviteli ablakokat.

Figyelmeztető ablak

A figyelmeztető felugró ablakok segítségével (=alert popup boxes) olyan adatokat közlünk a felhasználóval, melyeket feltétlenül meg kell ismernie.

A figyelmeztető ablak megjelenésekor a felhasználónak meg kell nyomnia az OK gombot a továbblépéshz (az oldal további használatához).

A figyelmeztető ablak mondattana:
alert(”*üzenet-szöveg*”);

Következő példánban egy gombot találunk, ami kattintásra egy figyelmeztető ablakot nyit meg:

```
<html>
<head>
<script type="text/javascript">
function show_alert()
{
alert("Hello! I am an alert box!");
}
</script>
</head>
<body>

<input type="button" onclick="show_alert()" value="Show alert box" />

</body>
</html>
```

A weboldal itt egyetlen beviteli mezőből áll, amit egy **Show alert box** feliratú gomb. A gombra kattintva a böngésző végrehajtja a **show_alert** nevű függvényt.
Ennek tartalma egy figyelmeztető-ablak megjelenítése. Azaz az előbbi gombra kattintva, a függvény végrehajtásának eredményeképpen megjelenik egy párbeszédpanel, **OK** és **X** (=escape=kilépés) gombbal, valamint sárga hátterű, felkiáltójeles háromszög mellett az üzenettel: **Hello! I am an alert box!**.

Mindaddig, míg az **OK** vagy **X** gombok közül valamelyiket meg nem nyomjuk, a böngészőt nem használhatjuk (az ugyanis nem reagál az egérkattintásra, hanem figyelmeztetőleg villogtatja a párbeszéd-panelt).

Megerősítő ablak

A megerősítő ablakok (=confirm boxes) parancsok vagy adatok megerősítésére ill. elfogadására szolgálnak. Megjelenésük után a felhasználónak **OK** vagy **Cancel** gombot kell nyomnia a továbblépéshez.

Az **OK** megnyomása esetén a doboz műveleti értéke IGAZ, **Cancel** esetén pedig HAMIS.

A megerősítő ablak mondattana:
confirm(”*üzenet-szöveg*”);

Következő példánban egy gombot találunk, ami kattintásra egy megerősítő ablakot nyit meg:

```
<html>
<head>
<script type="text/javascript">
function show_confirm()
{
```

```
var r=confirm("Press a button!");
if (r==true)
  {
  alert("You pressed OK!");
  }
else
  {
  alert("You pressed Cancel!");
  }
}
</script>
</head>
<body>

<input type="button" onclick="show_confirm()" value="Show a confirm
box" />

</body>
</html>
```

Ebben az esetben a **Show a confirm box** gomb megnyomásakor a **show_confirm** függvény parancssorát hajtatjuk végre.
Ennek első rendelkezése definiálja az **r** változót, aminek értéke megegyezik a felugró megerősítő ablak logikai értékével. A felugró ablak három gombbal (**OK**, **Cancel** és **X**), felirattal **Press a button!**, valamint egy kis kérdőjeles szövegbuborék-képecskével rendelkezik. Amíg nem nyomjuk meg valamelyik gombot, addig a böngésző a figyelmeztető ablaknál tapasztaltakhoz hasonlóan, nem használható. Az **OK** megnyomására egy **You pressed OK!** feliratú figyelmeztető ablak nyílik meg, melyet nyugtáznunk kell a böngésző-beli munkafolyamat folytatásához. A **Canel** megnyomásakor megjelenő figyelmeztető ablak felirata: **You pressed Cancel!**.
Mint látjuk, a két figyelmeztető ablak közti választást egy **if else** feltételes rendelkezéssel oldjuk meg, ami a megerősítő ablak IGAZ értéke esetén (vagyis ha **r==true**) az első, minden más esetben pedig a második figyelmeztetést jeleníti meg, így pl. hogyha az **X** gombot nyomjuk meg, a felirat akkor is az **else** esetnek megfelelő **You pressed Cancel!**.

Beviteli ablak
A beviteli ablakokat általában akkor használjuk, ha a felhasználótól egy értéket (pl. jelszavat) akarunk kérni az oldalra való belépéshez.

A beviteli ablakon **OK**, **Cancel** vagy **X** gomb megnyomásával juthatunk csak tovább a megadott értéknek megfelelő oldalra.

Az **OK** gomb megnyomására a beviteli ablak felveszi a beírt értéket, a **Cancel** vagy **X** gomb esetében az értéke **null** (azaz tulajdonképpen üres) lesz.

A beviteli ablak mondattana:
prompt(”*üzenet-szöveg*”,”*alapértelmezett érték*”);

Következő példánban egy gombot találunk, ami kattintásra egy beviteli ablakot nyit meg:

```
<html>
<head>
<script type="text/javascript">
function show_prompt()
{
var name=prompt("Please enter your name","Harry Potter");
if (name!=null && name!="")
  {
```

```
  document.write("Hello " + name + "! How are you today?");
  }
}
</script>
</head>
<body>

<input type="button" onclick="show_prompt()" value="Show prompt box" />

</body>
</html>
```

Mint látjuk, a weboldal betöltésekor csak a **Show prompt box** feliratú gombot tartalmazza. Erre kattintva a gép végrehajtja a **show_prompt** függvényt. Ennek parancssora először definiálja a **name** változót, melynek értékéül a beviteli ablak értékét választja. A beviteli ablak üzenet-szövege **Please enter your name**, alapértelmezett válasza pedig **Harry Potter** lesz.
A gomb megnyomására tehát felnyílik az ablak, három gombbal, és a szövegbeviteli mezőben az alapértelmezett (**Harry Potter**) értékkel.
Hogyha az **OK** gombot megnyomjuk, az ablak (és így a **name** változó is) felveszi a **Harry Potter** értéket; amit a parancssor **if** rendelkezésének feltételévelhasonlítunk össze. Hogyha a **name** értéke nem **null** ÉS nem **üres** (azaz **””**), akkor a böngésző kiírja a **Hello *name*! How are you today?** szöveget, ahol a ***name*** értéke jelen esetben **Harry Potter**, és a gomb eltűnik.

Hogyha a **Cancel** vagy **X** gombot nyomjuk meg, az ablak értéke **null**, így az **if** feltételes rendelkezés feltétele HAMISnak bizonyulván, nem kapunk feliratot.

Hogyha kitöröljük a szövegmezőből az alapértelmezett értéket és nem írunk be semmit, az ablak értéke **üres** (**””**), minek folytán megint nem kapunk feliratot.

Amíg a beviteli ablakon nem nyomunk meg egy gombot sem, addig itt sem folytathatjuk a böngészőbeli munkamenetet.

Ablak-üzenetek sortörése
Alábbi példánkban egy figyelmeztető ablak példáján bemutatjuk, hogyan lehet az ablakok üzenet-szövegébe sortörést írni:

```
<html>
<head>
<script type="text/javascript">
function disp_alert()
{
alert("Hello again! This is how we" + '\n' + "add line breaks to an
alert box!");
}
</script>
</head>
<body>

<input type="button" onclick="disp_alert()" value="Display alert box"
/>

</body>
</html>
```

Mint látjuk, a **Display alert box** feliratú gomb megnyomására a **disp_alert** függvény megjeleníti az **alert** párbeszédpanelt (figyelmeztető ablakot), melyben a
Hello again! This is how we
add line breaks to an alert box!

felirat jelenik meg.
A figyelmeztető ablak üzenet-szöveg-értékébe, a kettős idézőjelekbe írt többi szövegrészlet közé a + **'\n'** + kifejezést írva, a kérdéses helyen sortörés lesz.

Megerősítő és beviteli ablak-üzenetek sortörésében is ugyanígy járunk el, pl.:
confirm("Első sor" + '\n' + "Második sor"); ill.
prompt("Első sor" + '\n' + "Második sor","Alapértelmezett érték");

JavaScript – felugró ablakok:

Figyelmeztető ablak:

alert("*üzenet-szöveg*");

Megerősítő ablak:

confirm("*üzenet-szöveg*");
Kétféle (true/false) logikai értéket vehet fel!

Beviteli ablak:

prompt("*üzenet-szöveg*","*alapértelmezett érték*");
A beírt szöveget vagy null-t vehet fel értékként!

Sortörés az üzenet-szövegben: '\n'

```
alert("Első sor" + '\n' + "Második sor");
confirm("Első sor" + '\n' + "Második sor");
prompt("Első sor" + '\n' + "Második sor","Alapértelmezett érték");
```

Példák:

Megerősítő ablak:

```
function show_confirm()
{
var r=confirm("Press a button!");
if (r==true)
  {
  alert("You pressed OK!");
  }
else
  {
  alert("You pressed Cancel!");
  }
}

<input type="button" onclick="show_confirm()" value="Show a confirm box" />
```

Bevitali ablak:

```
function show_prompt()
{
var name=prompt("Please enter your name","Harry Potter");
if (name!=null && name!="")
  {
  document.write("Hello " + name + "! How are you today?");
  }
}

<input type="button" onclick="show_prompt()" value="Show prompt box" />
```

XII. JavaScript – függvények

A JS-függvények valamely esemény bekövetkezésekor vagy a parancssor meghívására lépnek életbe.

JavaScript – függvények

Annak értdekében, hogy a böngésző ne az oldal betöltésekor hajtsa végre az összes JS-rendelkezést, annek egyes részleteit függvényekbe zárhatjuk.

A függvényekbe írt kódot a böngésző csak bizonyos felhaszálói akció vagy egy különálló meghívó-parancsra hajtja végre. A függvények az oldal bármely részéből ill. más oldalakról is meghívhatók (feltéve, hogy utóbbi esetben a függvény egy külső .js fájlban található).

A függvények így a dokumentumnek mind a fejrészében, mind pedig (szöveg)testében megtalálhatók lehetnek; azonban annak érdekében, hogy a böngésző biztos beolvassa, mielőtt meghívásra kerülne, célszerű a **<head>** részbe írni.

Függvények definiálása

A függvény-definiálás mondattana:
function ***függvény-név*(*változó_1*,*változó_2*,…,*változó_x*)**
{
végrehajtandó kód
}

A függvény paraméterei a ***változó_1…változó_x***-szel jelölt változók ill. konkrét értékek. A **{** és **}** kapcsos zárójel a függvény(érték) kezdetét és végét jelzi.

A paraméterekkel nem rendelkező függvény neve után is fel kell tüntetnünk a zárójeleket!
Ne feledkezzünk meg a JS esetfüggéséről, azaz a **function** szót mindig kisbetűvel írjuk (különben a JS nem működik), ill. a függvény-névre mindig azonos alakban hivatkozzunk (különös tekintettel a kis- vagy nagybetűkre!

Egy gyors példa

Következő példánkban egy gomb megnyomásával meghívott függvény figyelmeztető ablakot jeleníttet meg a böngészővel:

```
<html>
<head>
<script type="text/javascript">
function displaymessage()
{
alert("Hello World!");
}
</script>
</head>

<body>
<form>
<input type="button" value="Click me!" onclick="displaymessage()" />
</form>

<p>By pressing the button above, a function will be called. The
function will alert a message.</p>

</body>
</html>
```

Mint látjuk, a **Click me!** Feliratú gomb megnyomásakor a **displaymessage** függvény megjeleníti a **Hello World!** feliratú figyelmeztető ablakot.

Hogyha a **<script>** elembe írt **alert(”Hello World!”);** rendelkezést nem ágyaztuk volna be egy függvénybe (jelen esetben a **displaymessage**-be), akkor már az oldal megjelenítésekor végrehajtotta volna a böngésző. Jelen esetben azonban a rendelkezésnek megfelelő üzenet csak a felhasználó gombnyomására (azaz egy meghatározott eseményre) jelenik meg.

A JS-eseményekről egy későbbi fejezetben (JavaScript – események) lesz szó.

A return rendelkezés
A függvényeket nemcsak parancsok végrehajtására, hanem értékek átalakítására is használhatjuk. A **return** rendelkezéssel beállíthatjuk, hogy a betáplálthoz képest milyen értéket szolgáltasson vissza a függvény.

Tehát az érték-képzésre használt függvényeknek **return** rendelkezést tartalmaznak.

A következő példánkban szereplő függvény visszaszolgáltatott értéke két változó (**a** és **b**) szorzata:

```
<html>
<head>
<script type="text/javascript">
function product(a,b)
{
return a*b;
}
</script>
</head>

<body>
<script type="text/javascript">
document.write(product(4,3));
</script>

<p>The script in the body section calls a function with two parameters
(4 and 3).</p>
<p>The function will return the product of these two parameters.</p>
</body>
</html>
```

Mint látjuk, a szövegtestbe írt **document.write** parancs meghívja a fejrészben lévő **product** függvényt a **4,3** paraméterekkel.
A függvény végrehajtja a **return a*b;** parancsot a két számon, mint **a** ill. **b** értéken, és a böngésző ezt írja ki.

Nem kell azonban közvetlenül a **product** függvénybe ill. annak meghívásába beírnunk a változók aktuális értékét, elegendő azoka különállóan definiálni:

```
<html>
<head>
<script type="text/javascript">
function product(a,b)
{
return a*b;
}
</script>
</head>

<body>
<script type="text/javascript">
var q=5;
var w=5;
```

```
document.write(product(q,w));
</script>

<p>The script in the body section calls a function with two parameters
(4 and 3).</p>
<p>The function will return the product of these two parameters.</p>
</body>
</html>
```

Az értékeket pedig magába a függvénybe ís beleírhatjuk:

```
<html>
<head>
<script type="text/javascript">
function product(a,b)
{
a=3;
b=4;
return a*b;
}
</script>
</head>

<body>
<script type="text/javascript">
document.write(product());
</script>

<p>The script in the body section calls a function with two parameters
(4 and 3).</p>
<p>The function will return the product of these two parameters.</p>
</body>
</html>
```

vagy még egyszerűbben:

```
<html>
<head>
<script type="text/javascript">
function product()
{
return 3*4;
}
</script>
</head>

<body>
<script type="text/javascript">
document.write(product());
</script>

<p>The script in the body section calls a function with two parameters
(4 and 3).</p>
<p>The function will return the product of these two parameters.</p>
</body>
</html>
```

Persze az utóbbi két megoldásnak csak elvi jelentőség van.

A JavaScript – változók érvényessége
A függvényeken belül (azok definiálásakor) meghatározott változók csak a függvényre vonatkoznak, azon kívül nem érvényesek. A függvény végrehajtása után a változó törlődik. Emiatt helyi változóknak (=local variables) nevezzük őket.
Mindezek értelmében tetszőleges számú függvény tartalmazhat ugyanolyan nevű helyi változókat, mivel ezek közül mindegyik csak az adott függvényre vonatkozik, így nam zavarja a többi működését.

Ezzel szemben a függvényeken kívül definiált változók az összes függvényre ill. egyéb műveletre vonatkoznak. Érvényességi idejük a definiálásuktól az oldal bezárásáig terjed.

Ennek megfelelően a függvény paramétereit a függvény definiálásakor helyi változó-nevekkel (pl. **a** és **b**) jelölhetjük; majd a függvényre való hivatkozáskor az éppen aktuális értékeket (konkrét számokat vagy más változókat) helyettesíthetünk be az előbbi változók vesszővel elválasztott helyére; ezzel voltaképpen a fügvény helyi változóihoz rendelünk új értékeket [azaz voltaképpen definiáljuk a függvény helyi változói(nak értéké)t]. Így a függvényen kívül és belül eltérő változók szerepelnek, melyek között a kapcsolatot a függvény-paraméterben elfoglalt helyük jelenti.

További példák
Első példánkban az előbb említett eljárást látjuk, ti. a függvényre való külső hivatkozásban megadott (szöveg) értéket felveszi a függvény-paraméterként szereplő helyi **txt** változó, s ezt az értéket aztán egy figyelmeztető ablakban jelenítteti meg a függvény:

```
<html>
<head>
<script type="text/javascript">
function myfunction(txt)
{
alert(txt);
}
</script>
</head>
<body>

<form>
<input type="button" onclick="myfunction('Hello')" value="Call
function">
</form>

<p>By pressing the button above, a function will be called with "Hello"
as a parameter. The function will alert the parameter.</p>

</body>
</html>
```

Mint látjuk, a **Call function** feliratú gomb megnyomására felnyílik a **txt** változó értékét (**Hello**) tartalmazó figyelmeztető ablak. A szöveges értéket itt is idézőjelbe tettük, ami (mivel a függvény-meghívás maga is ketős idézőjelben van,) egyszeres.

Második, igen egyszerű példánkban a függvény meghívása egy értéket szolgáltat, melyet a dokumentumtestbe írt rendelkezés kiír:

```
<html>
<head>
<script type="text/javascript">
function myFunction()
{
return ("Hello world!");
}
```

```
</script>
</head>
<body>

<script type="text/javascript">
document.write(myFunction())
</script>

</body>
</html>
```

A felirat tehát **Hello world!.**

JavaScript – függvények:

Függvények definiálása:

function *függvény-név*(*változó_1*,*változó_2*,…,*változó_x*)
{
végrehajtandó kód
}

JS-változók érvényessége:

- **Belső változók:** a függvények, feltételes rendelkezések paramétereibe és kódjába írt változók, melyek máshol nem szerepelnek az oldalban. Csak az adott parancs-tömbön belül érvényesek.
- **Külső változók:** A JS-kódban több tömbben is előforduló változók, melyeket az egymás után következő műveletek az összes későbbire nézve felülírnak.

Függvények meghívása:

<tag *esemény-attribútum*=”*függvény-név*(*érték_1,érték_2,…,érték_x*)” />

A return rendelkezés:

Az előbbi általános alakú függvénybe beírva a
return *változó_1+változó2+változó_x*;
kódot, a függvény felveszi a művelet értékét.

Példák:

Függvény definiálása:

```
function displaymessage()
{
alert("Hello World!");
}

<input type="button" value="Click me!" onclick="displaymessage()" />
```

A return rendelkezés:

```
function product(a,b)
{
return a*b;
}
document.write(product(4,3));
```

vagy:

```
function product(a,b)
{
return a*b;
}
```

```
var q=5;
var w=5;
document.write(product(q,w));
```

vagy:

```
function product(a,b)
{
a=3;
b=4;
return a*b;
}
document.write(product());
```

vagy:

```
function product()
{
return 3*4;
}
document.write(product());
```

Gombspecifikus figyelmeztetés:

```
function myfunction(txt)
{
alert(txt);
}
<input type="button" onclick="myfunction('Hello!')" value="Call function1">
<input type="button" onclick="myfunction('Good morning!')" value="Call function2">
```

XII. JavaScript – ciklusok: a for ciklus

A JS ciklusok során egy adott kódot előre megadott számban vagy egy feltétel fennállása alatt, ismételten végrehajtunk.

JavaScript – ciklusok

Gyakran előfordul, hogy egy parancs-sorozatot egymás után többször is végre kell hajtanunk. Ebben az esetben a közel azonos tartalmú részletek egymás utáni leírása helyett egyszerűbb egy ún. ciklust (=loop) beleírnunk a kódba.

A JS-ben kétféle ciklus használható:

- **for** a kód-tömb végrehajtását bizonyos, előre megadott számú alkalommal ismétli.
- **while** a kód-tömb végrehajtását egy előre megadott feltétel teljesüléséig ismétli.

A for cilkus

A **for** ciklust akkor használjuk, ha előre tudjuk, hány alkalommal kell lefuttatnunk a kódot.

A for ciklus mondattana:
for (*változó=kezdőérték*;*változó<=záróérték*;*változó=változó+növekmény*)
{
végrehajtandó kód
}

Értelmezzük az előbbi általános alakot egy konkrét példán keresztül!
Első példánkban az eredetileg **0** értékű **i** változót **5**-ig növeljük, és aktuális értékét minden egyes ciklusban kiíratjuk:

```
<html>
<body>
<script type="text/javascript">
var i=0;
for (i=0;i<=5;i++)
{
document.write("The number is " + i);
document.write("<br />");
}
</script>
</body>
</html>
```

Mint látjuk, először az **i** változót és annak értékét definiáljuk. Ezt akár el is hagyhatjuk, hiszen a **for** függvénybe írt **i=0** paraméter ezzel egyező értelmű; s egyúttal a kezdőértéket is meghatározza. Hogyha a **var i=9;** meghatározással élünk, de a függvényben továbbra is **i=0** áll, akkor a függvény a saját, helyi változója szerint számol; csak akkor tér át a külső változóra, ha a belsőnek nincs önálló értéke (vagyis egyszerűen csak **i** szerepel első paraméterként).
A második paraméter értelmében (ahová bármilyen más relációt is beírhatnánk), a parancssort addig ismételjük, míg **i** értéke kisebb **5**-nél, ill. legutoljára, mikor eléri azt.
A harmadik paraméter értelmében **i** értéke minden ciklus után eggyel megnő. Ide dekrementumot, vagy valamilyen más hozzárendelő műveletet írhatnánk (lásd a JavaScript – műveletek fejezetet).

Ennek megfelelően az oldalon megjelenő szöveg:
The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5

Az előbbi általános megfontolásoknak megfelelően át is alakíthatjuk a kódot:

```
<html>
<body>

<script type="text/javascript">
var i=3;
for (i; i >= -5; i-=2)
{
document.write("The number is " + i);
document.write("<br />");
}
</script>

</body>
</html>
```

Itt **i**-t külsőleg definiáljuk, értéke **3**. A parancs-tömb addig ismétlődik, míg értéke **-5**-nél nagyobb ill. azzal egyenlő nem lesz. Az **i** minden ciklus után **2**-vel csökken, amit az **i-=2** irövidítve felírt inkrementáló (hozzárendelő-)művelettel érünk el, melynek teljes alakja (melyet szintén alkalmazhatunk): **i=i-2**.

A megjelenő felirat eszerint:
The number is 3
The number is 1
The number is -1
The number is -3
The number is -5

Egy további példa: a címsor-típusok kiíratása
A **for** ciklust felhasználhatjuk a hatféle HTML-címsor azonos tartalommal való kiíratására, ti. a címsor-tagbe írandó címsor-rendűség sorozatos megváltoztatása révén:

```
<html>
<body>

<script type="text/javascript">
for (i = 1; i <= 6; i++)
{
document.write("<h" + i + ">This is heading " + i);
document.write("</h" + i + ">");
}
</script>

</body>
</html>
```

Mint látszik, **i** a ciklusok során itt egytől egyenként hatig nő, tehát a két részletből összeállított HTML-kódban a **This is heading** szöveget (és **i** aktuális értékét) egyre növekvő rendűségű címsor-tagek (**<h1>…<h6>**) fogják közre. Ennek megfelelően – egyre kisebbedő betűmérettel – megjelennek a címsorok:

This is heading 1
This is heading 2
This is heading 3
This is heading 4
This is heading 5
This is heading 6

Természetesen a végrehajtandó HTML-kódot összeállító **document.write** parancsot egyetlen sorba is írhatjuk:

```
document.write("<h" + i + ">This is heading " + i + "</h" + i + ">");
```

A függvény-paraméterek módosításával meg is fordíthatjuk a címsorok kiírási sorrendjét:

```
<html>
<body>

<script type="text/javascript">
for (i = 6; i >= 1; i--)
{
document.write("<h" + i + ">This is heading " + i + "</h" + i + ">");
}
</script>

</body>
</html>
```

JavaScript – ciklusok: a for ciklus:

JavaScript – ciklusok:

- **for** a kód-tömb végrehajtását bizonyos, előre megadott számú alkalommal ismétli.
- **while** a kód-tömb végrehajtását egy előre megadott feltétel teljesüléséig ismétli.

A for cilkus

A **for** ciklust akkor használjuk, ha előre tudjuk, hány alkalommal kell lefuttatnunk a kódot.

Mondattana:

for (*változó=kezdőérték;változó<=záróérték;változó=változó+növekmény*)
{
végrehajtandó kód
}

Példák:

Számok kiíratása:

```
for (i=0;i<=5;i++)
{
document.write("The number is " + i + ”<br />”);
}
```

Címsorok kiíratása:

```
for (i = 1; i <= 6; i++)
{
document.write("<h" + i + ">This is heading " + i + "</h" + i + ">");
}
```

XIII. JavaScript – ciklusok: a while és a do while ciklus

A JS ciklusok során egy adott kódot előre megadott számban vagy egy feltétel fennállása alatt, ismételten végrehajtunk.

A while ciklus

A **while** ciklus-függvény addig futtatja az értékéül megadott parancssort, amíg a paraméteréül megadott feltétel teljesül (érvényben van).
Hogyha a feltétel már kezdetben sem teljesül, a parancssor egyszer sem fut le!

A while ciklus mondattana:
while(*változó<=végérték*)
{
végrehajtandó kód
}

A **<=** helyett bármilyen más művelet is szerepelhet!

A fenti mondattan használatát egy példán keresztül mutatjuk be;
aholis egy szám értéke a rajta a kódban elvégzett műveletek során **0**-tól kezdve növekszik, míg túl nem lépi a paraméterben megadott feltételt:

```
<html>
<body>

<script type="text/javascript">
i=0;
while (i<=5)
{
document.write("The number is " + i);
document.write("<br />");
i++;
}
</script>

<p>Explanation:</p>
<p><b>i</b> is equal to 0.</p>
<p>While <b>i</b> is less than , or equal to, 5, the loop will continue
to run.</p>
<p><b>i</b> will increase by 1 each time the loop runs.</p>

</body>
</html>
```

Mint látjuk, **i** értéke kezdetben **0**. A végrehajtott parancssor először kiírja ezt az értéket (**The number is 0**), majd sortörést végez, és **i**-t megnöveli eggyel. Eután **i** értéke ismét kiírásra kerül, majd megint megnő eggyel, stb., mígnem értéke eléri az **5**-öt, amit szintén kiír még a program, de mivel annak lefutása után **i** értéke már **6**, a ciklus leáll.

Ennek megfelelően a megjelenő felirat:
The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5

A do while ciklus

A **do while** ciklus a **while** ciklus változata. Mint láttuk, a **while** ciklus nem fut le, hogyha a paraméterében megadott feltétel már kezdetben sem teljesül (azaz annak logikai értéke HAMIS).

A **do while** ciklus feltételét azonban a **while** függvény tartalmazza, melyet a végrehajtandó kód után írunk; amint az a mondattanából látszik:
do
{
végrehajtandó kód
}
while(*változó*<=*végérték*); (<= helyett természetesen más hozzárendelés is elképzelhető.)

Ez azt jelenti, hogy a parancs-tömböt a böngésző mindenképpen végrehajtja egyszer, mielőtt ellenőrizné, hogy a feltétel teljesül-e. Ha NEM, akkor nem ismétli a kód futtatását; ha IGEN, akkor pedig mindaddig ismétli, míg a feltétel IGAZ.

Lássunk egy példát az előbbi általánosságok gyakorlati alkalmazására:

```
<html>
<body>

<script type="text/javascript">
i = 0;
do
{
document.write("The number is " + i);
document.write("<br />");
i++;
}
while (i <= 5)
</script>

<p>Explanation:</p>

<p><b>i</b>  equal to 0.</p>

<p>The loop will run</p>

<p><b>i</b> will increase by 1 each time the loop runs.</p>

<p>While <b>i</b> is less than , or equal to, 5, the loop will continue
to run.</p>

</body>
</html>
```

Mint látjuk, ez a megelőző példánk átdolgozása a **do while** ciklusohoz, így eredménye a fentivel megegyezik. Tehát a kezdetben **0** értékű **i** értékét egy sorközzel együtt kiíratjuk, majd értékét eggyel megnöveljük, és ellenőrizzük, hogy így is teljesül-e még a feltétel. Hogyha teljesül, a parancssort megismételjük, stb..

A felirat tehát ugyanaz, mint korábban:
The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5

Hogyha módosítjuk a példát;

```
<html>
<body>

<script type="text/javascript">
i = 5;
do
{
document.write("The number is " + i + "<br />");
i++;
}
while (i <= 5)
</script>

</body>
</html>
```

akkor sem mutatkozik különbség, **i** ilyen (**5**-ös) értékénél a **while** függvény még egyszer futtatja a paranssort (hisz 5 ugyan már nem kiseb 5-nél, de még egyenlő vele), tehát kiíratja a
The number is 5
szöveget; a **do while** ciklus pedig eleve kiíratja azt, majd elllenőrzi, hogy az új, 6-os érték megfelel-e még a feltételnek, és mivel nem, nem futtatja le újra a parancssort.

JavaScript – ciklusok: a while és a do while ciklus:

A while ciklus:

A **while** ciklus-függvény addig futtatja az értékéül megadott parancssort, míg a paraméteréül megadott feltétel teljesül. Hogyha a feltétel már kezdetben sem teljesül, a parancssor egyszer sem fut le!

Mondattana:

while(*változó<=végérték*)
{
végrehajtandó kód
}
A **<=** helyett bármilyen más művelet is szerepelhet!

A do while ciklus:

A **do while** ciklus feltételét a kód után írt **while** függvény tartalmazza; ezért a parancssor mindenképpen lefut egyszer, mielőtt a program ellenőrzi, hogy a feltétel teljesül-e. Ha NEM, akkor nem ismétli a kód futtatását; ha IGEN, akkor pedig mindaddig ismétli, míg a feltétel IGAZ.

Mondattana:

do
{
végrehajtandó kód
}
while(*változó<=végérték*);
<= helyett más hozzárendelés is szerepelhet!

Példák:

while ciklus:

```
i=0;
while (i<=5)
{
document.write("The number is " + i);
document.write("<br />");
i++;
```

```
}
```

do while ciklus:

```
i = 0;
do
{
document.write("The number is " + i);
document.write("<br />");
i++;
}
while (i <= 5)
```

XIV. A break és continue rendelkezés

A break rendelkezés

A **break** rendelkezés beolvasásakor a böngésző leállítja az éppen (pl. ciklus-függvény értékeként) futó kód-blokk végrehajtását és a következő kód-részletre tér át (amennyiben talál ilyet), pl.:

```
<html>
<body>
<script type="text/javascript">
var i=0;
for (i=0;i<=10;i++)
{
if (i==3)
  {
  break;
  }
document.write("The number is " + i);
document.write("<br />");
}
</script>
<p>Explanation: The loop will break when i=3.</p>
</body>
</html>
```

Mint látjuk, példánkban az eredetileg **0** értékű **i** a tízig tartó, egyszerű növekményű **for** ciklus-függvénybe kerül, mely először ellenőrzi, hogy értéke három-e. Amennyiben nem, kiíratja az értéket, és folytatja a ciklust mindaddig, míg az **i==3** feltétel nem teljesül. Ekkor azonban életbe lép a **break;** rendelkezés, ami folytán ez az érték már nem kerül kiírásra, hanem a parancssor leáll.
A megjelenő felirat tehát:
The number is 0
The number is 1
The number is 2

Hogyha a függvényt 10-ről csökkenőnek választjuk [**var i=10;for(i;i>=0;i--)**], a felirat:
The number is 10
The number is 9
The number is 8
The number is 7
The number is 6
The number is 5
The number is 4

Hogyha pedig **var i=3;** akkor egyáltalán nincs felirat.

A continue rendelkezés

A **continue** rendelkezés a feltétel teljesülésekor csak az adott értékre nézve állítja le a kód további részének végrehajtását, és újraindítja a ciklust a következő értékkel, pl.:

```
<html>
<body>
<script type="text/javascript">
var i=0;
for (i=0;i<=10;i++)
{
if (i==3)
  {
  continue;
  }
document.write("The number is " + i);
```

```
document.write("<br />");
}
</script>

<p>Explanation: The loop will break the current loop and continue with
the next value when i=3.</p>
</body>
</html>
```

Ebben az esetben az **if** függvény **i==3** feltételének teljesülésekor a **continue;** rendelkezés meggátolja a **3**-as értékre vonatkozó kiírási művelet végrehajtását, és a következő (**i=4**-es) értékkel újraindítja a ciklust.
Ennek megfelelően a keletkező számsor:

The number is 0
The number is 1
The number is 2
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
The number is 10

Mint látjuk, a hármas szám nem szerepel. A for függvény számolási irányának megfordításakor is ugyanezt tapasztaljuk (csak a sorozat fordul meg).

A break és continue rendelkezés:

Break rendelkezés: break;

A **break;** rendelkezés beolvasásakor a böngésző leállítja az éppen futó kód-blokk végrehajtását és áttér a következőre.

Continue rendelkezés: continue;

A **continue** rendelkezés (pl. egy feltétel teljesülésekor) csak az adott értékre nézve állítja le a kód további részének végrehajtását, és újraindítja a tömböt/ciklust a következő értékkel.

Példák:

Break rendelkezés:

```
var i=0;
for (i=0;i<=10;i++)
{
if (i==3)
  {
  break;
  }
document.write("The number is " + i);
document.write("<br />");
}
```

Continue rendelkezés:

```
var i=0;
for (i=0;i<=10;i++)
{
if (i==3)
  {
  continue;
  }
```

```
document.write("The number is " + i);
document.write("<br />");
}
```

XV. JavaScript – a for in rendelkezés

A for in rendelkezés

A **for in** rendelkezés kódja egy adott objektum (=object) bizonyos változóján (=variable) egyszer fut végig.

Mondattana:
for(*változó* in *objektum*)
{
végrehajtandó kód
}

A kódot tehát az objektum (pl. változók sorozata) minden egyes elemén végrehajtjuk egyszer. Így az objektum összes elemén megismérteljük a kód végrehajtását.
A **for** függvény paraméteréül megadott **változó** lehet pl. egy névvel nevezett változó, egy adattömb (=array) eleme vagy egy objektum jellemzője.

A **for in** rendelkezés működését egy gyors példán keresztül mutatjuk be:

```
<html>
<body>
<script type="text/javascript">
/*var x;*/
var mycars = new Array();
mycars[0] = "Saab";
mycars[1] = "Volvo";
mycars[2] = "BMW";

for (x in mycars)
{
document.write(mycars[x] + "<br />");
}
</script>
</body>
</html>
```

Mint látjuk, a **for** függvény itt a **mycars** adattömböt alkotó **mycars[x]** változókon hajtja végre a kiíratási rendelkezést. Mivel az utóbbi egyúttal a **mycars[x]** változókban szereplő **x** változó értékeit, tehát magát az **x**-et is, definiálja, ezt felesleges külön megtennünk – amint megjegyzés-jelek közé is tettük.

A megjelenő felirat:
Saab
Volvo
BMW

A for in rendelkezés:

A **for in** rendelkezés kódja egy adott objektum bizonyos vagy összes változóján végrehajtásra kerül egyszer.

Mondattana:
for(*változó* in *objektum*)
{
végrehajtandó kód
}

Példa a for in rendelkezésre:

```
var mycars = new Array();
mycars[0] = "Saab";
mycars[1] = "Volvo";
```

```
mycars[2] = "BMW";

for (x in mycars)
{
document.write(mycars[x] + "<br />");
}
```

XVI. JavaScript – események

A JS-események (=events) olyan (pl. felhasználói) akciók, melyeket a JS felismer.

Események

A JS használatával dinamikus weboldalakat készíthetünk. Ennek érdekében fel kell ismernünk a felhasználó akcióit, amire az ún. JavaScript események (=events) nyújtanak lehetőséget.

Minden weboldal-elem rendelkezet bizonyos esemény-típussal, amelyhez hozzárendelhetjük a JavaScript elindulását (vagy valamilyen, a korábbitól eltérő formázást). Pl. egy nyomógomb esetében az **onClick** attribútum (és esemény-név) segítségével utalhatunk a felhasználó gombnyomásakor végrehajtandó JS-parancssorra. A HTML-elemekhez társított eseményeket tehát a HTML-tagekbe, attribútumként írjuk.

Példák esemény-típusokra:

- egérkattintás;
- weboldal vagy kép betöltése;
- egér-értintés;
- beviteli mező kijelölése (egy HTML-űrlapon);
- (Submit) gomb megnyomása;
- billentyű-leütés.

Az eseményekkel legtöbbször JS-függvényekre utalunk, melyek nem futnak le mindaddig, míg az esemény be nem következik.

Az összes JavaScript-esemény megtalálható JavaScript – útmutatónkban.

Az onLoad és onUnload esemény

Az **onLoad** és **onUnload** események az oldal betöltésekor ill. bezárásakor következnek be.

Az **onLoad** eseményt gyakorta használjuk a felhasználó böngésző-típusának és -verziójának ellenőrzésére, hogy az ennek megfelelő weboldal-verziót jeleníthessük meg.

Az **onLoad** és **onUnload** eseményeket a felhasználó oldalra való be- vagy kilépésekor készítendő cookie-k kezelésére is gyakran alkalmazzuk. Pl. az első belépéskor egy beviteli ablakkal rákérdezhetünk a felhasználó nevére, és eltárolhatjuk azt egy cookie-ban. Amikor a felhasználó legközelebb megnyitja az oldalt, már a saját nevét tartalmazó üdvözletet jeleníthetünk meg számára (az oldalon vagy egy felugró ablakban).

Az onFocus, onBlur és onChange esemény

Az **onFocus**, **onBlur** és **onChange** eseményeket leginkább (a felhasználó által megadott) értékek validálására használjuk, pl. űrlapoknál.

Alább az **onChange** esemény alkalmazására látunk példát:

```
<input type="text" size="30" id="email" onchange="checkEmail()">
```

Ha a felhasználó megváltoztatja a beviteli mező tartalmát, a böngésző az új tartalomra vonatkozólag lefuttatja a **checkEmail** függvényt.

Az onSubmit esemény

Az **onSubmit** esemény az összes beviteli mezőn értékének, vagyis a teljes űrlap-tartalom továbbítás előtti ellenőrzésére szolgál.

Alább látunk erre egy példát:

```
<form method="post" action="xxx.htm" onsubmit="return checkForm()">
```

Mint látjuk, az **onSubmit** esemény-attribútumot a **<form>** tagbe (és nem a submit gombba) írtuk. Az űrlap továbbítása előtt a **checkForm** nevű függvény ellenőrzi a beírt adatokat. Hogyha azok helyesek, a

függvény logikai értéke IGAZ, és az űrlap továbbításra kerül. Ellenkező esetben a továbbítási művelet megszakad.

Az onMouseOver és onMouseOut esemény
Az **onMouseOver** és **onMouseOut** eseményeket leginkább egyfajta ál-nyomógombok készítéséhez használjuk.

Pl. az alábbi link egér-érintésekor egy figyelmeztető ablak nyílik meg (**An onMouseOver event** felirattal):

```
<a href="http://www.w3schools.com" onmouseover="alert('An onMouseOver event');return false"><img src="w3s.gif" alt="W3Schools" /></a>
```

Események:
A JS-események olyan akciók, melyeket a JS felismer.

Esemény-attribútumok:

Kijelölés megszűnése (pl. szövegmezőé):	onblur
Tartalom-változás:	onchange
Egér-kattintás:	onclick
Dupla egérkattintás:	ondblclick
Betöltési hiba:	onerror
Kijelölés:	onfocus
Billentyű lenyomása:	onkeydown
Billentyű lenyomása vagy lenntartása:	onkeypress
Billenytyű felengedése:	onkeyup
Egérgomb lenyomása:	onmousedown
Egérkurzor elmozdítása:	onmousemove
Egérérintés megszűnése:	onmouseout
Egérérintés (kezdete):	onmouseover
Egérgomb felengedése:	onmouseup
Ablak átméretezése:	onresize
Szöveg-kijelölés:	onselect
Weboldal betöltése:	onload
Weboldal bezárása:	onunload
Tartalom (űrlap-adatok) továbbítása:	onsubmit

Példák:
onchange:

```
<input type="text" size="30" id="email" onchange="checkEmail()">
```

onsubmit:

```
<form method="post" action="xxx.htm" onsubmit="return checkForm()">
```

onmouseover:

```
<a href="http://www.w3schools.com" onmouseover="alert('An onMouseOver event');return false">
<img src="w3s.gif" alt="W3Schools" />
</a>
```

XVII. JavaScript – try catch rendelkezés

A **try catch** rendelkezéssel kód-tömbökben lévő hibákat deríthetünk fel.

JavaScript – hibakeresés (catching)

Az Internetet böngészve találkozhattunk olyan oldalakkal, melyeknél egy felugró ablak az oldal JS kódjának működés közbeni hibájára figyelmeztetett bennünket, és feltette a **Do you wish to debug?** kérdést.

Az efféle hibaüzenetek leginkább a webes fejlesztők számára hasznosak, a közönséges felhaszálók ilyenkor egyszerűen elhagyják az oldalt.

A következőkben bemutatjuk, hogy lehet elkerülni az ilyen helyzetek kialakulását és így az oldal látogatottságának csökkenését.

A try catch rendelkezés

A **try catch** rendelkezés **try** részébe írt kódot a böngésző megpróbálja végrehajtani (mint az oldal normális tartalmát). Hogya eközben valami hiba lép fel, akkor a **catch** rész ennek okát megkeresi és egy hibaüzenetben kiírja. Ezután különféle lehetőségeket ajánlhatunk fel a felhasználónak a továbblépésre. Így az oldal megtekintője értesül a probléma okáról és lehetősége van tovább böngészni az oldalt.

Mondattan:

try
{
végrehajtandó kód
}
catch(err)
{
hiba esetén végrehajtandó kód
}

A **try** ill. **catch** szavakat itt is kisbetűvel kell írnunk, különben a JS nem működik!

Példák

Első példánkban egy gomb megnyomásakor **Welcome guest!** feliratú figyelmeztető ablaknak kellene megjelennie, csakhogy az erre utaló (**try** alatti) kód hibás: **alert** helyett **adddlert** szót tartalmaz:

```
<html>
<head>
<script type="text/javascript">
var txt="";
function message()
{
try
  {
  adddlert("Welcome guest!");
  }
catch(err)
  {
  txt="There was an error on this page.\n\n";
  txt+="Error description: " + err.description + "\n\n";
  txt+="Click OK to continue.\n\n";
  alert(txt);
  }
}
</script>
</head>
```

```
<body>
<input type="button" value="View message" onclick="message()" />
</body>
</html>
```

Ezért a **catch** függvény lép életbe, aminek **err** paramétere (változója) felveszi azt a szöveges értéket, melyet a böngésző a kód hiba-vizsgálata után kiad (**A várt elem objektum**). Ezt az értéket a **catch** alatti parancssor beépíti a **txt** változó értékébe, majd a **txt** változót kiíratja egy figyelmeztető ablakba. A figyelmeztető ablak nyugtázása (**OK**) vagy kikapcsolása (**X**) után az oldal egyaránt tovább használható.

A **txt** értékét megadó kódot természetesen össze is vonhatjuk:

```
<html>
<head>
<script type="text/javascript">
var txt="";
function message()
{
try
  {
  adddlert("Welcome guest!");
  }
catch(err)
  {
  txt="There was an error on this page.\n\n" + "Error description: " +
err.description + "\n\n Click OK to continue.\n\n";
  alert(txt);
  }
}
</script>
</head>

<body>
<input type="button" value="View message" onclick="message()" />
</body>
</html>
```

Második példánkban az előzővel azonos hiba miatt egy megerősítő ablak jelenik meg. Az ablak döntésünk nyomán kelt logikai értékének megfelelően (**OK** = IGAZ; **Cancel/X** = HAMIS) visszatérünk az oldalra, vagy a böngésző visszairányít minket a honlapra.

Mint látjuk, a függvények itt is a fejrészben, tehát a dokumentumtesttől elkülönítve szerepelnek!

```
<html>
<head>
<script type="text/javascript">
var txt="";
function message()
{
try
  {
  adddlert("Welcome guest!");
  }
catch(err)
  {
  txt="There was an error on this page.\n\n";
  txt+="Click OK to continue viewing this page,\n";
  txt+="or Cancel to return to the home page.\n\n";
```

```
  if(!confirm(txt))
    {
    document.location.href="http://www.w3schools.com/";
    }
  }
}
</script>
</head>

<body>
<input type="button" value="View message" onclick="message()" />
</body>
</html>
```

A **View message** gomb megnyomása és a **try** tömbbe írt kód hibás futása után működésbe lép a **catch** függvény. Először az **err** változó felveszi a hibaüzenet-értéket, melyet azonban itt nem használunk fel, hanem egy általános szöveget iratunk ki az addig üres **txt** változóba. Ezután megnyílik a **confirm** ablak, benne a **txt** változó értékével, azaz a **catch** alatti parancssorban megadott szöveggel
(**There was an error on this page.**
Click OK to continue viewing this page,
or Cancel to return to the home page.).

Az **if** függvényt a megerősítő ablak logikai értékének, tehát a felhasználó döntésének kiértékelésére használjuk. Hogyha a a **confirm** ablak értéke NEM (**!**) (nem **confirm**, azaz nem megerősítő), akkor a függvény értékébe írt parancssor szerint visszalépünk a kezőoldalra (**http://www.w3schools.com/**). Hogyha az ablak értéke IGEN, akkor az **if** függvény értéke NEM, tehát a link nem lép érvénybe, és megmaradunk az eredeti oldalon.

Ennek megfelelően az **OK** gomb megnyomásakor az eredeti oldalon maradunk, míg a **Cancel** vagy az **X** nyomásakor visszakerülünk a kezdőoldalra.

A throw rendelkezés
A **throw** rendelkezést a **try catch** kiegészítéseként használhatjuk, abban az esetben, hogyha (több) konkrét hiba-lehetőséget és válaszépést kívánunk megfogalmazni. Részletesebben lásd a következő fejezetet!

A try catch rendelkezés:
A **try catch** rendelkezés **try** részébe írt kódot a böngésző megpróbálja végrehajtani. Ha valami hiba lép fel, akkor ezt félbehagyva (a már lefutott kód eredménye látható marad!) átugrik a **catch** részbe írt kód végrehajtására.

Mondattana:

```
try
{
végrehajtandó kód
}
catch(err)
{
hiba esetén végrehajtandó kód
}
```

Példák:
Hibaüzenet:

```
function message()
{
try
  {
```

```
  adddlert("Welcome guest!");
  }
catch(err)
  {
  txt="There was an error on this page.\n\n";
  txt+="Error description: " + err.description + "\n\n";
  txt+="Click OK to continue.\n\n";
  alert(txt);
  }
}

<input type="button" value="View message" onclick="message()" />
```

Megerősítő ablak navigációs opcióval:

```
function message()
{
try
  {
  adddlert("Welcome guest!");
  }
catch(err)
  {
  txt="There was an error on this page.\n\n";
  txt+="Click OK to continue viewing this page,\n";
  txt+="or Cancel to return to the home page.\n\n";
  if(!confirm(txt))
    {
    document.location.href="http://www.w3schools.com/";
    }
  }
}

<input type="button" value="View message" onclick="message()" />
```

XVIII. JavaScript – a throw rendelkezés

A throw rendelkezés

A **throw** rendelkezéssel tagoltabbá tehetjük az **try catch** rendelkezések végrehajtását, amennyiben külön, pontos hiba-lehetőségeket és válaszlépéseket (pl. hibaüzeneteket) fogalmazhatunk meg. Így elérhetjük, hogy a felhaználó rendeltetésszerűen hasznája az oldalt, miáltal a hibás működés is megszűnhet.

Mondattana:
throw ***megoldás***

A **megoldás** segítségével beállíthatjuk, mit tegyen a böngésző a hiba megszüntetése érdekében. Ez lehet egy parancssor (=string), egész szám (=integer), logikai (IGAZ/HAMIS) érték (=boolean) ill. valamilyen objektum (=object).

A **throw** parancsot mindig kisbetűvel írjuk!

Példa

Következő példánkban egy beviteli ablakba írt számot ellenőrzünk, mielőtt kiíratnánk azt a weboldalra:

```
<html>
<body>
<script type="text/javascript">
var x=prompt("Enter a number between 0 and 10:","");
try
{
if (x>=0 && x<=10)
  {
  document.write("Your number is " + x);
  }
if(x>10)
  {
  throw "Err1";
  }
else if(x<0)
  {
  throw "Err2";
  }
else if(isNaN(x))
  {
  throw "Err3";
  }
}
catch(er)
{
if(er=="Err1")
  {
  alert("Error! The value is too high");
  }
if(er=="Err2")
  {
  alert("Error! The value is too low");
  }
if(er=="Err3")
  {
  alert("Error! The value is not a number");
  }
}
```

```
</script>
</body>
</html>
```

Mint látjuk, a megnyíló beviteli ablakba beírt szám lesz az **x** változó értéke, amit azután megpróbálunk kiíratni a **try** parancssorba írt első **if** függvénnyel. Ennek végrehajtási feltétele, hogy $0 \leq \mathbf{x} \leq 10$.
Ha ez nem teljesül, a parancssor tovább fut, és ellenőrizni kezdi a beírt számot. Hogyha az túl magas, akkor átutal minket a **catch** függvény **er** változójának megfelelő értékéhez rednelt hibaüzenethet, melyet a gép rögtön ki is ír. A másik két hibalehetsőéghez [**x** túl kicsi ill. nem szám (**NaN** = not a number)] is ugyanígy egy-egy hibaüzenet tartozik.
A hibaüzenetek hatására a felhasználó ki fogja javítani a hibásan megadott számot, és így a kiíratő parancs működésbe lép, és megjelenik a **Your number is *x*** felirat.

A kód hibája, hogy a **prompt** ablak üres (**””**) értéke esetén is kiírja a **Your number is** szöveget. Ezt kiküszöbölhetjük a következő módosításokkal:

```
<html>
<body>
<script type="text/javascript">
var x=prompt("Enter a number between 0 and 10:","");
try
{
if (x>=0 && x<=10 && x!="")
  {
  document.write("Your number is " + x);
  }
else if(x=="")
  {
  throw "Err0";
  }
else if(x>10)
  {
  throw "Err1";
  }
else if(x<0)
  {
  throw "Err2";
  }
else if(isNaN(x))
  {
  throw "Err3";
  }
}
catch(er)
{
if(er=="Err0")
  {
  alert("Error! The value is missing!");
  }
if(er=="Err1")
  {
  alert("Error! The value is too high!");
  }
if(er=="Err2")
  {
  alert("Error! The value is too low!");
  }
```

```
if(er=="Err3")
  {
  alert("Error! The value is not a number!");
  }
}
</script>
</body>
</html>
```

Mint látjuk, itt már nem kerül kiírásra üres **x**-érték, hanem az **Err0** értékű **err**-változóhoz a **catch** függvényben hozzárendelt figyelmeztető ablak jelenik meg (**Error! The value is missing!**).A parancssorokat is átstrukturáltuk.

A throw rendelkezés:

A throw rendelkezés:

A **throw** rendelkezéssel tagoltabbá tehetjük az **if else** rendelkezések végrehajtását, amennyiben külön, pontos hiba-lehetőségeket és válaszlépéseket fogalmazhatunk meg. Egyúttal elérhetjük, hogy a felhasználó rendeltetésszerűen használja az oldalt.

Mondattana:
throw *megoldás*

Az isNaN() függvény:

Az **isNaN()** függvény paraméterébe írt adat szám- vagy egyéb jellegétől függően HAMIS vagy IGAZ értéket vesz fel.

Mondattana:
isNaN(*adat*)

Példa a throw rendelkezés és az isNaN() függvény alkalmazására:

```
var x=prompt("Enter a number between 0 and 10:","");
try
{
if (x>=0 && x<=10 && x!="")
  {
  document.write("Your number is " + x);
  }
else if(x=="")
  {
  throw "Err0";
  }
else if(x>10)
  {
  throw "Err1";
  }
else if(x<0)
  {
  throw "Err2";
  }
else if(isNaN(x))
  {
  throw "Err3";
  }
}
```

```
catch(er)
{
if(er=="Err0")
  {
  alert("Error! The value is missing!");
  }
if(er=="Err1")
  {
  alert("Error! The value is too high!");
  }
if(er=="Err2")
  {
  alert("Error! The value is too low!");
  }
if(er=="Err3")
  {
  alert("Error! The value is not a number!");
  }
}
```

XIX. JavaScript – különleges karakterek

A JS által megjelenített szövegek egyszerű formázását különleges karakterekkel végezzük, melyeket a parancssor többi elemétől fordított perjelekkel (\) különítünk el.

Különleges karakterek parancssorba írása

Az egyszeres idézőjeleket (=apostrophes), kétszeres idézőjeleket (=quotes), sortöréseket és egyéb különleges karaktereket fordított perjellel (\) választjuk el a parancssor többi elemétől.

Lássuk pl. a következő JS-kódot:

```
var txt="We are the so-called "Vikings" from the north.";
document.write(txt);
```

Mint korábban tapasztalhattuk, a JS értékfolyamok idézőjeltől idézőjelig tartanak; így a fenti szövegből csak a **We are the so-called** részlet fog megjelenni.

A probléma orvoslására a szövegen belüli idézőjelek elé fordított perjeleket írunk, miáltal azokat a böngésző különleges karakterként értelmezi:

```
var txt="We are the so-called \"Vikings\" from the north.";
document.write(txt);
```

A parancssor eredménye tehát a teljes szöveg lesz:
We are the so-called ”Vikings” from the north.

Az alábbi táblázatban öszefoglaltuk a szöveg-folyamokba (=text strings) perjelek után írható, különleges karakterré válható írásjeleket és jelentésüket:

Kód	Jelentés (karakter a megjelenített szövegben)
\'	Egyszeres idézőjel (’)
\"	Kétszeres idézőjel (”)
\\	Fordított perjel (\)
\n	Sortörés bekezdések között (=new line)
\r	Sortörés bekezdésen belül (=carriage return)
\t	Tabulátor
\b	Törlés (=backspace)
\f	Előtolás (=form feed, erdetileg a papír sornyi előtolása a mátrixnyomtatóban)

Különleges karakterek:

A JS szöveges értékeibe (”*Szöveg*”) írható különleges karakterek kódolása:

Egyszeres idézőjel (’): \’
Kétszeres idézőjel (”): \”
Fordított perjel(\): \\
Sortörés bekezdések közt: \n
Sortörés bekezdésen belül: \r
Tabulátor: \t
Törlés: \b
Előtolás: \f

XX. JavaScript – útmutató

Az alábbiakban összefoglaljuk a JavaScript írásakor szem előtt tartandó főbb alapismereteket.

A JavaScript esetfüggő!

Pl. a **myfunction** és a **myFunction** nevű függvény nem azonos egymással; ugyanígy a **myVar** változó sem egyezik meg a **myvar**-ral.

A JS tehát esetfüggő, így oda kell figyelnünk a kis- és nagybetűk következetes használatára a változók, objektumok és függvények definiálásakor és meghívásakor. Azonban, a HTML-lel és CSS-sel ellentétben, ezek nevei tartalmazhatnak nagybetűket.

Szóköz

A JavaScript (A HTML-hez és CSS-hez hasonlóan) összevonja a többszörös szóközöket; így azokat csak a kód olvashatóságának javítására érdemes alkalmaznunk, pl. a következő két rendelkezés ugyanazt jelenti:

```
name="Hege";
name = "Hege";
```

bár utóbbi olvashatóbb.

Kód-sorok tördelése

Mint láttuk, nem kell minden rendelkezést külön sorba írnunk, hogyha pontosvesszőkkel választjuk el őket.

A megfordított eset is lehetséges, azaz egyetlen rendelkezést több sorba is írhatunk. Mivel a sortörés (enter) a kódban egyúttal a rendelkezés végét jelentené, ebben az esetben fordított perjeleket kell tennünk a sorok végére, pl.:

```
document.write("Hello \
World!");
```

A kód-sort azonban csak az egyes alkotóelemeken belül szabad megtörni, azok határá nem; tehát az előbbi példához hasonló, de a függvény és annak paramétere közt megtört rendelkezés már hibás lenne:

```
document.write \
("Hello World!");
```

(A legutóbbi kód tehát hibás!)

JavaScript – útmutató (alapszabályok):

- **A JS esetfüggő, ügyeljünk a nagy- és kisbetűk következetes használatára!**
- **A változók nevei _-jellel vagy betűvel kezdődjenek!**
- **A JS-ben a szóközök összeolvadnak, azok csak az olvashatóság érdekében alkalmazhatók!**
- **Az egyes rendelkezések soron belül pontosvesszőkkel elválaszthatók, vagy enterrel új sorba elkülönítendők.**
- **Egyetlen rendelkezést több sorba tördelhetünk, a \ jel segítségével. A tördelés csak az elemeken belül történhet, pl.:**

```
document.write("Hello \
World!");
```

JavaScript
II. KÖNYV: OBJEKTUMOK

I. JavaScript objektumok – Bevezetés

A JavaScript egy ún. objektum-orientált programozási nyelv [=Object Oriented Programming language (OOP)]. Az OOP-nyelvek lehetővé teszik saját objektumok és változó-típusok definiálását.

Objektum-orientált programozás
A JS egy objektum-orientált programozási nyelv (OOP), azaz tetszőleges objektumok és változók definiálására ad lehetőséget.

Saját objektumok definiálásával azonban csak később foglalkozunk, a haladó szintű leírásban (III. könyv). Egyelőre a JS alapértelmezett objektum-típusaival és azok használatával ismerkedünk meg részletesebben.

Az objektum (=object) egy különleges státuszú adat, melyre jellemzők (=properties) és módszerek (=methods) vonatkoznak.

Jellemzők
A jellemzők az objektummal kapcsolatos értékek.

A következő példában a **length** (=hosszúság) jellemző segítségével kiíratjuk egy szöveg-folyam karaktereinek számát (beleértve a szóközöket is):

```
<script type="text/javascript">
var txt="Hello World!";
document.write(txt.length);
</script>
```

Először definiáljuk a **txt** változót és annak értékét, majd kiíratjuk a **txt** változó **length** jellemzőjét. Mivel a **Hello World!** felirat 12 karakterből áll, a weboldalon megjelenő szöveg: **12**.

Módszerek
A módszerek (=methods) az objektumokon vlgrehajtott akciók.

A következő példában az **str** objektumon (változón) végrehajtjuk a **toUpperCase()** módszert (függvényt), miáltal annak tartalma csupa nagybetűvel jelenik meg a weboldalon:

```
<script type="text/javascript">
var str="Hello world!";
document.write(str.toUpperCase());
</script>
```

Mint látjuk, először definiáljuk az **str** változót és annak értékét, majd kiíratjuk azt **toUpperCase()** módszer mellett:
HELLO WORLD!
eredménnyel.

Objektumok – bevezetés:
length (szöveghosszúság) módszer: .length
Pl.: var txt=”Hello World!”;
var len=(txt.length)=12

toUpperCase (nagybetűs) módszer: .toUpperCase()
Pl.: var str=”Hello World!”
document.write(str.toUpperCase()); = HELLO WORLD!

II. JavaScript – a string objektum

A JS **string** objektum szövegrészletek kezelésére való.

Néhány gyors példa

Az alábbiakban néhány példát találunk a **string** objektum alkalmazási lehetőségeire.

Első példánkban egy szöveg-részlet (=string) karakter-számát meghatározó JS-et látunk:

```
<html>
<body>

<script type="text/javascript">

var txt = "Hello World!";
document.write(txt.length);

</script>

</body>
</html>
```

Mint látjuk, ez megegyezik a bevezetőben látottal.

Második példánkban szövegrészletek formázását látjuk:

```
<html>
<body>

<script type="text/javascript">

var txt = "Hello World!";

document.write("<p>Big: " + txt.big() + "</p>");
document.write("<p>Small: " + txt.small() + "</p>");

document.write("<p>Bold: " + txt.bold() + "</p>");
document.write("<p>Italic: " + txt.italics() + "</p>");

document.write("<p>Fixed: " + txt.fixed() + "</p>");
document.write("<p>Strike: " + txt.strike() + "</p>");

document.write("<p>Fontcolor: " + txt.fontcolor("green") + "</p>");
document.write("<p>Fontsize: " + txt.fontsize(6) + "</p>");

document.write("<p>Subscript: " + txt.sub() + "</p>");
document.write("<p>Superscript: " + txt.sup() + "</p>");

document.write("<p>Link: " + txt.link("http://www.w3schools.com") +
"</p>");

document.write("<p>Blink: " + txt.blink() + " (does not work in IE,
Chrome, or Safari)</p>");

</script>

</body>
</html>
```

Először definiáljuk a **txt** változót (azaz a formázandó objektumot) és annak értékét.

Ezután az elérni kívánt hatásra utaló feliratokkal együtt kíratjuk az előbbi értéket, melynek megjelenését a következő jellemző-függvényekkel állatjuk be:

- **big()** (nagy betűméretű),
- **small()** (kis betűméretű),
- **bold()** (félkövér),
- **italics()** (dőlt),
- **fixed()** (Monotype gépi kód),
- **strike()** (áthúzott),
- **fontcolor("*szín*")** (bizonyos színű),
- **fontsize(6)** (pl. 6-os betűméretű),
- **sub()** (alsó indexbe írt),
- **sup()** (felső indexbe írt),
- **link()** (link-formázású),
- **blink()** (villogó).

Harmadik példánk szövegrészleteit (**string** objektumait) a **toLowerCase()** és **toUpperCase()** módszerekkel (függvényekkel) kis- ill. nagybetűkkel jeleníttetjük meg:

```
<html>
<body>

<script type="text/javascript">

var txt="Hello World!";
document.write(txt.toLowerCase() + "<br />");
document.write(txt.toUpperCase());

</script>

</body>
</html>
```

Mint látjuk, a **txt** változó értéke itt is **Hello World!**. Ezt az első parancs **toLowerCase()** függvénye csupa kis, a második parancs **toUpperCase()** függvénye pedig csupa nagybetűvel jeleníti meg:

hello world!
HELLO WORLD!

Harmadik példánkban a **match()** módszerrel (függvénnyel) keresünk szavakat egy objektum szövegében:

```
<html>
<body>

<script type="text/javascript">
var str="Hello world!";
document.write(str.match("world") + "<br />");
document.write(str.match("World") + "<br />");
document.write(str.match("worlld") + "<br />");
document.write(str.match("world!"));
</script>

</body>
</html>
```

Mint látjuk, itt az **str** változó értéke **Hello world!**. A parancssor további soraiban a **world**, **World**, **worlld** és **world!** betűösszetételekre keresünk a **match()** segítségével, aminek eredménye a következő weboldal:

world
null
null

world!
Amint látjuk, a második és harmadik keresés nem eredményezett találatot, így annak értéke **null**.

Némiképpen kibővíthetjük a keresést (a könnyebb kiértékelhetőség érdekében):

```
<html>
<body>

<script type="text/javascript">
var txt="Hello world!";
var find1="world";
var find2="vorld";
var find3="World";
var find4="orl";
var van="Megtalálható."
var nincs="Nincs találat."
</script>

<table style="border:3px solid red;">
<tr>
<th>A keresett szó:</th>
<th>A keresés eredménye:</th>
</tr>

<tr>
<th>
<script type="text/javascript">
document.write(find1)
</script>
</th>
<td>
<script type="text/javascript">
if(txt.match(find1)==find1)
{
document.write(van);
}
else
{
document.write(nincs);
}
</script>
</td>
</tr>

<tr>
<th>
<script type="text/javascript">
document.write(find2)
</script>
</th>
<td>
<script type="text/javascript">
if(txt.match(find2)==find2)
{
document.write(van);
}
```

```
else
{
document.write(nincs);
}
</script>
</td>
</tr>

<tr>
<th>
<script type="text/javascript">
document.write(find3)
</script>
</th>
<td>
<script type="text/javascript">
if(txt.match(find3)==find3)
{
document.write(van);
}
else
{
document.write(nincs);
}
</script>
</td>
</tr>

<tr>
<th>
<script type="text/javascript">
document.write(find4)
</script>
</th>
<td>
<script type="text/javascript">
if(txt.match(find4)==find4)
{
document.write(van);
}
else
{
document.write(nincs);
}
</script>
</td>
</tr>
</table>

</body>
</html>
```

Mint látjuk, itt az eredményeket a kereső-függvénnyel együtt egy HTML-táblázatba foglaltuk.

Mivel ugyanazt a keresési utasítást itt többször lefuttatjuk, célszerű ciklust alkalmazni:

```
<html>
```

```
<head>
    <title></title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <script type="text/javascript">
    var txt="Hello world!";

    var find=new Array();
    find[0]="world";
    find[1]="vorld";
    find[2]="World";
    find[3]="orl";

    var van="Megtalálható."
    var nincs="Nincs találat."
    </script>
</head>

<body>
    <table style="border:3px solid red;">
    <tr>
    <th>A keresett szó:</th>
    <th>A keresés eredménye:</th>
    </tr>

    <script type="text/javascript">
    for (x in find)
        {
        document.write("<tr><th>" + find[x] + "</th><td>");
        if(txt.match(find[x])==find[x])
            {
            document.write(van);
            }
            else
            {
            document.write(nincs);
            }
        document.write("</td></tr>");
        }
    </script>
    </table>
</body>
</html>
```

Az eredmény megegyezik az előző kódéval; azaz a szó megtalálhatóságát táblázatos formában íratjuk ki; a különbség csupán az, hogy a megismétlődő műveleteket egy **for** ciklussal végeztettük el, és nem a kód manuális megismétlésével.

Negyedik példánkban a **replace()** módszer segítségével egy szöveg-láncolat megegadott betűösszetételeit másra cseréljük ki:

```
<html>
<body>
<script type="text/javascript">

var str="Visit Microsoft!";
document.write(str.replace("Microsoft","W3Schools"));
```

```
</script>
</body>
</html>
```
Mint látjuk, az **str** változó eredeti, **Visit Microsoft!** értékéből a **Microsoft** betűösszetételt **W3Schools**-ra cseréltük, de ugyanígy kicserélhettük volna a szöveg **s** betűit **z**-re is:
```
<html>
<body>
<script type="text/javascript">

var str="Visit Microsoft!";
document.write(str.replace("s","z"));

</script>
</body>
</html>
```
Ekkor azonban az eredmény **Vizit Microsoft!**, azaz csupán az első **s**-t cseréltük le.
Ezt bizonyítja a következő kód is:
```
<html>
<body>
<script type="text/javascript">

var str="Visit Microsoft Microsoft!";
document.write(str.replace("Microsoft","ZULU"));

</script>
</body>
</html>
```
melynek eredménye: **Visit ZULU Microsoft!**; tehát csak az első **Microsoft**-ot cseréltük le.
Mindkettőt lecseréli azonban a következő kód:
```
<html>
<body>
<script type="text/javascript">

var str="Visit Microsoft Microsoft!";
str=str.replace("Microsoft","ZULU");
str=str.replace("Microsoft","ZULU");
document.write(str);

</script>
</body>
</html>
```
Mint látjuk, itt az **str** változó értékét kétszer is újradefiniáljuk, először az első, másodszor pedig a második **Microsoft** szót lecserélve. Így a végül kiírásra kerülő érték:
Visit ZULU ZULU!
A **replace()** függvény egyszeri végrehajtásával tehát csak egy (a legelőrébb álló) szót (betű-összetételt) cserélhetünk le a szöveg-stringből; ha a következőket is le akarjuk cserélni, akkor a parancsot meg kell ismételni.
Ezt megtehetjük egy **while** ciklussal. A kódot mindaddig ismételtetjük, míg a **match()** módszerrel megvizsgálva az eredményt, annak értéke **null**-tól különbözne; azaz még volna benne lecserélendő elem.
Az ehhez szükséges kód:
```
<html>
<body>
<script type="text/javascript">
var str="Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut
nunc nunc, interdum accumsan, laoreet nec, ultrices eget, eros.
```

```
Phasellus laoreet, nibh ac lacinia rhoncus, magna turpis suscipit
lectus, in posuere neque leo at purus. Nullam felis velit, volutpat ut,
consectetuer id, placerat ac, est. Nunc vel pede eu felis facilisis
pharetra. Nam ac magna. Praesent ullamcorper facilisis massa. Vivamus
imperdiet, velit nec rutrum vestibulum, tortor turpis vestibulum velit,
rhoncus tincidunt metus arcu vel justo. Suspendisse ligula arcu,
dignissim at, sollicitudin ac, pellentesque suscipit, lectus. Vivamus
nibh. Suspendisse nec tellus. Fusce sed arcu a nibh consectetuer
fringilla. Pellentesque bibendum risus. Ut tincidunt, massa vitae
posuere tincidunt, turpis turpis sodales velit, ut congue felis nisl a
metus. Aenean ac quam. Praesent vestibulum, arcu sed laoreet viverra,
orci nisl faucibus urna, sit amet euismod ante velit in ipsum.";
document.write(str);
while (str.match("o")!==null)
    {
    str=str.replace("o","ZULU");
    }
document.write(str);
</script>
</body>
</html>
```

A böngésző a fenti oldal futtatásakor kétszer is kiírja a „Lorem ipsum”-szöveget; egyszer az eredeti alakban, másodszor (közvetlenül az első után) pedig úgy, hogy minden **o** betű helyén a **ZULU** betűcsoport áll.

Hatodik példánkban egy szöveg-stringbeli szó első előfordulási helyét íratjuk ki az **indexOf()** módszerrel:

```
<html>
<body>

<script type="text/javascript">
var str="Hello world!";
document.write(str.indexOf("d") + "<br />");
document.write(str.indexOf("WORLD") + "<br />");
document.write(str.indexOf("world"));
</script>

</body>
</html>
```

A JS futtatásának eredménye:

10
-1
6

Jelezve,hogy a **d** betű a **11**., a **world** szó pedig a **6**. betű-helyen található a szövegben, míg a **WORLD** szó nincs benne (helye ezért **-1**). Mint látjuk, azon szavaknak, melyek a szövegben a nem fordulnak elő, a hely-számuk **-1**; ezért az **indexOf()** függvényt a **match()**-hoz hasonlóan szavak jelenlétének felderítésére is használhatjuk.

String-objektum – kézikönyv

A **string** objektumokra vonatkozó összes jellemzőt és módszert (és rövid jellemzésüket, példákkal) megtaláljuk string-objektum – kézikönyvünkben.

A string objektum

A string **objektum**okat (pl. változók értékeként szereplő) szövegrészletek (**jellemző**kkel ás **módszer**ekkel való) kezeléséhez használjuk.

Pl. a **length** jellemző segítségével kiírathatjuk a stringben lévő karakterek számát:

```
var txt="Hello world!";
document.write(txt.length);
```

Az eredmény: **12**.

Hasonlóképpen a **toUpperCase()** módszerrel a string tartalmát csupa nagybetűvel írathatjuk ki:

```
var txt="Hello world!";
document.write(txt.toUpperCase());
```

Az eredmény: **HELLO WORLD!**.

Ugyanezt érthetjük el az alábbi kóddal is:

```
var txt=”Hello world!”;
txt=txt.toUpperCase();
document.write(txt);
```

A string objektum:

A szöveg-értékkel bíró objektumokat (pl. változókat) sringeknek nevezzük.

Definiálása:

var *változó-név*=”*Szöveg*”; vagy
var *változó-név*=new String(”*Szöveg*”);

Módszerek stringekre:

- **HTML-formázás**
  A változó-érték betűit formázó függvények (melyek – pl. kiíratáskor – a stringet a megfelelő HTML-tagbe ágyazva adják vissza), pl.:
  változó-név.bold()
  Típusai:
  - **anchor()** (könyvjelző)
  - **link()** (hiperlink)
  - **big()** (nagy betűméretű)
  - **small()** (kis betűméretű)
  - **toUpperCase()** (nagybetűs)
  - **toLowerCase()** (kisbetűs)
  - **bold()** (félkövér)
  - **italics()** (dőlt)
  - **fixed()** (monospace gépi kód)
  - **strike()** (áthúzott)
  - **fontcolor(”*szín*”)** (bizonyos színű)
  - **fontsize(6)** (pl. 6-os betűméretű)
  - **sub()** (alsó indexbe írt)
  - **sup()** (felső indexbe írt)
  - **link()** (link-formázású)
  - **blink()** (villogó).
- **Karakter-számláló**
  A változó-érték karaktereinek száma (szóközökkel és írásjelekkel együtt, pl.:
  változó-név.length
- **Szó/betűösszetétel-egyezés keresése**
  A stringet a **match()** módszerrel megvizsgálva, a kifejezés értéke találat esetén megegyezika keresett szóval, egyébként **null**:
  változó-név.match(”*keresett_szó*”)
- **Szó/betűösszetétel-hely keresése**
  A stringet a **search()** módszerrel megvizsgálva, a kifejezés értéke találat esetén megegyezika keresett szó helyszámával, egyébként -1. Az első betű helyszáma 0. Mondattan:
  változó-név.search(”*keresett_szó*”)

- **Szó/betűösszetétel cseréje**
  A **replace()** módszerrel a string egy megadott szavának első előfordulását lecserélhetjük egy másik szóra. Az összes megadott szó vagy betű(összetétel) lecseréléséhez a **replace()** módszert ismételgetni kell. Mondattana:
  változó-név.replace(”*lecserélendő_szó*”,”*beírandó_szó*”)
- **Első előfordulási hely megállapítása**
  Az **indexOf()** függvény felveszi egy keresett szó vagy betű első előfordulásának sorszámát. Az első betű sorszáma 0. Ha a kifejezés nem található, értéke -1. Mondattana:
  változó-név.indexOf(”*keresett_szó*”)
- **Utolsó előfordulási hely megállapítása**
  Az **lastIndexOf()** függvény felveszi egy keresett szó vagy betű utoló előfordulásának sorszámát. Az első betű sorszáma 0. Ha a kifejezés nem található, értéke -1. Mondattana:
  változó-név.lastIndexOf(”*keresett_szó*”)
- **Adott helyszámú betű másolása**
  A **charAt()** módszer értékként felveszi a paraméterébe írt helyszámú betűt. Az első betű helyszáma 0. Mondattan:
  változó-név.charCodeAt(*0*)
- **Adott helyszámú betű Unicode-másolása**
  A **charAt()** módszer értékként felveszi a paraméterébe írt helyszámú betű Unicode-értékét. Az első betű helyszáma 0. Mondattan:
  változó-név.charCodeAt(*0*)
- **Stringek egyesítése**
  A **concat()** módszer paraméterébe írt stringek tartalmának egymásutánját hozzáfűzi a hivatkozott kezdő-stringhez, és ezt veszi fel értékül. Pl.:
  változó_1.concat(”*változó_2*”,”*változó_3*”,…,*változó_x*”)
- **Unicode egyesítése stringgé**
  A **String.fromCharCode()** módszer paraméterébe írt Unicode-számokka jelölt betűket string-ként adja vissza, pl.:
  változó-név=String.fromCharCode(65,66,67);
- **Megadott kezdő- és végértékű szövegrészlet kimásolása (stringből)**
  A **slice()** módszer a string kijelölt részletét veszi fel értékként. A kijeölést az első (kezdő)értéknél kezdjük, és a második (opcionális, záró)értéknél fejezzük be. Az első betű értéke 0. A kijelölő-értékek negatívak is lehetnek, ekkor a string végétől számítanak. Pl.:
  változó.slice(*kezdőérték*,*végérték*).
- **Megadott (pozitív) kezdő- és végértékű szövegrészlet kimásolása (stringből)**
  A **substring()** módszer gyakorlatilag megegyezik a **slice()**-szal, csak abban különbözik, hogy az index-értékek nem lehetnek negatívak! Mondattan:
  változó.substring(*kezdőérték*,*végérték*).
- **Megadott hosszúságú szövegrészlet kimásolása (stringből)**
  A **substr()** módszer a string egy kezdőértéktől a végéig vagy megadott karakter-hosszig tartó részletét adja vissza. A kijelölést a kezdőértéknél kezdjük, és a második paraméternek megfelelő számú karakteren (ill. a teljes hátralévő részen át) folytatjuk. Az első paraméter negatív is lehet, ekkor a string végétől számít. Mondattan:
  változó.substr(*kezdőérték*,*hossz*).
- **Teljes érték visszaadása**
  A **valueOf()** módszer a string teljes szövegét visszadja. E módszert nem kell külön kiírnunk; a változó-név megadásakor a böngésző a háttérben futtatva, e függvénnyel véteti fel annak értékét. Mondattana:
  változó-név.valueOf()
- **String átalakítása array-jé**
  A **split()** módszerrel egy stringet a kijelölt karaktereknél feldarabolva és vesszőkkel elválasztva megadott tag-számú array-jé (érték-listává) alakítunk. Hogyha a képezhető array-elemek száma nagyobb, mint a képzendő, akkor az array csak az első tagokat fogja tartalmazni, míg meg nem telik. Mondattan:

változó.split(”*kijelölt_elválasztó-karakter*”,”*képzendő_array-elemek_száma*”)

Mintapélda:

Egy változó nagybetűs kiíratása kétféle módszerrel:

1.:
```
var txt="Hello world!";
document.write(txt.toUpperCase());
```

2.:
```
var txt=”Hello world!”;
txt=txt.toUpperCase();
document.write(txt);
```

III. JavaScript – a date objektum

A JS **date** objektum dátumok és időpontok kezelésére való.

Néhány gyors példa

Első példánkban a **Date()** módszer segítségével kiíratjuk az aktuális (gépi) dátumot:

```
<html>
<body>
<script type="text/javascript">

var d=new Date();
document.write(d);

</script>
</body>
</html>
```

Azaz a **d** változó értékéül a **Date()** függvény aktuális értékét választjuk, majd kiíratjuk azt:
Tue Jan 18 14:34:39 UTC+0100 2011

Második példánkban a **getTime()** módszerrel kiszámoljuk, mennyi idő telt el 1970.I.1. óta:

```
<html>
<body>
<script type="text/javascript">

var d=new Date();
document.write(d.getTime() + " milliseconds since 1970/01/01");

</script>
</body>
</html>
```

Először ismét az aktuálisidő-értékkel definiáljuk a **d** változót, majd végrehajtjuk a következő rendelkezést; így milliszekundumokban megkapjuk az eltelt időt. Mint látuk, a kezdő-dátumot ill. mértékegységet nem tudjuk befolyásolni (hiszen azt csak statikus szövegként adtuk meg), az óra mindig 1970.I.1-jétől milliszekundumokban adja meg az időt.

Harmadik példánkban a **setFullYear()** módszerrel beállítjuk a böngésző belső órájától kapott dátumot megváltoztatjuk:

```
<html>
<body>
<script type="text/javascript">

var d = new Date();
d.setFullYear(1992,10,3);
document.write(d);

</script>
</body>
</html>
```

Először a **d** változóhoz rendeljük a belső óra aktuális értékét, majd a dátumot átállítjuk (2010.II.18-áról) **1992.X.3**-ára. Az így megjelenő óra-érték:
Tue Nov 3 14:47:57 UTC+0100 1992
Mint látjuk, az időjelzés többi eleme nem változott.

Negyedik példánkban az eredeti, UTC-nek (=Coordinated Universal Time=Egyezményes Világidő) megfelelő idő-formátumot olvashatóbb szöveg-stringgé alakítjuk:

```
<html>
```

```
<body>
<script type="text/javascript">

var d=new Date();
document.write("Original form: ");
document.write(d + "<br />");
document.write("To string (universal time): ");
document.write(d.toUTCString());

</script>
</body>
</html>
```

Mint látjuk, először kiíratjuk (az **Oroginal form:** szöveg mögé) a dátumot (vagyis a **d** változó értékét) az eredeti UTC-alakban, majd újra kiíratjuk, de már **d.toUTCString()** módszerrel. A két felirat ennek megfelelően:

Original form: Tue Jan 18 15:53:59 UTC+0100 2011
To string (universal time): Tue, 18 Jan 2011 14:53:59 UTC

Ötödik példánkban a **getDay()** módszerrel kiírjuk a hét napjának nevét:

```
<html>
<body>
<script type="text/javascript">

var d=new Date();
var weekday=new Array(7);
weekday[0]="Sunday";
weekday[1]="Monday";
weekday[2]="Tuesday";
weekday[3]="Wednesday";
weekday[4]="Thursday";
weekday[5]="Friday";
weekday[6]="Saturday";

document.write("d.getDay()= "+d.getDay()+"<br />");
document.write("Today is " + weekday[d.getDay()]);

</script>
</body>
</html>
```

Az első rendelkezés értelmében a **d** változóhoz rendeljük az UTC-időértéket; ezután a **weekday** változó értékéül megadjuk a héttagú **new Array(7)**-et, ahol a számmal jelölt array-tagokhoz, mint változókhoz, egyenként hozzárendeljük a nap-neveket.
Végül kiíratjuk a napnév azonosítására szolgáló **getDay()** függvény értékét, melyet a **d** változó (aktuális) értékéből képezünk. Ez keddi napon **2**.
Legvégül pedig kiíratjuk a napnevet is, ti. kiválasztjuk a **weekday** arrayből a **d.getDay()** függvény értékének (**2**) megfelelő **weekday[2]** változót, és kiíratjuk annak értékét (**Tuesday**).
A megjelenő felirat tehát:

d.getDay()= 2
Today is Tuesday

Hatodik példánkban egy digitális órát készítünk weboldalunkra:

```
<html>
<head>
<script type="text/javascript">
function startTime()
```

```
{
var today=new Date();
var h=today.getHours();
var m=today.getMinutes();
var s=today.getSeconds();
// add a zero in front of numbers<10
m=checkTime(m);
s=checkTime(s);
document.getElementById('txt').innerHTML=h+":"+m+":"+s;
t=setTimeout('startTime()',500);
}

function checkTime(i)
{
if (i<10)
  {
  i="0" + i;
  }
return i;
}
</script>
</head>

<body onload="startTime()">
<div id="txt"></div>
</body>
</html>
```

Mint látjuk, az oldal betöltésekor elindítja a **startTime()** függvényt. Ez először a **today** változó értékéül adja a belső órajelet, majd annak részeiből definiálja a **h** óra-, a **m** perc- és az **s** másodperc-értéket. Ezután az **m** és **s** értékekre lefuttatja a **checkTime(i)** függvényt, azaz az **m** és **s** értéket behelyettesíti az **i** belső változó helyére, és amennyiben **10**-nél kisebbnek bizonyul, egy nullát beilleszt eléje, és úgy adja vissza. Ennek jelentősége, hogy az UTC-időnél a 10 alatti perc és másodperc-értékek egy számjegyben jelennek meg.
Ezután kettőspontokkal elválasztva kiíratjuk a **h**, **m** és **s** értékeket a **txt **-be.
Legvégül beállítjuk, hogy a **startTime()** függvény értékét (azaz a megjelenített idő-értéket) a program **500** milliszekundumonként (azaz fél másodpercenként) számítsa ki. Így a kijelzés legfeljebb ±0,5s-ot késhet vagy siethet (az órajelhez képest). Hogyha a legutóbbi értéket **500** helyett **5000**-nek vesszük, akkor az óra-kijelzés csak öt mésodpercenként fog frissülni, tehát egy helyett öt-öt másodperces váltásokkal fog ketyegni (az előzőhöz képest, amikor másodpercentként kétszer is frissült).

Date-objektum – kézikönyv
A **date** objektumokra vonatkozó összes jellemzőt és módszert (és rövid jellemzésüket, példákkal) megtaláljuk date-objektum – kézikönyvünkben.

Date objektumok beállítása
A **date** objektumot dátumok és időpontok kezelésére használjuk; a **Date()** konstruktor (=constructor) révén.

A dátumot négyféleképpen fejezhetjük ki a **Date()** konstruktorral (példákat lásd alább):

```
new Date() // current date and time
new Date(milliseconds) //milliseconds since 1970/01/01
new Date(dateString)
new Date(year, month, day, hours, minutes, seconds, milliseconds)
```

A legtöbb paraméter opcionális; hogyha nem töltjük ki őket, a program nullának veszi az értéküket.

A **date** objektum beállítása után módszereket hajthatunk végre rajta. A legtöbb módszer a helyi vagy globális (UTC/GMT) idő-adat lekérdezésére vagy módosítására szolgál.

Az összes dátumot (pontosabban időtartamot) az 1970. január 1. 00:00:00 UTC-időponthoz képest, milliszekundunokban adják meg, egy napot előírásszerűen 86.400.000 milliszekundumosnak tekintve. Mivel egy nap hossza ennél valamivel hosszabb (lásd: szökőnapok) azóta több ún. másodperc-(előre)ugrást kellett végrehajtani az UTC-időben.

Néhány példa dátum-beállításra:

```
var today = new Date()
var d1 = new Date("October 13, 1975 11:13:00")
var d2 = new Date(79,5,24)
var d3 = new Date(79,5,24,11,33,0)
```

Az így beállított dátumok megjelenítésére alkalmas kód:

```
<html>
<head>
<script type="text/javascript">
var today = new Date()
var d1 = new Date("October 13, 1975 11:13:00")
var d2 = new Date(79,5,24)
var d3 = new Date(79,5,24,11,33,0)
</script>
</head>

<body>
<script type="text/javascript">
document.write(today+"<br />");
document.write(d1+"<br />");
document.write(d2+"<br />");
document.write(d3);
</script>
</body>
</html>
```

A kód eredménye sorrendben:
Tue Jan 18 18:45:15 UTC+0100 2011
Mon Oct 13 11:13:00 UTC+0200 1975
Sun Jun 24 00:00:00 UTC+0200 1979
Sun Jun 24 11:33:00 UTC+0200 1979

Dátumok beállítása
A **Date()** objektumra vonatkozó módszerekkel könnyev átállíthatjuk a dátum-értéket.

Az alábbi példában a **Date()** objektumnak egy bizonyos értéket adunk:

```
var myDate=new Date();
myDate.setFullYear(2010,0,14);
```

Hogyha lefuttatjuk a következő parancssort:

```
<html>
<head>
<script type="text/javascript">
var myDate=new Date();
myDate.setFullYear(2010,0,14);
```

```
</script>
</head>

<body onload="startTime()">
<script type="text/javascript">
document.write(myDate);
</script>
</body>
</html>
```

látjuk, a generált felirat
Thu Jan 14 18:50:43 UTC+0100 2010
lesz. Ennek megfelelően látható, hogy a **setFullYear** módszerrel valóban csak a beírt értékeket (év/hó/nap) változtattuk meg, a többi továbbra is az órajelből adódik.

Egy másik példaként a jelenhez képest öt nappal előreállítjuk a dátumot:

```
var myDate=new Date();
myDate.setDate(myDate.getDate()+5);
```

E parancssorban először beállítjuk a **myDate** változót, melynek értékéül a jelenlegi gépi dátumot választjuk.
Ezután a **myDate** értékéül szolgáló dátumot átállítuk, a **setDate()** függvénnyel. Enknek értéke **myDate.getDate()+5**. A **getDate()** függvény visszaadja (=returns) a **myDate** értékből a hónap napját (jelen esetben 18), majd ehhez hozzáadunk **5**-öt. Ennek eredménye 23; azaz a **myDate**-re vonatkozó **setDate()** függvény ezt a nap-számot állítja be a változóban. Hogyha a hozzáadással a hónap értéke is megváltozna, azt a program automatikusan figyelembe veszi.
Hogyha a **myDate** változót a fentiekkel analóg módon kiíratjuk, a következőt kapjuk:
Sun Jan 23 19:00:14 UTC+0100 2011
A jelenlegi idő pedig:
Tue Jan 18 19:00:39 UTC+0100 2011
tehát számításunk helyes volt.

A fenti második rendelkezési ki is tagolhatjuk:

```
<html>
<head>
<script type="text/javascript">
var myDate=new Date();
var day=myDate.getDate()+20
myDate.setDate(day);
</script>
</head>

<body onload="startTime()">
<script type="text/javascript">
document.write(myDate + "<br >");
document.write(myDate.getDate() + "<br >");
document.write(new Date() + "<br >");
document.write(new Date().getDate());
</script>
</body>
</html>
```

ekkor a **setDate()** függvény paraméterét külön sorban számítjuk ki, és egy változó (**day**) formájában helyettesítjük be az összefüggésbe.
Végül kiíratjuk a **myDate** változóra vonatkozó **getDate()** módszer értékét is, mely azonban ekkor, azaz a **myDate.setDate()** rendelkezés után 18-ról (február) **7**-re változott, ami igazolja, hogy a **setDate()** függvény a kisebb időegységektől a nagyobbakban okozott változást is figyelembe veszi.
Ezutánl a **new Date()** függvény aktuális értékét, majd pedig külön a nap-értékét íratjuk ki. Mint látjuk, soha

nem a böngésző belső órájának állását válzoztatjuk meg, hanem egy annak aktuális értékét reprezentáló változó értékét, azaz a parancssor eredménye a következő szöveg:
Mon Feb 7 19:14:38 UTC+0100 2011
7
Tue Jan 18 19:14:38 UTC+0100 2011
18

Dátumok összehasonlítása
A **Date** objektumot dátumok összehasonlítására is alkalmazhatjuk.
Következő példánkban a mai dátumot hasonlítjuk össze **2010.I.14**-ével:

```
var myDate=new Date();
myDate.setFullYear(2010,0,14);
var today = new Date();

if (myDate>today)
  {
  alert("Today is before 14th January 2010");
  }
else
  {
  alert("Today is after 14th January 2010");
  }
```

Mint látjuk, a jelenlegi dátumot a **today** és a **myDate** változó egyaránt értékként tartalmazza, utóbbit azonban az év/hó/nap tekintetében átírjuk 2010.I.14-ére.
Mivel most 2011.I.18. van, a **myDate>today** feltétel (függvény vagy módszer-paraméter) logikai értkée HAMIS, így az **else** feltételes rendelkezés lép életbe, vagyis a megjelenő figyelmeztető-ablak felirata:
Today is after 14th January 2010

A date objektum:
A JS **date** objektum dátumok és időpontok kezelésére való.
Definiálása:
new Date() //Az aktuális dátum és idő
new Date(*ezredmásodperc*) //Az 1970.01.01. óta eltelt ezredmésodpercek (ms-ok) száma.
new Date(*dátum-string*)
new Date(*év, hó, nap, óra, perc, másodperc, ezredmásodperc*)
[A legtöbb paraméter opcionális; hogyha nem töltjük ki őket, a program nullának veszi az értéküket.]

Néhány példa dátum-beállításra:
var *today* = new Date()
var *d1* = new Date("October 13, 1975 11:13:00")
var *d2* = new Date(79,5,24)
var *d3* = new Date(79,5,24,11,33,0)

Módszerek dátumokra:
- **Helyi idő visszaadása:**
  - **A hónap napjának visszaadása**
    A **getDate()** módszer a hónap napját (1-31) veszi fel értékül. Mondattana:
    változó-név.getDate()
  - **A hét napjának visszaadása**
    A **getDay()** módszer a hét napját (0-6) veszi fel értékül. A vasárnap 0-nak, a hétfő 1-nek számít stb.. Mondattana:
    változó-név.getDay()
  - **Négyjegyű évszám visszaadása**
    A **getFullYear()** módszer az évszám négyjegyű alakját (pl. 2011) veszi fel értékül.

Mondattana:
változó-név.getFullYear()

- **Az órák visszaadása**
  A **getHours()** módszer a dátum óra-számát (0-23) veszi fel értékül. Mondattana:
  változó-név.getHours()
- **A milliszekundumok visszaadása**
  A **getMilliseconds()** módszer a dátum milliszekundum-számát (0-999) veszi fel értékül. Mondattana:
  változó-név.getMilliseconds()
- **A percek visszaadása**
  A **getMinutes()** módszer a dátum perc-számát (0-59) veszi fel értékül. Mondattana:
  változó-név.getMinutes()
- **A hónapok visszaadása**
  A **getMonth()** módszer a hónap-számot (0-11) veszi fel értékül. A Január száma 0, a Februáré 1 stb.. Mondattana:
  változó-név.getMonth()
- **A másodpercek visszaadása**
  A **getSeconds()** módszer a dátum másodperc-számát (0-59) veszi fel értékül. Mondattana:
  változó-név.getSeconds()
- **A világidő visszaadása milliszekundumokban**
  A **getTime()** módszer az 1970.01.01. óta eltelt milliszekundumok számát (pl. 1296557475420) veszi fel értékül. Mondattana:
  változó-név.getTime()

- **Helyi idő átállítása:**
  - **A hónap napjának átállítása**
    A **setDate()** módszer a paraméterébe írt nap-számot (1-31) helyi adatként értelmezve átállítja a dátumot. Mondattana:
    változó-név.setDate(”*31*”)
  - **Négyjegyű évszám átállítása**
    A **setFullYear()** módszer a paraméterébe írt négyjegyű évszámot (pl. 2011) helyi adatként értelmezve átállítja a dátumot. Mondattana:
    változó-név.setFullYear(”*2011*”)
  - **Az órák átállítása**
    A **setHours()** módszer a paraméterébe írt óra-számot (0-23) helyi adatként értelmezve átállítja a dátumot. Mondattana:
    változó-név.getHours(”*23*”)
  - **A milliszekundumok átállítása**
    A **setMilliseconds()** módszer a paraméterébe írt milliszekundum-számot (0-999) helyi adatként értelmezve átállítja a dátumot. Mondattana:
    változó-név.setMilliseconds(”*999*”)
  - **A percek átállítása**
    A **setMinutes()** módszer a paraméterébe írt perc-számot (0-59) helyi adatként értelmezve átállítja a dátumot. Mondattana:
    változó-név.setMinutes(”*59*”)
  - **A hónapok átállítása**
    A **setMonth()** módszer a paraméterébe írt hónap-számot (0-11) helyi adatként értelmezve átállítja a dátumot. A Január száma 0, a Februáré 1 stb.. Mondattana:
    változó-név.setMonth(”*11*”)
  - **A másodpercek átállítása**
    A **setSeconds()** módszer a paraméterébe írt másodperc-számot (0-59) helyi adatként értelmezve átállítja a dátumot. Mondattana:
    változó-név.setSeconds(”*59*”)
  - **UTC-re vonatkoztatott idő megadása milliszekundumokban**
    A **setTime()** módszerrel bármely dázumot megadhatunk, hogyha a paraméterébe az

1970.01.01. és a dátum közti milliszekundumok pozitív vagy negatív számát (pl. -1296557475420) beírjuk. Mondattana:
változó-név.setTime("*időtartam_milliszekundumokban*")

- **UTC-idő visszaadása:**
  - **A hónap napjának UTC-re vonatkoztatott visszaadása**
    A **getUTCDate()** módszer a változó értékét helyi dátumnak tekintve visszaadja az adott időponthoz tartozó UTC nap-számát (1-31). Azaz a helyi idő alapján kiszámítja az UTC-időt és az ahhoz tartozó napot adja meg. Mondattana:
    változó-név.getUTCDate()
  - **A hét napjának UTC-re vonatkoztatott visszaadása**
    A **getUTCDay()** módszer a helyinek tekintett dátum-érték alapján számított UTC hét-napját (0-6) veszi fel értékül. A vasárnap 0-nak, a hétfő 1-nek számít stb.. Mondattana:
    változó-név.getUTCDay()
  - **Négyjegyű évszám visszaadása UTC-re vonatkoztatva**
    A **getUTCFullYear()** módszer a helyinek tekintett dátum-érték alapján számított UTC-évszám négyjegyű alakját (pl. 2011) veszi fel értékül. Mondattana:
    változó-név.getUTCFullYear()
  - **Az órák UTC-re vonatkoztatott visszaadása**
    A **getUTCHours()** módszer a helyinek tekintett dátum-érték alapján számított UTC-dátum óra-számát (0-23) veszi fel értékül. Mondattana:
    változó-név.getUTCHours()
  - **A milliszekundumok UTC-re vonatkoztatott visszaadása**
    A **getUTCMilliseconds()** módszer a helyinek tekintett dátum-érték alapján számított UTC-dátum milliszekundum-számát (0-999) veszi fel értékül. Mondattana:
    változó-név.getUTCMilliseconds()
  - **A percek UTC-re vonatkoztatott visszaadása**
    A **getUTCMinutes()** módszer a helyinek tekintett dátum-érték alapján számított UTC-dátum perc-számát (0-59) veszi fel értékül. Mondattana:
    változó-név.getUTCMinutes()
  - **A hónapok UTC-re vonatkozatott visszaadása**
    A **getUTCMonth()** módszer a helyinek tekintett dátum-érték alapján számított UTC hónap-számát (0-11) veszi fel értékül. A Január száma 0, a Februáré 1 stb.. Mondattana:
    változó-név.getUTCMonth()
  - **A másodpercek UTC-re vonatkoztatott visszaadása**
    A **getUTCSeconds()** módszer a helyinek tekintett dátum-érték alapján számított UTC-dátum másodperc-számát (0-59) veszi fel értékül. Mondattana:
    változó-név.getUTCSeconds()
- **UTC-idő átállítása:**
  - **A hónap napjának UTC-átállítása**
    A **setUTCDate()** módszer a paraméterébe írt nap-számot (1-31) UTC adatként értelmezve átállítja a dátumot. Mondattana:
    változó-név.setUTCDate("*31*")
  - **Négyjegyű évszám UTC-átállítása**
    A **setUTCFullYear()** módszer a paraméterébe írt négyjegyű évszámot (pl. 2011) UTC adatként értelmezve átállítja a dátumot. Mondattana:
    változó-név.setUTCFullYear("*2011*")
  - **Az órák UTC-átállítása**
    A **setUTCHours()** módszer a paraméterébe írt óra-számot (0-23) UTC adatként értelmezve átállítja a dátumot. Mondattana:
    változó-név.getUTCHours("*23*")
  - **A milliszekundumok UTC-átállítása**
    A **setUTCMilliseconds()** módszer a paraméterébe írt milliszekundum-számot (0-999) UTC adatként értelmezve átállítja a dátumot. Mondattana:
    változó-név.setUTCMilliseconds("*999*")

- **A percek UTC-átállítása**
  A **setUTCMinutes()** módszer a paraméterébe írt perc-számot (0-59) UTC adatként értelmezve átállítja a dátumot. Mondattana:
  változó-név.setUTCMinutes(”*59*”)
- **A hónapok UTC-átállítása**
  A **setUTCMonth()** módszer a paraméterébe írt hónap-számot (0-11) UTC adatként értelmezve átállítja a dátumot. A Január száma 0, a Februáré 1 stb.. Mondattana:
  változó-név.setUTCMonth(”*11*”)
- **A másodpercek UTC-átállítása**
  A **setUTCSeconds()** módszer a paraméterébe írt másodperc-számot (0-59) UTC adatként értelmezve átállítja a dátumot. Mondattana:
  változó-név.setUTCSeconds(”*59*”)

- **Dátumok kiíratása:**
  - **Szabványos dátum-szöveg kiíratása**
    A **toDateString()** módszerrel az UTC szerinti gépi dátumot olvasható (pl. Mon Jan 01 2011) formátumú string-gé alakíthatjuk. Mondattana:
    változó-név.toDateString()
  - **Helyi formátumú dátum-szöveg kiíratása**
    A **toLocaleDateString()** módszerrel az UTC szerinti gépi dátumot olvasható (pl. 2011. január 1.), helyi (magyar) formátumú string-gé alakíthatjuk. Mondattana:
    változó-név.toLocaleDateString()
  - **Helyi formátumú teljes dátum kiíratása**
    A **toLocaleString()** módszerrel az UTC szerinti gépi dátum- és órajelzést olvasható (pl. 2011. január 1. 17:55:55), helyi (magyar) formátumú string-gé alakíthatjuk. Mondattana:
    változó-név.toLocaleString()
- **Óra-állások kiíratása:**
  - **Szabványos óra-szöveg kiíratása**
    A **toTimeString()** módszerrel az UTC szerinti gépi órajelzést olvasható (pl. 17:55:55), nemzetközi (angol) formátumú string-gé alakíthatjuk. Mondattana:
    változó-név.toTimeString()
  - **Helyi formátumú óra-szöveg kiíratása**
    A **toLocaleTimeString()** módszerrel az UTC szerinti gépi órajelzést olvasható (pl. 17:55:55), helyi (magyar) formátumú string-gé alakíthatjuk. Mondattana:
    változó-név.toLocaleTimeString()
- **Formátum-átalakítások:**
  - **Date objektum stringgé alakítása**
    A **toString()** módszerrel a dátum-értékű változókat (és általában a date- és egyéb, nem szöveg-értékű objektumokat) stringgé alakíthatjuk. Mondattana:
    változó-név.toString()
  - **Date objektum stringgé alakítása az UTC szerint**
    A **toUTCString()** módszerrel a helyinek tekintett dátum-értékű változókat (és általában a date- és egyéb, nem szöveg-értékű objektumokat) stringgé alakíthatjuk, a függvény azonban a helyi időből számítható UTC-t veszi fel. Mondattana:
    változó-név.toUTCString()
  - **Hagyományos dátum átalakítása UTC-vé**
    Az **UTC()** függvény a paraméterébe írt dátum-elemek (az első hámon kötelező, pl. 2011,1,1,17,55,55,55,555) által megadott dátum és az UTC alap-dátuma (1970.01.01.) közt eltelt milliszekundumokat veszi fel értékként. Mondattana:
    Date.UTC(*év,hónap,nap,óra,perc,másodperc,ezredmásodperc*)
  - **Date-string átalakítása UTC-vé**
    A **parse()** függvény a paraméteréül megadott dátum-string (date-**string**!) és az UTC alap-dátuma (1970.01.01.) közt eltelt milliszekundumokat veszi fel értékként. Mondattana:
    Date.parse(”*dátum vagy dátum-változónév*”)
- **Egyéb módszerek:**

- **A GMT-időeltolódás visszaadása percekben**
  A **getTimezoneOffset()** módszer a Greenwich-i középidőhöz (GMT = Greenwich Mean Time) képesti időeltolódást adja meg percekben. Magyarország a GMT+1 időzónában van, azaz itt egy órával később van Greenwich-hez képest. Itt a függvény értéke -60, jelezve, hogy az aktuális időből 60 percet kell levonni, hogy megkapjuk a GMT-t. Mondattana:
  változó-név.getTimezoneOffset()
- **Teljes érték visszaadása**
  A **valueOf()** módszer a dátum [**getTime()** függvény] teljes értékét visszadja (ami tehát az 1970.01.01. óta eltelt milliszekundumokat jelenti). E módszert nem kell külön kiírnunk; a változó-név megadásakor a böngésző a háttérben futtatva, e függvénnyel véteti fel annak értékét. Mondattana:
  változó-név.valueOf()

Mintapéldák:

A gépi dátum módosítása:

```
var d = new Date();
d.setFullYear(1992,10,3);
document.write(d);
```

Specifikus dátum nap-számának kiíratása:

```
var d = new Date("July 21, 1983 01:15:00");
document.write(d.getDate());
```

A jelenlegi hónap nevének kiíratása:

```
var d=new Date();
var month=new Array(12);
month[0]="January";
month[1]="February";
month[2]="March";
month[3]="April";
month[4]="May";
month[5]="June";
month[6]="July";
month[7]="August";
month[8]="September";
month[9]="October";
month[10]="November";
month[11]="December";
document.write("The current month is " + month[d.getMonth()]);
```

Az 1970.01.01. óta eltelt évek számának kiíratása:

```
var minutes=1000*60;
var hours=minutes*60;
var days=hours*24;
var years=days*365;
var d=new Date();
var t=d.getTime();
var y=t/years;
document.write("It's been " + Math.round(y) + " years since 1970/01/01!");
```

A felhasználó időzónájának szöveges kiíratása:

```
var d = new Date()
var gmtHours = -d.getTimezoneOffset()/60;
```

```
document.write("The local time zone is: GMT " + gmtHours);
```

Digitális óra kiíratása:

```
function startTime()
{
var today=new Date();
var h=today.getHours();
var m=today.getMinutes();
var s=today.getSeconds();
// add a zero in front of numbers<10
m=checkTime(m);
s=checkTime(s);
document.getElementById('txt').innerHTML=h+":"+m+":"+s;
t=setTimeout('startTime()',500);
}

function checkTime(i)
{
if (i<10)
  {
  i="0" + i;
  }
return i;
}

<body onload="startTime()">
<div id="txt"></div>
```

Dátumok összehasonlítása:

```
var myDate=new Date();
myDate.setFullYear(2010,0,14);
var today = new Date();

if (myDate>today)
  {
  alert("Today is before 14th January 2010");
  }
else
  {
  alert("Today is after 14th January 2010");
  }
```

IV. JavaScript – az array (tömb) objektum

Az array (=tömb) objektum egy változó több lehetséges értékét tartalmazó hierarchikus lista.

Néhány egyszerű példa

Első példánkban egy változó hoz array-ként több lehetséges értéket rendelünk, majd kiíratjuk őket:

```
<html>
<body>

<script type="text/javascript">
var i;
var mycars = new Array();
mycars[0] = "Saab";
mycars[1] = "Volvo";
mycars[2] = "BMW";

for (i=0;i<mycars.length;i++)
{
document.write(mycars[i] + "<br />");
}

document.write(mycars.length);
</script>

</body>
</html>
```

Először definiáljuk az **i** változót, bár erre nem volna szükség, hiszen később, a (kezdő)érték hozzárendelése is definiálásnak számít.
Ezután a **mycars** változóhoz hozzárendelünk egy új array-t (**new Array()**), melynek három, 0-tól 2-ig számozott eleme van; ezek a **mycars** véltozó lehetséges értékei.
Ezután egy **for** ciklussal kiíratjuk a **mycars** összes lehetségs értékét; azaz, mint a paraméterekből látszik, az **i** változóhoz (ami itt helyi változó, és mivel kívül másutt nem használjuk, felesleges is volt általános váéltozóként definiálni) hozzárendeljük a **0** kezdőértéket. A **for** ciklus mindaddig fut, míg **i** kisebb, mint a **mycars.length** kifejezés értéke, azaz a **mycars** változóhoz tartozó array hossza (listalemeinek száma); és **i** értéke ciklusonként eggyel növekszik (**i++**).
Mint látjuk, a ciklus egyetlen kiíró lépésből áll, mikor az **i** aktuális értékének megfelelő sorszámú **mycars**-értéket íratjuk ki az array-ből.
Végül kiíratjuk a **mycars.length** feltétel értékét is, hogy a **for** második paraméterét jobban értelmezhessük.
A megjelenő felirat:

Saab
Volvo
BMW
3

Mint látjuk, **mycars.length**=3, azaz a **mycars** array három elemből áll, és a **for** ciklus addig fut, míg **i<3**, vagyis **i** értéke 0-2 közt változván, valóban három elemet íratunk ki.
Hogyha a paramétert átírjuk:

```
for (i=0;i<10;i++)
```

akkor a kimenet:

Saab
Volvo
BMW
undefined
undefined
undefined
undefined
undefined

undefined
undefined
3
jelezve, hogy az **i=2** után következő esetekhez nincsenek array-elemek.

Második példánkban a **for in** ciklus segítségével íratjuk ki ugyanezen array értékeit:

```
<html>
<body>
<script type="text/javascript">
/*var x;*/
var mycars = new Array();
mycars[0] = "Saab";
mycars[1] = "Volvo";
mycars[2] = "BMW";

for (x in mycars)
{
document.write(mycars[x] + "<br />");
}
</script>
</body>
</html>
```

Mint látjuk, a **for** függvény itt a **mycars** érték-tömböt alkotó **mycars[x]** értékeken hajtja végre a kiíratási rendelkezést, azaz az összes **x**-re kiírja a megfelelő **mycars[x]**-értéket. Mivel a **for** függvényben az **x** (belső) változót is inherensen definiáltuk, melynek értékeit mindenképpen csak a **mycars** array adhatja meg, az **x** külső definíciója felesleges – amint megjegyzés-jelek közé is tettük.

A megjelenő felirat:
Saab
Volvo
BMW

Array-objektum – kézikönyv
Az **array** objektumokra vonatkozó összes jellemzőt és módszert (és rövid jellemzésüket, példákkal) megtaláljuk array-objektum – kézikönyvünkben.

Mi az az array?
Az **array** egy különleges változó, melynek egyszerre több értéke lehet.

Bizonyos, hasonló jellegű változó-érték párokból (pl. autótípusok) listákat képezhetünk. Hogyha a listaelemek mindegyike külön változót képvisel, a lista így néz ki:

```
var car1="Saab";
var car2="Volvo";
var car3="BMW";
```

Ez az alak azonban nehezen kezelhető, hogyha pl. sok (mondjuk 300) autómárka közül valamilyen ciklussal kívánunk megkeresni egyet. Ekkor célszerűbb **array**-t használnunk, melynek ahol a változó-név és az érték-sorszám elkülönül, így egyszerűbb JS-tel kezelhető.

Az array az összes változó-értéket egy összefoglaló (változó-)név alatt tartalmazza. E névre hivatkozva bármely érték könnyen elérhető, az azonosítószáma révén.

Array készítése
Array-ket háromféleképpen definiálhatunk; ezeket e következő három példában mutatjuk be, melyek mindegyike egy **myCars** nevű array-objektumot határoz meg:

1.:

```
var myCars=new Array(); // regular array (add an optional integer
myCars[0]="Saab";       // argument to control array's size)
myCars[1]="Volvo";
myCars[2]="BMW";
```

Ez az array-k definiálásának közönséges módszere, ahol minden egyes értékhez egy beépített számértéket társítunk, az egyes értékek könnyebb azonosítására.

2.:

```
var myCars=new Array("Saab","Volvo","BMW"); // condensed array
```

Ez a példa egy összevont array-alakot mutat, ahol az érték-alternatívákat a **new Array()** konstruktor (=constructor) paramétereként adjuk meg.

3.:

```
var myCars=["Saab","Volvo","BMW"]; // literal array
```

Itt az egyes érték-alternatívákat egyetlen, összetett értékként egyszerűen hozzárendeljük a változó-névhez; vagyis a **myCars** értékét közvetlenül, és nem a **new Array()** konstruktoron keresztül adjuk meg.

Hogyha számokat vagy IGAZ/HAMIS értékeket írunk az **array**-be, akkor a változó jellege szöveg(-string) helyett szám vagy logikai érték lesz.

Hivatkozás array-re

Az array-ben szereplő egyes értékekre az array (azaz a gazda-változó) nevével és az érték sorszámával hivatkozunk. A sorszámozás 0-val kezdődik.

A következő rendelkezés

```
document.write(myCars[0]);
```

az első (**”Saab”**) értékre utal, így eredménye a **Saab** felirat lesz.

Array-értékek módosítása

A meglévő array-ben szereplő értékek módosításához (hogyha az eredeti array-elem manuális átírása nem lehetséges) elegendő, ha a módosítandóval azonos sorozatszámmal egy eltérő értéket írunk a felsorolás végére.

Az alábbi kód

```
<html>
<body>

<script type="text/javascript">
var mycars = new Array();
mycars[0] = "Saab";
mycars[1] = "Volvo";
mycars[2] = "BMW";
mycars[0] = "Opel";

for (x in mycars)
{
document.write(mycars[x] + "<br />");
}
</script>

</body>
</html>
```

eredménye tehát az

Opel
Volvo
BMW
felirat, jelezve, hogy a **mycars[0]** változó értéke **Saab**-ról **Opel**-re változott.

További példák
Első példánkban két array összekapcsolását látjuk a **concat()** módszer segítségével:

```
<html>
<body>
<script type="text/javascript">

var parents = ["Jani", "Tove"];
var children = ["Cecilie", "Lone"];
var family = parents.concat(children);
document.write(family);

</script>
</body>
</html>
```

Elöször meghatérozzuk a **parents** és **children** összetett (array) változókat, majd egy harmadik (**family**) változó értékeként egyesítjük előbbit az utóbbival, a **concat()** függvény segítségével. A kiírt szöveg:
Jani,Tove,Cecilie,Lone

Második példánkban három array-t egyesítünk, ugyancsak a **concat()** függvénnyel:

```
<html>
<body>
<script type="text/javascript">

var parents = ["Jani", "Tove"];
var brothers = ["Stale", "Kai Jim", "Borge"];
var children = ["Cecilie", "Lone"];
var family = parents.concat(brothers, children);
document.write(family);

</script>
</body>
</html>
```

Mint látjuk, először itt is külön-külön definiáljuk az egyesítendő változókat, majd a legelsőhöz a **concat()** fügvénynyel hozzácsatoljuk a másik kettőt, melyeknek neve a függvény paraméterében, vesszővel elválasztva szerepel. A felirat:
Jani,Tove,Stale,Kai Jim,Borge,Cecilie,Lone

Ugyanezt az eredményt érhetjük el, ha az array-ket a három eltérő módszerrel állítjuk össze, a **concat()** függvényeket pedig egymásba ágyazzuk:

```
<html>
<body>
<script type="text/javascript">

var parents = new Array();
parents[0]="Jani";
parents[1]="Tove";
var brothers = new Array("Stale", "Kai Jim", "Borge");
var children = ["Cecilie", "Lone"];
var family = parents.concat(brothers.concat(children));
document.write(family);
```

```
</script>
</body>
</html>
```

Harmadik példánkban egy array összes elemének egy (szöveg-)stringbe történő összeolvasztását láthatjuk, a **join()** módszer révén:

```
<html>
<body>
<script type="text/javascript">

var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write(fruits.join() + "<br />");
document.write(fruits.join("+") + "<br />");
document.write(fruits.join(" and "));

</script>
</body>
</html>
```

Itt a **fruits** array elemeit először közvetlenül egymás után, az alapértelmezett vesszőkkel elválasztva íratjuk ki, majd sortörést hagyunk. A második sorban az értékek között vessző helyett + szerepel (de lehet pontosvessző vagy szóköz vagy sortörés is, ha azt írunk az idézőjelek kőzé), a harmadik sorban pedig az **and** szócska (a többitől szóközökkel elválasztva, mint a kódból is látszik). A kód eredményéül kapott három sor tehát:

Banana,Orange,Apple,Mango
Banana+Orange+Apple+Mango
Banana and Orange and Apple and Mango

Mivel a **documents.write(fruits);** rendelkezéssel eleve az egész arrayt kiírná a gép, vesszőkkel elválasztva és szóközök nélkül, a **join()** módszernek az elválasztójel megadásában van jelentősége; pl. az array-elemeket könnyen HTML-listává alakíthatjuk:

```
<html>
<body>
<script type="text/javascript">

var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write("<ul><li>" + fruits.join("</li><li>") + "</li></ul>");

</script>
</body>
</html>
```

Mint látjuk, itt a lista elé és mögé, valamint az array-értékek közé a megfelelő HTML-tageket kíratva, egy HTML-listét kapunk:

- **Banana**
- **Orange**
- **Apple**
- **Mango**

Negyedik példánkban eltávolítjuk egy array utolsó elemét a **pop()** módszer segítségével:

```
<html>
<body>
<script type="text/javascript">

var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write(fruits.pop() + "<br />");
document.write(fruits + "<br />");
```

```
document.write(fruits.pop() + "<br />");
document.write(fruits);

</script>
</body>
</html>
```

Mint látjuk, a **pop()** függvény mindig felveszi annak az array-nak az utolsó értékét, melyre vonatkoztatjuk, és eltávolítja azt; így a négy gyümölcs közül először csak három, majd kettő kerül kiírásra.
Hogyha a **fruits** változónak csak egy értéke van (de az array-ként megadva!), akkor a **pop()** függvény azt is törli, hogyha pedig nem íratjuk át a függvény értékét a képernyőre vagy egy másik változóba, akkor rögtön el is vész ez az érték, mint az alábbi kód és kimenet példáján is látszik. A **pop()** függvény egyszerű változókra nincs hatással, csak array-kre;

```
<html>
<body>
<script type="text/javascript">

var fruits = ["Banana"];
document.write(fruits.pop() + "<br />");
document.write(fruits + "(nothing) <br />");
document.write(fruits.pop());

</script>
</body>
</html>
```

A kimenet:
Banana
(nothing)
undefined
tehát látjuk, hogy a **pop()** függvény értéke a leválasztás után már nem marad meg.
Egy gyors példa a „lecsípett” érték azonnali átiratására:

```
<html>
<body>
<script type="text/javascript">

var fruits = ["Banana", "Orange"];
var fru = fruits.pop();
document.write(fruits + "<br />");
document.write(fru);

</script>
</body>
</html>
```

A kimenet:
Banana
Orange

Ötödik példánkban a **push()** módszerrel hozzáadunk egy elemet az array végéhez:

```
<html>
<body>
<script type="text/javascript">

var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write(fruits.push("Kiwi") + "<br />");
document.write(fruits.push("Lemon","Pineapple") + "<br />");
document.write(fruits);
```

```
</script>
</body>
</html>
```
Amint látjuk, a **fruits** array-változó négy értéke kzé felveszünk egy ötödiket; ekkor a **fruits.push(”Kiwi”)** rendelkezés értéke **5**; majd még két elemet hozzáadva **7**-re emelkedik; amint azt a változó kiíratásakor is látjuk, a kimenetben:
5
7
Banana,Orange,Apple,Mango,Kiwi,Lemon,Pineapple

Hatodik példánkban egy array elemeinek sorrendjét megfordítjuk a **reverse()** módszerrel:
```
<html>
<body>
<script type="text/javascript">

var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write(fruits.reverse());

</script>
</body>
</html>
```
Amint az a kimenetből:
Mango,Apple,Orange,Banana
látszik, az array elmei változatlanok,de sorrendjük megfordult.

Hetedik példánkban, a **pop()** módszer analógiájára a **shift()** módszerrel eltávolítjuk egy array első értékét:
```
<html>
<body>
<script type="text/javascript">

var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write(fruits.shift() + "<br />");
document.write(fruits + "<br />");
document.write(fruits.shift() + "<br />");
document.write(fruits);

</script>
</body>
</html>
```
A kimenet:
Banana
Orange,Apple,Mango
Orange
Apple,Mango
Mint az magából a példából is kitetszik, a ***változónév***.**pop()** kifejezéshez hasonlóan a ***változónév*.shift()** kifejezés értéke is minden egyes végrehajtáskor megváltozik; így lehetséges, hogy az első és második kiíratáskor eltérő értéket tartalmaz.Vagyis a függvény mindig új értéket csíp le az array elejéből, ha pedig nem maradt ott több, akkor értéke **unidentified** lesz.

Nyolcadik példánkban a **slice()** módszerrel elemeket jelölünk ki egy array-ből.
A **slice()** módszerrel egy array bizonyos részletét (tehát voltaképpen egy kisebb array-t) jelölhetjük ki. A módszer mondattana: ***array*.slice(*kezdőérték*, *végérték*)**.
A kezdőértéket (ami egy szám) mindig megadjuk; megmutatja, hogy hányadik sorszámú elemnél kezdjük a kijelölést. Pl. ha értéke 1, akkor az elsőt (0 sorszámút) kivéve az array összes elemét kijelöljük. A kijelölést

az array végétől is kezdhetjük; ekkor a kezdőérték negatív szám. Hogyha pl. a kezdőérték -2, akor az array két utolsó elemét jelöltük ki.
Hogyha a kijelölt kezdőértéktől nem az array végéig kívánunk kijelölni, meg kell adnunk a végérték sorszámát is, ami már nem kerül kijelölésre.
A példában az négytagú array-ből képezhető összes kiemelést bemutatjuk:

```
<html>
<body>
<script type="text/javascript">

var fruits = ["Banana", "Orange", "Apple", "Mango"];

document.write(fruits + "<br /><br />");

document.write(fruits.slice(0) + "<br />");
document.write(fruits.slice(-4) + "<br /><br />");

document.write(fruits.slice(0,3) + "<br />");
document.write(fruits.slice(-4,-1) + "<br />");
document.write(fruits.slice(1) + "<br />");
document.write(fruits.slice(-3) + "<br /><br />");

document.write(fruits.slice(0,2) + "<br />");
document.write(fruits.slice(-4,-2) + "<br />");
document.write(fruits.slice(1,3) + "<br />");
document.write(fruits.slice(-3,-1) + "<br />");
document.write(fruits.slice(2) + "<br />");
document.write(fruits.slice(-2) + "<br /><br />");

document.write(fruits.slice(0,1) + "<br />");
document.write(fruits.slice(-4,-3) + "<br />");
document.write(fruits.slice(1,2) + "<br />");
document.write(fruits.slice(-3,-2) + "<br />");
document.write(fruits.slice(2,3) + "<br />");
document.write(fruits.slice(-2,-1) + "<br />");
document.write(fruits.slice(3) + "<br />");
document.write(fruits.slice(-1) + "<br />");

</script>
</body>
</html>
```

Mint látjuk, először kiíratjuk az array értékét.
Ezután kiemeljük az első négy elemet (azaz a teljes array-t), a rendelkezésre álló kétféle (elölről ill. hátulról számoló eljárással).
Harmadszor kiemeljük az első három, majd a második három elemet, ugyancsak két-kétflleképpen, ami négy sor kódot jelent.
Negyedszer kiemeljük az első kettő, a második-harmadik és a harmadik-negyedik elemet, ami három pár, azaz hat sor kódot jelent.
Ötödször kiemeljük külön mind a négy elemet, ami négy pár, azaz nyolc sor kódot jelent.
A kiírt szöveg:
Banana,Orange,Apple,Mango

Banana,Orange,Apple,Mango
Banana,Orange,Apple,Mango

Banana,Orange,Apple

Banana,Orange,Apple
Orange,Apple,Mango
Orange,Apple,Mango

Banana,Orange
Banana,Orange
Orange,Apple
Orange,Apple
Apple,Mango
Apple,Mango

Banana
Banana
Orange
Orange
Apple
Apple
Mango
Mango

Kilencedik példánkban egy array elemeit ABC szerint növekvő sorrendbe állítva kiíratjuk, a **sort()** módszerrel:

```
<html>
<body>
<script type="text/javascript">

var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write(fruits.sort());

</script>
</body>
</html>
```

Mint látjuk, egyszerűen a **sort()** módszer alkalmazzuk egy négytagú array-re. A megjelenő kimenet:
Apple,Banana,Mango,Orange

Hogyha ezután kiíratjuk a **fruits** array-t,

```
<html>
<body>
<script type="text/javascript">

var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write(fruits.sort() + "<br />");
document.write(fruits);

</script>
</body>
</html>
```

az eredmény:
Apple,Banana,Mango,Orange
Apple,Banana,Mango,Orange
jelezve, hogy a **fruits** array elemeinek sorendje megváltozott, azaz a **sort()** módszerrel nemcsak időlegesen, hanem véglegesen újradefiniáltuk az array-t, persze csak sorrendileg.
Ezt a következő, tagoltabban felírt kóddal is bizonyíthatjuk:

```
<html>
<body>
```

```
<script type="text/javascript">

var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
document.write(fruits);

</script>
</body>
</html>
```
amelynek eredménye:
Apple,Banana,Mango,Orange.
Ez azt jelenti, hogy a **fruits.sort();** rendelkezéssel nemcsak a rendelkezés idejére, hanem a parancssor teljes további részére is átrendeztük az array-t.,

Tizedik példánkban az **n.sort()** módszert egy szám-array növekvő sorba rendezéséhez használjuk:
```
<html>
<body>
<script type="text/javascript">

function sortNumber(a, b)
{
return a - b;
}

var n = ["10", "5", "50", "25", "100", "1"];
document.write(n.sort(sortNumber));

</script>
</body>
</html>
```
Mint látjuk, a numerikus rendezés irányát a **sortNumber** függvénnyel adjuk meg, melynek belső, **a** és **b** változói a rendezett elemek közötti összefüggést mutatják meg. A kapott felirat:
1,5,10,25,50,100

Tizenegyedik példánkban ugyanezen számsort fordítva rendezzük (azaz tagjait numerikusan csökkenő sorba állítjuk):
```
<html>
<body>
<script type="text/javascript">

function sortNumber(a, b)
{
return b - a;
}

var n = ["10", "5", "50", "25", "100", "1"];
document.write(n.sort(sortNumber));

</script>
</body>
</html>
```
Mint látjuk, ez csak a **sortNumber(a,b)** függvény parancsában szereplő változók sorrendjében szerepel az előzőtől, az eredmény pedig a korábbival ellenkező:
100,50,25,10,5,1

Tizenkettedik példánkban a **splice()** módszerrel hozzáadunk egy elemet egy array-höz.
A **splice()** módszerrel egy array megadott helyén értékeket távolíthatunk el ill. szúrhatunk be. Mondattana:
***array-név*.splice(*hely-szám*,*eltávolítandó_elemek_száma*,*beillesztenő_elem1*,*beillesztendő_elem2*,…);**
A **hely-szám** megadja, hogy az array hanyadik eleme elé szúrjuk be az elemet, illetve azt is jelentheti, hogy hanyadik elemet töröljük. Az **eltávolítandó elemek számával** megadhatjuk, hogy hány elemet kívánunk törölni [az első törlendő elem (lásd: **hely-szám**) után soronkövetkezők lévők közül]. A **beillesztendő elemek** helyére vesszővel elválasztva beírhatjuk az array-be felveendő szavakat (idézőjelben), számokat, logikai értékeket stb..

```
<html>
<body>
<script type="text/javascript">

var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write("Removed: " + fruits.splice(2,1,"Lemon") + "<br />");
document.write(fruits);

</script>
</body>
</html>
```

Mint látjuk, a lista **2**. számú (azaz sorban harmadik) helyéről eltávolítjuk az ott álló **Apple** értéket, és helyette (formálisan eléje) beszúrjuk a **Lemon**t. Az eredmény:
Removed: Apple
Banana,Orange,Lemon,Mango

Tizenharmadik példánkban a **toString()** módszert alkalmazzuk egy array-re, azaz vesszővel elválasztva kiíratjuk annak összes értékét. Ez gyakorlatilag megegyezik azzal, mintha az array-néven, azaz a fő-változón a **document.write()** módszert végrehajtanánk:

```
<html>
<body>
<script type="text/javascript">

var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write(fruits.toString());
document.write("<br />" + fruits);

</script>
</body>
</html>
```

A böngésző válasza mindkét kiíratási parancsra azonos:
Banana,Orange,Apple,Mango
Banana,Orange,Apple,Mango

Tizennegyedik példánkban az **unshift()** módszer segítségével újabb értékeket illesztünk egy array elejéhez:

```
<html>
<body>

<script type="text/javascript">

var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write(fruits.unshift("Kiwi") + "<br />");
document.write(fruits.unshift("Lemon","Pineapple") + "<br />");
document.write(fruits);

</script>
```

```
<p><b>Note:</b> The unshift() method does not work properly in Internet
Explorer, it only returns undefined!</p>

</body>
</html>
```

Az array definiálása után először a **Kiwi** értéket szúrjuk be a lista elejére, maja az újabb, már **Kiwi**-vel kezdődő array elejére a **Lemon** és **Pineapple** értéket is.
Az első két felirat IE-ben nem működik (**undefined** érték jelenik meg); Firefoxban azonban látszik, hogy az array-elemek száma előbb **5**-re, majd **7**-re emelkedett, amiről a szavak kiíratásakor, azaz a harmadik kimenet-sorban meggyőződhetünk. Mint látjuk, az **unshift()** módszer a megadott szó-listát egyszerűen beírja az array elé; így többszöri beíratások után a régebben eléíratott szavak már nem elöl, hanem hátrébb állnak. A böngészőben megjelenő szöveg:

5
7
Lemon,Pineapple,Kiwi,Banana,Orange,Apple,Mango

Note: The unshift() method does not work properly in Internet Explorer, it only returns undefined!

V. JavaScript – a logikai érték (boolean) objektum

A logikai érték (=boolean, azaz IGAZ/HAMIS érték) objektumot nem logikai értékék logikaivá alakítására használjuk.

Néhány egyszerű példa

Első példánkban számok és stringek logikai jelentését vizsgáljuk a **new Boolean()** objektummal:

```
<html>
<body>
<script type="text/javascript">

var b1=new Boolean( 0);
var b2=new Boolean(1);
var b3=new Boolean("");
var b4=new Boolean(null);
var b5=new Boolean(NaN);
var b6=new Boolean("false");

document.write("0 is boolean "+ b1 +"<br />");
document.write("1 is boolean "+ b2 +"<br />");
document.write("An empty string is boolean "+ b3 + "<br />");
document.write("null is boolean "+ b4+ "<br />");
document.write("NaN is boolean "+ b5 +"<br />");
document.write("The string 'false' is boolean "+ b6 +"<br />");

</script>
</body>
</html>
```

Első, **b1** változónk értéke megegyezik a **0** számra vonatkoztatott **new Boolean()** módszer, azaz logikaiérték-vizsgálat eredményével. Mint az alább közölt kiíratási eredményből látszik, a **0** szám logikai értéke HAMIS.
A második változóba írt **1**-es szám logikai értéke IGAZ.
Az üres **new Boolean("")** módszer logikai értéke HAMIS.
A negyedik rész-példa szerint a **null** logikai érték (nem szöveg, nincs idézőjelben!) jelentése HAMIS.
Az ötödik, **NaN** (=not a number) kifejezés logikai értéke hamis.
A hatodik, „false" szót (nem logikai érték!) tartalmazó **new Boolean()** módszer értéke IGAZ; mivel a módszernek van szöveges tartalma (a harmadik esettel ellentétben). Az idézőjelekbe tett logikai érték tehát szöveggé válik. A kimenet:

0 is boolean false
1 is boolean true
An empty string is boolean false
null is boolean false
NaN is boolean false
The string 'false' is boolean true

Logikaiérték-objektum – kézikönyv

Az logikaiérték objektumokra vonatkozó összes jellemzőt és módszert (és rövid jellemzésüket, példákkal) megtaláljuk logikaiérték-objektum – kézikönyvünkben.

Logikaiérték-objektumok kezelése

A logikaiérték-objektum kétféle, vagyis IGAZ vagy HAMIS értékkel rendelkezhet.

A következő kód egy **myBoolean** nevű objektumot (változót) definiál:

```
var myBoolean=new Boolean();
```

Ha a logikaiérték-objektumnak (a zárójelbe írva) nincs kezdeti értéke, vagy az 0, -0, null, ””, false, undefined vagy NaN, akkor logikai értéke HAMIS.

Minden más esetben [még akkor is, hogyha a ”false” szöveget (stringet) tartalmazza, tehát van értelmezhető tartalma], IGAZ.

A következő rendelkezések mindegyike HAMIS kezdeti értékű logikaiérték-objektumokat definiál:

```
var myBoolean=new Boolean();
var myBoolean=new Boolean(0);
var myBoolean=new Boolean(null);
var myBoolean=new Boolean("");
var myBoolean=new Boolean(false);
var myBoolean=new Boolean(NaN);
```

Az alábbiak pedig IGAZ logikai értékűeket:

```
var myBoolean=new Boolean(1);
var myBoolean=new Boolean(true);
var myBoolean=new Boolean("true");
var myBoolean=new Boolean("false");
var myBoolean=new Boolean("Richard");
```

V. JavaScript – a math objektum

A math (=matematikai) objektumot matematikai jellegű rendelkezésekben használjuk.

Néhány egyszerű példa

Első példánkban a **round()** módszer alkalmazását láthatjuk:

```
<html>
<body>
<script type="text/javascript">

document.write(Math.round(0.60) + "<br />");
document.write(Math.round(0.50) + "<br />");
document.write(Math.round(0.49) + "<br />");
document.write(Math.round(-4.40) + "<br />");
document.write(Math.round(-4.60));

</script>
</body>
</html>
```

A parancssor eredménye:

1
1
0
-4
-5

Mint látjuk, a **Math.round()** módszer a paraméterként beleírt számot (tizedestörtet) egészekre kerekíti, vagyis a programozás szaknyelván kifejezve, a legközelebbi egész értékét adja vissza.

Második példánkban a **random()** módszerrel 0 és 1 közötti számokat generálunk, majd ezeken végzünk műveleteket:

```
<html>
<body>
<script type="text/javascript">

//return a random number between 0 and 1
document.write(Math.random() + "<br />");

//return a random integer between 0 and 10
document.write(Math.floor(Math.random()*11));

</script>
</body>
</html>
```

Az eredmény mindig változó, de lehet pl.:

0.3463347091974428
6

Amint látjuk, a **Math.random()** módszer egy 16 értékesjegyből álló, 0 és 1 közötti decimális számértéket „ad vissza”, melyen aztán további (kiíratő, szorzási stb.) műveleteket végezhetünk. Természetesen egy adott pillanatban valószínűleg a kód minden egyes **Math.random()** módszerénekeltérő lesz az értéke; így, ha csupán egyetlen véletlenszerű értéket szeretnénk több helyen felhasználni, akkor egyetlen módszer értékét egy változóhoz kell rendelnünk.

Harmadik példánkban a **max()** módszert alkalmazzuk egy számsor legnagyobb elemének kíratására:

```
<html>
<body>
<script type="text/javascript">
```

```
document.write(Math.max(5,10) + "<br />");
document.write(Math.max(0,150,30,20,38) + "<br />");
document.write(Math.max(-5,10) + "<br />");
document.write(Math.max(-5,-10) + "<br />");
document.write(Math.max(1.5,2.5) + "<br />");
document.write(Math.max() + "<br /><br />");

var numb = 9;
var numbe = 19
document.write(Math.max(numb,numbe) + "<br />");

</script>
</body>
</html>
```

A **Math.max()** módszer tehát megadja a paraméterébe vesszővel elválasztva beleírt számok közül a legnagyobbat. (Negatív számok között természetesen a legkisebb abszolútértékűt fogjakiválasztani, mint az a példa 3-4. sorából látszik.) Hogyha nem adunk meg paramétert, értéke **-Infinity**.
Mint látjuk, az értékeket változóként is megadhatjuk. A parancssor által kiírt szöveg:

10
150
10
-5
2.5
-Infinity

19

Kérdés: hogyan lehet egy array értékei közül a legnagyobbat kiválasztani anélkül, hogy elemet kéne kiemelni belőle?

Negyedik példánkban a **min()** módszert a legkisebb számérték kikeresésére alkalmazzuk:

```
<html>
<body>
<script type="text/javascript">

document.write(Math.min(5,10) + "<br />");
document.write(Math.min(0,150,30,20,38) + "<br />");
document.write(Math.min(-5,10) + "<br />");
document.write(Math.min(-5,-10) + "<br />");
document.write(Math.min(1.5,2.5) + "<br />");
document.write(Math.min() + "<br /><br />");

var numb = 9;
var numbe = 19
document.write(Math.min(numb,numbe) + "<br />");

</script>
</body>
</html>
```

A **Math.max()** ellentettjeként a **Math.min()** módszer a paraméterébe írt számok ill. változók közül a legkisebbet írja ki:

5
0
-5
-10

1.5
Infinity

9
Ha nem adunk meg számot, a módszer értréke **Infinity**. Hogyha egyetlen szót adunk meg (idézőjelek között), akkor az érték megegyezik a szóval, ha többet, akkor pedig **NaN**. Hogyha egy logikai értéket adunk meg (pl. true), akkor a módszer-érték megegyező lesz vele, hogyha többet (akér egyezőeket is), akkor pedig **0** az érétk. Mindebből leszűrhetjük, hogy a **max()** ill. **min()** **Math**-módszerek elsőrenden számok kezelésére valók!

Math-objektum – kézikönyv
Az **math** objektumokra vonatkozó összes jellemzőt és módszert (és rövid jellemzésüket, példákkal) megtaláljuk math-objektum – kézikönyvünkben.

A math objektum
A **math** objektum segítségével matematikai rendelkezéseket tehetünk. Több matematikai állandót és módszert tartalmazhatnak.

A **math** jellemzők és módszerek mondattana:
```
var x=Math.PI;
var y=Math.sqrt(16);
```
Az **x** változó értéke π = 3,14; az **y**-é pedig gyök **16** = 4.

A **math** objektum nem konstruktor. A **math**-hoz tartozó összes jellemző és módszer meghívható a **Math** objektummal, anélkül, hogy annak értékét definiálnánk.

Matematikai állandók
A JS-ben a **math** objektum révén nyolc matematikai állandó áll rendelkezésünkre, ezek:

$$e, \pi, \sqrt{2}, \sqrt{\frac{1}{2}}, \ln(2), \ln(10), \log_2(e), \lg(e).$$

A JS-ben ezek a következő kifejezésekkel érhetők el:
```
Math.E
Math.PI
Math.SQRT2
Math.SQRT1_2
Math.LN2
Math.LN10
Math.LOG2E
Math.LOG10E
```

Matematikai módszerek
A **math** objektummal elérhető állandók mellett különféle matematikai módszereket is alkalmazhatunk a JS-ben.

A következő példában a **Math** objektumra vonatkozó **round()** módszerrel egy számot a legközelebbi egészértékre kerekítünk:
```
document.write(Math.round(4.7));
```
A rendelkezés eredménye a weboldalon:
5

Második példánkban pedig a **Math** objektum **random()** módszerével egy 0 és 1 közötti számot generálunk és íratunk ki:
```
document.write(Math.random());
```
A kód a weboldalon egy az alábbihoz haonló, 16 értékesjegyű tizedestörtet eredmlényezhet:

0.4609322827257597

Harmadik példánkban a **Math** objektum **floor()** és **random()** módszereivel egy 0 és 10 közötti számot íratunk ki a weboldalra:

```
document.write(Math.floor(Math.random()*11));
```

A **Math.random()*11** kifejezés 0-tól 10.99999…-ig terjedő számot eredményezhet, melyet a **Math.floor()** módszerrel mindig lefelé (egészre) kerekítünk; így, mivel a kerekítendő érték szinte mindig kisebb 11-nél, az oldalra 0 és 10 közötti számok kerülnek ki. Hogyha a számérték kerek (pl. itt nemcsak 0-10, hanem 11 is lehet, egyetlen esetben), akkor az értéke további változtatás nélkül kerül kiírásra.
A kód kimenete ennek megfelelően pf. **5** lehet.

IV. JavaScript – a RegExp objektum

Az **RegExp** objektumnév a regular expression (=kifejezés-alak) szintagma rövidítése.

RegExp-objektum – kézikönyv
Az **RegExp** objektumokra vonatkozó összes jellemzőt és módszert (és rövid jellemzésüket, példákkal) megtaláljuk RegExp-objektum – kézikönyvünkben.

Mi az a RegExp?
Az **RegExp** (regular expression) egy karakter-mintázatot leíró objektum.

Pl. szövegben való kereséshez meghatározhatjuk vele azokat a betű-összetételeket, melyeket meg kívánunk találni.

A mintázat mindössze egyetlen kakarterből is állhat; de bonyolultabb is lehet, s így felhasználható mondat-elemzésre, formátum-ellenőrzésre (pl. űrlapoknál), bizonyos szavak lecserélésére stb..

A regular expression objektumot tehát szövegek hatékony mintázat-egyeztetésére ill. kereső-helyettesítő feladatokra alkalmazhatjuk.

A **RegExp** objektum mondattana kétféle lehet:
1.:
var *változónév*=new RegExp(*mintázat*,*feltételek*);

vagy

2.:
var *változónév* =/*mintázat*/*feltételek*;

ahol a **mintázat** (=pattern) a keresendő kifejezést, a **feltételek** (=modifiers) pedig a keresési feltételeket (pl. teljes szövegben, esetfüggően stb.) jelentik.

RegExp feltételek
A feltételeket (=modifiers) esetfüggetlen és általános (=global) keresésekhez használjuk.

Az **i** feltétel az esetfüggetlen, a **g** pedig a globális keresést jelenti; azaz utóbbi esetben a program az összes megfelelelő szót megkeresi, azaz nem áll le az első találat után, mint egyébként.

Első példánkban esetfüggetlen keresést hajtunk végre egy rövid szövegben:

```
<html>
<body>

<script type="text/javascript">
var str = "Visit W3Schools";
var patt1 = /w3schools/i;
document.write(str.match(patt1));
</script>

</body>
</html>
```

Először definiáljuk a string-et, melyben keresni fogunk, majd a **match()** módszer paraméterét (a **patt1** változóban); végül pedig kiíratjuk a **str** változón **patt1** mintázattal elvégzett keresés eredményét:
W3Schools
Amint látszik, a keresőszó és a találat kis- és nagybetűs írásmódban eltér.

Második példánkban (esetfüggő!) globális keresést hajtunk végre az **is** szóra vonatkozólag:

```
<html>
<body>
<script type="text/javascript">

var str="Is this all there is?";
var patt1=/is/g;
document.write(str.match(patt1));

</script>
</body>
</html>
```

Az eredmény:

is,is

jelzi, hogy a string-ben kétszer található meg az **is** betűösszetétel, egyszer a **this** szó végén, másodszor pedig a mondat végén (**is**). Mint látjuk, a string-et és a **match()** módszer paraméterét itt is változókként, külön definiáltuk.

Hogy a keresés alapértelmezetten esetfüggő és nem globális, azt a példa módosításával jól megfigyelhetjük:

```
<html>
<body>
<script type="text/javascript">

var str="is Is this all there Is?";
var patt1=/Is/;
document.write(str.match(patt1));

</script>
</body>
</html>
```

Itt ugyanis az eredmény csak **Is**, azaz a módszer a mondat elején álló **is**-t nem vette figyelembe, valamint a második **Is**-t sem. Tehát ha valamelyik feltételt nem adjuk meg, akkor a program automatikusan annak ellenkezőjét hajtja végre (globális helyett lokális, nem esetfüggő helyett esetfüggő stb. keresést).

Harmadik példánkban globális és esetfüggetlen módon keresünk rá az **is** szóra:

```
<html>
<body>
<script type="text/javascript">

var str="Is this all there is?";
var patt1=/is/gi;
document.write(str.match(patt1));

</script>
</body>
</html>
```

Mint látjuk, a két feltétel-paramétert egymás után írva adjuk meg (sorrendjük nem lényeges, tehát **gi** helyett **ig**-et is írhatunk).

Az eredmény:

Is,is,is

mutatja, hogy a mondat elején (nagybetűvel) és végén (kicsivel) álló **is** szócska mellett a **this** szó végén is megtalálható (kisbetűvel) az **is** (esetfüggetlen) betű-összetétel.

A test() módszer

A **test()** módszerrel egy érték string-beli meglétét vizsgáljuk a következő példában:

```
<html>
```

```
<body>
<script type="text/javascript">

var patt1=new RegExp("hing");
document.write(patt1.test("The best things in life are free"));

</script>
</body>
</html>
```
Mint látjuk, a **patt1** változó tartalmát (**hing**) keressük a **test()** függvény paramétereként beírt szövegben; ami megtalálható benne (**things**), így a függvény értéke IGAZ, tehát a parancssor eredménye:
true

Az exec() módszer
Az **exec()** módszer a **test()**-hez hasonlóan egy megadott értéknek egy string-ben való megtalálhatóságát vizsgála, de a megtalálhatóságot nem logikai értékkel, hanem a keresett érték, a sikertelenséget pedig a **null** érték felvételével jelzi:
```
<html>
<body>
<script type="text/javascript">

var patt1=new RegExp("hing");
document.write(patt1.exec("The best things in life are free"));

</script>
</body>
</html>
```
Mivel a string-ben megtalálható a **hing** betűösszetétel, az **exec()** függvény értéke **hing** lesz (és nem null), amint az a kimenetből is látszik:
hings

JavaScript
III. KÖNYV: HALADÓ SZINT

I. JavaScript – Böngésző-felismerés

A **navigator** objektum a felhasználó böngésző-típusával kapcsolatos információkat tartalmazza.

Böngésző-felismerés

Az eddig tanult JS-ek szinte minden JS-kompatibilis böngészőn működnek; azonban bizonyos – főleg régebbi – programokkal azért adódhatnak kompatibilitási problémák. Ezért olyakor hasznosnak bizonyul, ha webes szolgáltatóként felismertetjük az oldalt leklrdező böngésző típusát, hogy megfelelő, megjeleníthető tartalmat küldjünk számára.

Ennek legjobb módja, ha weboldalunknak különféle megjelenési lehetőségeket biztosítunk aszerint, hogy milyen típusú böngészővel töltik le őket. A **navigator** objektumot ennek érdekében használjuk, mivel tartalmazza a felhasználó böngészőjének nevét, verziószámát és egyéb adatokat is. A **navigator** objektumra egyelőre nincsenek általánosan elfogadott előírások, de az összes fontosabb böngésző támogatja a használatát.

A navigator objektum

A **navigator** objektum a felhasználó böngészőjével kapcsolatos összes lényeges adatot tartalmazza, mint azt az alábbi példa mutatja:

```
<html>
<body>
<script type="text/javascript">

document.write("Browser CodeName: " + navigator.appCodeName);
document.write("<br /><br />");
document.write("Browser Name: " + navigator.appName);
document.write("<br /><br />");
document.write("Browser Version: " + navigator.appVersion);
document.write("<br /><br />");
document.write("Cookies Enabled: " + navigator.cookieEnabled);
document.write("<br /><br />");
document.write("Platform: " + navigator.platform);
document.write("<br /><br />");
document.write("User-agent header: " + navigator.userAgent);

</script>
</body>
</html>
```

A kód kimenete IE mellett:

Browser CodeName: Mozilla

Browser Name: Microsoft Internet Explorer

Browser Version: 4.0 (compatible; MSIE 7.0; Windows NT 5.1; SIMBAR Enabled; SIMBAR={FAAFD2B9-E1C5-4d0f-8147-D77A7CD3B0A1}; SIMBAR=0; .NET CLR 1.1.4322; .NET CLR 2.0.50727; InfoPath.1; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)

Cookies Enabled: true

Platform: Win32

User-agent header: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; SIMBAR Enabled; SIMBAR={FAAFD2B9-E1C5-4d0f-8147-D77A7CD3B0A1}; SIMBAR=0; .NET CLR 1.1.4322; .NET CLR 2.0.50727; InfoPath.1; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)

Firefox mellett:
Browser CodeName: Mozilla

Browser Name: Netscape

Browser Version: 5.0 (Windows; hu)

Cookies Enabled: true

Platform: Win32

User-agent header: Mozilla/5.0 (Windows; U; Windows NT 5.1; hu; rv:1.9.2.13) Gecko/20101203 Firefox/3.6.13 (.NET CLR 3.5.30729)

II. JavaScript – Cookies

A JS **cookie**-kat (=sütiket) leggyakrabban a felhasználó azonosítására alkalmazzák.

Mi az a cookie?

A **cookie** egy változó, melyet a felhasználó gépén tárolunk. Akkor, mior az adott számítógép lekérdez egy oldalt, automatikusan elküldi az oldallal kapcsolatos cookie-kat is. A JS segítségével cookie-kat készíteni és fogadni egyaránt tudunk.

Néhány cookie-típus:

- Név-cookie (=name cookie)
  Tegyük fel, hogy a felhasználónak a weboldalra történő első belépéskor meg kell adnia a nevét; melyet aztán egy cookie-ban eltárolunk. Mikor a következő alkalommal meglátogatja az oldalt, máris névre szóló üdvözletet írhatunk ki neki (pl. **Welcome John Doe!**); amihez a nevet az (eltárolt) cookie-ból vesszük.
- Jelszó-cookie (=password cookie)
  Egy másik eset, hogy a felhasználónak az oldal használata során jelszót kell megadnia. Ezt azután (ill. esetenként, bizonyos böngésző-típusokban csak akkor, hogyha a felhasználó ezt engedélyezi) szintén eltárolhatjuk egy cookie-ban. A legközelebbi látogatáskor a felhasználó jelszavát automatikusan betöltheti a böngésző, a cookie-ból.
- Dátum-cookie (=date cookie)
  Elképzelhető az is, hogy a felhasználói lekérdezés időpontját tároljuk el egy cookie-ban, majd a következő alkalommal kiíratjuk vagy felhasználjuk ezt az adatot. Például kiírathatjuk a legutóbbi letöltésének idejét: **Your last visit was on Tuesday August 11, 2005!**. Az adat ismét a cookie-ból származik.

Cookie-k készítése és tárolása

A cookie-k használatához egyetlen, hosszú példát fogunk összeállítani és megvizsgálni. Egy, a felhasználó nevét tároló cookie-t készítünk. Az első letöltéskor a látogatónak meg kell adnia a nevét, melyet azután egy cookie-ban eltárolunk, majd a következő látogatáskor egy üdvözletben kiíratunk.

Először egy függvényt írunk, mely kiírja a felhasználó nevét egy **cookie** változóba:

```
function setCookie(c_name,value,expiredays)
{
var exdate=new Date();
exdate.setDate(exdate.getDate()+expiredays);
document.cookie=c_name+ "=" +escape(value)+
((expiredays==null) ? "" : ";expires="+exdate.toUTCString());
}
```

A függvény paraméterei a cookie nevét, értékét és érvényességi idejét (napokban) tartalmazzák.
A függvény parancssorában először definiáljuk az **exdate** helyi változót, melyet az ezen cookie-készítő függvény futtatásakor aktuális dátummal teszünk egyenlővé.
Ezt a dátum-értéket azután megnöveljük az érvénysségi idővel, így az **exdate** változó már azt a dátumot tartalmazza, amikor a cookie érvényessége lejár.
Ezután létrehozzuk a cookie-dokumentumot, melybe a következőket íratjuk bele:
először a cookie nevét és értékét, majd a lejárati dátumot, UTC-alakban.

Ezek után létrehozunk egy másik függvényt, amivel ellenőrizzük a cookie beállítását:

```
function getCookie(c_name)
{
if (document.cookie.length>0)
  {
  var c_start=document.cookie.indexOf(c_name + "=");
  if (c_start!=-1)
    {
```

```
    c_start=c_start + c_name.length+1;
    var c_end=document.cookie.indexOf(";",c_start);
    if (c_end==-1) c_end=document.cookie.length;
    return unescape(document.cookie.substring(c_start,c_end));
    }
  }
return "";
}
```

Mindenekelőtt betöltjük a **c_name** nevű cookie-t, majd egy **if** feltételes rendelkezés feltételében ellenőrizzük, tartalmaz-e egyáltalán valamit.
Hogyha eszerint a **document.cookes** objektum tartalmazza a cookie-(ka)t, akkor megvizsgáljuk, hogy az emeggyezik-e az imént eltárolttal. Hogyha a függvény megtalálja a cookie-nkat, akkor visszaadja annak értékét, ellenkező esetben viszont egy üres (szöveg-)stringet.

Utolsó lépésként létrehozzuk a cookie megléte esetén üzenetet kiíratő függvényt. Hogyha a cookie nem található, akkor egy beviteli dobozban ismét elkéri a felhasználó nevét:

```
function checkCookie()
{
var username=getCookie('username');
if (username!=null && username!="")
  {
  alert('Welcome again '+username+'!');
  }
else
  {
  username=prompt('Please enter your name:',"");
  if (username!=null && username!="")
    {
    setCookie('username',username,365);
    }
  }
}
```

Mint látjuk tehát, a **checkCookie()** fügvény először betölti az **username** nevű cookie-fájl tartalmát az **username** változóba, majd ellenőrzi, hogy annak értéke nem **null** vagy üres-e (**””**). Hogyha nem, akkor névre szóló üdvözletet írat ki egy figyelmeztető dobozban.
Ha a cookie nem tartalmaz semmit, akkor egy beviteli ablakban ismét elkéri a felhasználó nevét, és hogyha a dobozt kitöltöttük (amit ellenőriz is), eltárolja az adatot az **username** cookie-ba, **365** napos érvényességgel.

Íme a teljes parancssor (weboldalban):

```
<html>
<head>
<script type="text/javascript">
function getCookie(c_name)
{
if (document.cookie.length>0)
  {
  var c_start=document.cookie.indexOf(c_name + "=");
  if (c_start!=-1)
    {
    c_start=c_start + c_name.length+1;
    var c_end=document.cookie.indexOf(";",c_start);
    if (c_end==-1) c_end=document.cookie.length;
    return unescape(document.cookie.substring(c_start,c_end));
    }
```

```
  }
return "";
}

function setCookie(c_name,value,expiredays)
{
var exdate=new Date();
exdate.setDate(exdate.getDate()+expiredays);
document.cookie=c_name+ "=" +escape(value)+
((expiredays==null) ? "" : ";expires="+exdate.toUTCString());
}

function checkCookie()
{
var username=getCookie('username');
if (username!=null && username!="")
  {
  alert('Welcome again '+username+'!');
  }
else
  {
  username=prompt('Please enter your name:',"");
  if (username!=null && username!="")
    {
    setCookie('username',username,365);
    }
  }
}
</script>
</head>

<body onload="checkCookie()">
</body>
</html>
```

Mint látjuk, a három függvényt a weboldal fejrészében tároltuk el; a dokumentumtestben (pontosaban a **<body>** tagben) pedig csak az őket az oldal betöltésekor meghívó attribútumot helyeztük el.

[Pontosabban először a legutolsó, **checkCookie()** függvényt hívjuk csak meg, s abból hivatkozzzuk a második, **setCookie()**-t. Az első, **getCookie()** függvény feladata csupán az esetleg meglévő cookie tartalmának mielőbbi beolvasása, hogy a harmadik ill. esetleg a második (addigra már szintén betöltött) függvény meghívásakor már készen álljon.]

A cookie-érték megadása (beviteli dobozzal) mindig a **checkCookie()** függvény betöltésekor kezdődik, és itt, vagyis az utolsó függvény utolsó alfüggvényében (felételes rendelkezésében) került beállításra a (**365** napos) érvényességi idő is.

Az oldal betöltésekor megjelenik a beviteli ablak, amibe egy szót megadva, azontúl 365 napon át minden betöltésekor/frissításekor egy **Welcome again *név*!** feliratú figyelmeztető doboz jelenik meg.
Hogyha a gépen tárolt cookie-kat töröljük, az oldal ismét elkéri a jelszót; hogyha pedig azt nem adjuk meg, akkor a legközelebbi betöltéskor ismét elkéri stb..

III. JavaScript – Űrlap-hitelesítés

JavaScript – Űrlap-hitelesítés

A JS-tet HTML űrlapok adatainak szerverre küldése előtti validálására (=validating), azaz hitelesítésére is felhasználhatjuk.

A JS-tel megoldható leggyakoribb űrlapadat-ellenőrzési feladatok:

- Üresen hagyott-e kitöltendő mezőket a felhasználó?
- Valódi e-mail címet adott-e meg a felhasználó?
- Helyes-e a felhasználó által megadott dátum?
- Megfelelő jellegű adattal (számal/szöveggel/logikai értékkel) töltötte-e fel a beviteli mezőt a felhasználó? (Pl. mobilszám helyett nem e-mail címet adott-e meg?)

Kitöltendő mezők ellenőrzése

Az alábbi példánkban szereplő függvény ellenőrzi, hogy nem maradt-e üresen egy (kötelezően kitöltendő) szövegbeviteli mező. Ha üres, a böngésző megjelenít egy erre utaló figyelmeztető ablakot, a függvény pedig felveszi a **false** értéket, ami miatt a böngésző nem továbbíthatja az űrlapot:

```
function validateForm()
{
var x=document.forms["myForm"]["fname"].value
if (x==null || x=="")
  {
  alert("First name must be filled out");
  return false;
  }
}
```

Mint látjuk, a **validateForm()** függvény először az **x** belső változóba betölti a **myForm** űrlap **fname** nevű szövegmezőjének értékét; majd egy feltételes rendelkezést tesz arra az esetre, ha az érték **null** ill. üres (**””**). Ekkor hibaüzenetet írat ki, és felveszi a **false** értéket. (Ellenkező esetben nem történik semmi.)

Mint a példa alábbi, teljesebb alakjában látjuk, a **validateForm()** függvényt az űrlap továbbításakor hívjuk meg:

```
<html>
<head>
<script type="text/javascript">

function validateForm()
{
var x=document.forms["myForm"]["fname"].value
if (x==null || x=="")
  {
  alert("First name must be filled out");
  return false;
  }
}

</script>
</head>
<body>

<form name="myForm" action="demo_form.asp" onsubmit="return
validateForm()" method="post">
First name: <input type="text" name="fname">
<input type="submit" value="Submit">
</form>
```

```
</body>
</html>
```
Mint látjuk, az oldal betöltésekor megjelenik a **First name:** felirat, a **fname** nevű szövegmező és a **Submit** feliratú továbbító-gomb.

A **<form>** küldésekor (**onsubmit**) meghívjuk a **validateForm()** függvényt; aminek eredményeként kitöltetelenség esetén egy **First name must be filled out** feliratú figyelmeztető-ablak jelenik meg; ellenkező esetben a beírt tartalom megjelenik a **demo_form.asp** oldalon (mint „target:"_top"").

E-mail cím ellenőrzése
Az alábbi függvénnyel ellenőrizhetjük, hogy egy szövegmezőbe írt e-mail cím megfelel-e a vele szemben támasztható általános „mondattani" követelményeinek.
Tehát ellenőrzi, hogy a bevitt adatok között található-e @-jel, ill. legalább egy pont (.); ugyanígy azt is, hogy a @ nem az első jel-e az e-mail címben (ez ugyanis nem lehetséges), és azt is, hogy az utolsó pont a @-jel után, illetve a cím utolsó karaktere az utolsó pont után minimum két karakter távolságban található-e (ez is előírás az e-mail címekre nézve).

A függvény kódja:
```
function validateForm()
{
var x=document.forms["myForm"]["email"].value
var atpos=x.indexOf("@");
var dotpos=x.lastIndexOf(".");
if (atpos<1 || dotpos<atpos+2 || dotpos+2>=x.length)
  {
  alert("Not a valid e-mail address");
  return false;
  }
}
```

Ezt az alábbi teljesebb weboldal az e-mail cím továbbításakor tölti be:
```
<html>
<head>
<script type="text/javascript">

function validateForm()
{
var x=document.forms["myForm"]["email"].value
var atpos=x.indexOf("@");
var dotpos=x.lastIndexOf(".");
if (atpos<1 || dotpos<atpos+2 || dotpos+2>=x.length)
  {
  alert("Not a valid e-mail address");
  return false;
  }
}

</script>
</head>
<body>

<form name="myForm" action="demo_form.asp" onsubmit="return
validateForm();" method="post">
Email: <input type="text" name="email">
<input type="submit" value="Submit">
```

```
</form>

</body>
</html>
```

Mint látjuk, az oldalon megjelenik az **Email:** felirat, egy szövegbeviteli mező és a **Submit** gomb.
Ennek megnyomásakor meghívódik a **validateForm()** függvény; mely először betölti a beírt cím-adatot az **x** változóba.
Ezután definiáljuk az **atpos** és **dotpos** változókat, melyek a @ és az utolsó pont elhelyezkedési indexét kapják értékül.
Végül egy **if** feltételes rendelkezést adunk ki. Hogyha a @ hely-száma **1**-nél kisebb (azaz a legelső, **0** indexű helyen található) [**atpos<1**], vagy ha a legutolsó pont indexe a kukacénál nem nagyobb legalább kettővel [**dotpos<atpos+2**], vagy ha a legultolsó pont után nem található még legalább két karakter [**dotpos+2>x.length**], akkor a függvény felveszi a **false** értéket, és figyelmeztetést küld (**Not a valid e-mail address**). Mint látjuk, ezek a feltételek megfelelnek az **x@xx.xx** általános e-mail címnek (vagyis, mint tapasztaljuk, ez a legkevesebb karakterből álló cím, melyet a feltételes rendelkezés még elfogad (és kiír a **demo_form.asp** oldalra.

IV. JavaScript – Animáció

A JS-tel animált képeket készíthetünk.

JavaScript – Animáció

A JS segítségével animált képeket is készíthetünk, ti. úgy, hogy különféle eseményekhez megfelelő képek megjelenítését rendeljük egyetlen helyre.

A következő példában egy linkként működő képet állítunk be egy HTML-oldalon. A képhez egy **onMouseOver** és egy **onMouseOut** eseményt rendelünk, melyek egy-egy JS függvény meghívásával váltogatják a képeket.

A HTML-kód

Az ehhez szükséges HTML-kód:

```
<a href="http://www.w3schools.com" target="_blank">
<img border="0" alt="Visit W3Schools!" src="b_pink.gif" id="b1"
onmouseOver="mouseOver()" onmouseOut="mouseOut()" /></a>
```

Mint látjuk, a kép **id**-vel is rendelkezik, ami lehetővé teszi, hogy további JS-ekben is hivatkozhassunk rá.

Az **onMouseOver** esemény értelmében (lásd a JS-ben) a kép egérérintésekor a böngésző betölti a megfelelő JS-függvényt, ami a képet egy másikra cseréli.

Az **onMouseOut** esemény szerint, azaz az egérérintés megszűnésekor, ismét a böngésző meghív egy másik JS-függvényt, mely visszaállítja az eredeti képet.

A JavaScript-kód

A képek közötti váltást az alábbi JS-kóddal érhetjük el:

```
<script type="text/javascript">
function mouseOver()
{
document.getElementById("b1").src ="b_blue.gif";
}
function mouseOut()
{
document.getElementById("b1").src ="b_pink.gif";
}
</script>
```

Mint látjuk, a **mouseOver()** függvény a **b_blue.gif**, a **mouseOut()** pedig a **b_pink.gif** képet jeleníti meg a **b1** azonosítójú **<img>** elemben.

A teljes kód:

```
<html>
<head>

<script type="text/javascript">
function mouseOver()
{
document.getElementById("b1").src ="b_blue.gif";
}
function mouseOut()
{
document.getElementById("b1").src ="b_pink.gif";
}
</script>

</head>
```

```
<body>

<a href="http://www.w3schools.com" target="_blank">
<img border="0" alt="Visit W3Schools!" src="b_pink.gif" id="b1"
width="26" height="26" onmouseover="mouseOver()"
onmouseout="mouseOut()" /></a>

</body>
</html>
```

A weboldalon csupán a **http://www.w3schools.com** címre mutató, rózsaszín árnyalatú képecske (**b_pink.gif**) jelenik meg, mely egérérintésre kékre vált (**b_blue.gif**).

V. JavaScript – Image maps

Az **image map** (=kép-térkép) egy kép, melyen klikkelhető (linkként működő) területeket jelölünk ki.

HTML image maps

HTML-könyvünkből megtanultuk, hogy az image map egy link-területekkel rendelkező kép. Általában minden egyes kijelölt területhez egy-egy linket rendelünk. Ha ezekre kattintunk, a link aktivizálódik és a hivatkozott dokumentum megjelenik.

Image map kibővítése JavaScripttel

Az image map **<area>** tagjeihez esemény-attribútumokat rendelhetünk, melyek alkalmasak egy-egy JS-függvény meghívására is. Az **<area>** tag az **onClick, onDblClick, onMouseDown, onMouseUp, onMouseOver, onMouseMove, onMouseOut, onKeyPress, onKeyDown, onKeyUp, onFocus** és **onBlur** eseményeket támogatja.

Lássuk tehát image map példánkat, melyhez némi JS-et adtunk:

```
<html>
<head>

<script type="text/javascript">
function writeText(txt)
{
document.getElementById("desc").innerHTML=txt;
}
</script>

</head>
<body>

<img src ="planets.gif" width ="145" height ="126" alt="Planets"
usemap="#planetmap" />

<map name="planetmap">
<area shape ="rect" coords ="0,0,82,126"
onMouseOver="writeText('The Sun and the gas giant planets like Jupiter
are by far the largest objects in our Solar System.')"
href ="sun.htm" target ="_blank" alt="Sun" />

<area shape ="circle" coords ="90,58,3"
onMouseOver="writeText('The planet Mercury is very difficult to study
from the Earth because it is always so close to the Sun.')"
href ="mercur.htm" target ="_blank" alt="Mercury" />

<area shape ="circle" coords ="124,58,8"
onMouseOver="writeText('Until the 1960s, Venus was often considered a
twin sister to the Earth because Venus is the nearest planet to us, and
because the two planets seem to share many characteristics.')"
href ="venus.htm" target ="_blank" alt="Venus" />
</map>

<p id="desc"></p>

</body>
</html>
```

Mint látjuk, az image map összes területéhez egy-egy **onMouseOver** attribútumot, éspedig szöveg-kiíratást rendeltünk. Hogyha az egyes területek fölé visszük az egeret, ez az attribútum meghívja a fejrészbe írt

writeText() függvényt, ami **txt** helyi változójába betölti az attriútumba ugyanezen függvény paramétereként beírt bolygó-ismertető szöveget; melyet a függvény egyetle renedlkezésével a **desc** azonosítójú **<p>** elembe íratunk ki.
Az eredmény: az oldal betöltésekor megjelenik a linkekre jellemző, automatikus kék keretetezésű image-map. Hogyha ennek semleges területe fölé megyünk, nem történik smmi; de hogyha az égitesteket reprezentáló rész fölé ér egerünk, akkor a kép alatt megjelenik az erre vonatkozó ismertető szöveg (vagyis a **<p>** elem). Végül, a kijelölt területekre kattintva, új ablakban megjelenik az illető égitest fényképe.

VI. JavaScript – Időzített események

A JS végrehajtását bizonyos idő elteltéhez köthetjük; az ily módon leírt eseményeket időzítettnek nevezzük (=timing events).

JavaScript – Időzített események

A JS használatával egyes kód-részleteket meghatározott idő eltelte után is végrehajtathatuk, ilymódo ún. időzített eseményeket (=timing events) létrehozva.

Az JS-ben könnyű időzíteni az eseményeket. Erre két fő módszer szolgál:

- **setTimeout()** a kódot egy későbbi időpontban hajtatja végre
- **clearTimeout()** érvényteleníti a **setTimeout()**-ot.

A **setTimeout()** és **clearTimeout()** módszerek a HTML DOM **window** objektumára (is) vonatkoznak.

A setTimeout() módszer

Mondattana:

var *változónév*=setTimeout("*JS-rendelkezés*",*időtartam_milliszekundumban*);

A **setTimeout()** módszer egy érték képzésére („visszaadására") szolgál, melyet a **változónév** változóban tárolunk el.A **setTimeout()** beállítás e változóra való hivatkozással törölhető.

A **setTimeout()** függvény első paramétere a végrehajtandó kód, vagy az azt taratalmazó függvényt meghívó parancs. A második paraméter megmutatja, hogy a **setTimeout()** meghívásától számítva hány milliszekundum múlva induljon a beírt vagy hivatkozott kód végrehajtása.

A **setTimeout()** módszerre vonatkozó első példánkban szereplő gomb megnyomása után három másodperccel egy figyelmeztető ablak jelenik meg:

```
<html>
<head>

<script type="text/javascript">
function timeMsg()
{
var t=setTimeout("alertMsg()",3000);
}
function alertMsg()
{
alert("Hello");
}
</script>

</head>
<body>

<form>
<input type="button" value="Display alert box in 3 seconds"
onClick="timeMsg()" />
</form>

</body>
</html>
```

Mint látjuk, az oldalt egy egyetlen (**Display alert box in 3 seconds** feliratú) gombból álló **<form>** elem foglalja el. A gombra kattintva elindítjuk a **timeMsg()** függvényt.

Az utóbbi a (csak formális jelentőségű) **t** belső változó értékéül rendeli a **setTimeout()** függvényt, melynek értelmében az **alertMsg()** függvényt 3000 ms múlva végre kell hajtani.

3000 ms múlva tehát futni kezd az **alertMsg()** függvény is, ami egy **Hello** feliratú figyelmeztető ablakot nyit meg.

Második **setTimeout()**-os példánkban az időzítő-függvényt végtelen hurokként (=infinite loop), önmaga meghívására alkalmazzuk.

A gomb megnyomásakor a beviteli szövegmezőbe írt szám mintegy számlálóként, nullától kezdődően elkezd növekedni (a végtelenségig):

```
<html>
<head>

<script type="text/javascript">
var c=0;
var t;
var timer_is_on=0;

function timedCount()
{
document.getElementById('txt').value=c;
c=c+1;
t=setTimeout("timedCount()",1000);
}

function doTimer()
{
if (!timer_is_on)
  {
  timer_is_on=1;
  timedCount();
  }
else
  {
  alert("The timer is working yet!");
  }
}
</script>

</head>
<body>

<form>
<input type="button" value="Start count!" onClick="doTimer()">
<input type="text" id="txt">
</form>

<p>Click on the button above. The input field will count forever,
starting at 0.</p>

</body>
</html>
```

A „használati utasítást” tartalmazó bekezdés fölött tehát megjelenik egy **Start count!** feliratú gomb, mellette egy üres szövegmezővel. Ebbe próbaképpen beírhatunk valamit (pl. a nevünket), de ha üresen hagyjuk, az sem baj, mert mindkét esetben ugyanaz fog történni a gomb megnyomására.
Ekkor ugynis az **onClick** attribútum meghívja a **doTimer()** függvényt.
Mint látjuk, a fejrészben először definiáljuk a **c**, **t** és **timer_is_on** változókat.

A gomb megnyomásakor tehát a **doTimer()** elindít egy **if** feltételes rendelkezést.
Ez mindenekelőtt megvizsgálja a **timer_is_on** változó logikai értékét. Hogyha az NEM (vagyis **timer_is_on=0=false=!timer_is_on**), akkor futtatja a függvény tartalmát, vagyis a **timer_is_on** változó értékét átállítja **1**-re, és meghívja a **timedCount()** függvényt. Ellenkező esetben (**else**), vagyis hogyha a **timer_is_on** változó logikai értéke **1=IGAZ** volt, nem indítja el újra a **timedCount()**-ot, hanem figyelmeztető üzenetet küld (**The timer is working yet!**). Amig ezt le nem **OK**-zzuk, addig a számláló áll.

A számlálót a **timedCount()** függvény indítja el, mely először kiíratja a **txt** azonosítójú szövegbeviteli mezőbe a **c** változó értékét (**0**) – ekkor az oda korábban beírt szöveg eltűnik. Ezután a függvény a **c** értékét megnöveli eggyel, majd újra meghívja önmagát, 1000ms-os, azaz egy másodperces késleltetéssel. Az eredmény: a **c** értéke minden egyes meghíváskor eggyel nő, és ezt az új értéket e függvény a következő meghíváskor kiírja a szövegmezőbe.

Mindebből következik, hogyha a **timer_is_on** változó kezdeti értékét **0** vagy **false** helyett **1**-nek vagy **true**-nak vesszük, akkor gombnyomásra nem indul el a számláló, hanem megjelenik a **The timer is working yet!** figyelmeztetés.
Ugyanígy **c** megváltoztatásával a számlálás kezdeti értékét, a **timedCount()** parancssorának átírásával pedig a számlálás lépéseit és periódusidejét változtathatjuk meg.
Mint látjuk, ugyanezen parancssorban az érték megadásával újradefiniáljuk a **t** változót, így annak korábbi, külön (külső változóként való) megadása szükségtelen, mert csak a **timedCount()** függvényen belül hivatkozunk rá.
Hogyha pedig kiiktatjuk a **doTimer()** ellenőrző-függvényt, és az **onClick** attribútumból közvetlenül a **timedCount()**-ot hívjuk meg, akkor minden egyes gombnyomáskor gyorsulni fog a számláló (feltéve, hogy nem a korábbi gombnyomásoktól számított egész másodperckor nyomunk), mivel így újabb és újabb időpontoktól kezdve ciklusonként egyre nagyobb és nagyobb hozzátétek keletkeznek a **c** változóban.

A clearTimeout() módszer
Mondattana:
clearTimeout(*setTimeout()-változó*)

Hogy ez mit jelent, azt az alábbi példán keresztül értjük meg.

Példánkban az előbbi végtelen ciklust használjuk fel, azzal a különbséggel, hogy egy **Stop count!** gombot is hozzáadunk az oldalhoz, a számlálás leállítására:

```
<html>
<head>

<script type="text/javascript">
var c=0;
var t;
var timer_is_on=0;

function timedCount()
{
document.getElementById('txt').value=c;
c=c+1;
t=setTimeout("timedCount()",1000);
}

function doTimer()
{
if (!timer_is_on)
  {
  timer_is_on=1;
  timedCount();
```

```
  }
else
  {
  alert("The timer is working yet!");
  }
}

function stopCount()
{
clearTimeout(t);
timer_is_on=0;
}
</script>

</head>
<body>

<form>
<input type="button" value="Start count!" onclick="doTimer()" />
<input type="text" id="txt" />
<input type="button" value="Stop count!" onclick="stopCount()" />
</form>

<p>
Click on the "Start count!" button above to start the timer. The input
field will count forever, starting at 0. Click on the "Stop count!"
button to stop the counting. Click on the "Start count!" button to
start the timer again.
</p>

</body>
</html>
```

Mint látjuk, a **<form>** elem beviteli mezőjének **Start count!** gombja itt is meghívja a végtelen számálás függvényeit, melyek az előbbiekhez telejsen hasonlóan működnek.
Hogyha azonban a **Stop count!** gombot megnyomjuk, meghívódik a **stopCount()** függvény, amely először kitörli a **t** változó értékét, azaz leállítja a **timedCount()** függvény újbóli meghívogatását, és ezzel a számlálást. Majd pedig a **timer_is_on** logikai változót visszaállítja **0=false** értrékre, miáltal a **Start count!** gombbal újra elindítható a számlálás (nem jelenik meg a **The timer is working yet!** figyelmeztetés). Így tehát egy stopperórát készítettünk, amit az egyik gomb megnyomásával elindíthatuink, a másikkal pedig leállíthatunk. A megjelenő szám az elteltz másodperceket jelenti. Mint a megelőző példában, a **Start count!** gomb többszöri egymás utáni megnyomása figyelmeztetést von maga után (mialatt a számláló leáll); viszont a **Stop count!** gomb nyomása után ez a figyelmeztetés elmarad, hiszen a **timer_is_on** váltzozó értékét a **stopCount()** függvény visszaállította HAMIS-ra; így a **doTimer()** függvény feltételes rendelkezésének első (**if** alatti) szakasza lép életbe.

A **Start count!** gomb többszöri megnyomásához társított figyelmeztetéshez hasonlót a **Stop count!**-ra nézve is beállíthatunk,. Hogyha a **stopCount()** függvény rendelkezéseit a **timer_is_on** változó IGAZ vagy HAMIS állapotához kötjük:

```
<html>
<head>

<script type="text/javascript">
var c=0;
var t;
var timer_is_on=0;
```

```
function timedCount()
{
document.getElementById('txt').value=c;
c=c+1;
t=setTimeout("timedCount()",1000);
}

function doTimer()
{
if (!timer_is_on)
  {
  timer_is_on=1;
  timedCount();
  }
else
  {
  alert("The timer is working yet!");
  }
}

function stopCount()
{
if (timer_is_on==true)
  {
  clearTimeout(t);
  timer_is_on=0;
  }
else
  {
  alert("You have already stopped the timer!")
  }
}

</script>

</head>
<body>

<form>
<input type="button" value="Start count!" onclick="doTimer()" />
<input type="text" id="txt" />
<input type="button" value="Stop count!" onclick="stopCount()" />
</form>

<p>
Click on the "Start count!" button above to start the timer. The input
field will count forever, starting at 0. Click on the "Stop count!"
button to stop the counting. Click on the "Start count!" button to
start the timer again.
</p>

</body>
</html>
```

Mint látjuk, ha a **stopCount()** függvényt a számláló álló állapota mellett nyomjuk meg, akkor megjelenik a **You have already stopper the timer!** figyelmeztetés. Hogyha ezt még a számlálás elindítása előtt tesszükmeg, akkor is megjelenik, hiszen a **timer_is_on** változó értéke ekkor is **0**.
Figyelemreméltó még, hogy míg az előző példában a script második sorában definiált, és a **timedCount()** függvényben értékkel újradefiniált **t** változó itt már külső, két függvényre is vonatkozó szereppel bír. Ez azt jelenti, hogy az előző példával ellentétben, ha kitöröljük a script második sorából, akkor a scriptnek nem volna szabad működnie. Azonban, mint tapasztalom, mégis működik. Mindenesetre a **t** változó külső definiálása ebben az esetben valóban indokoltabb, mint az előzőben, bár, úgy látszik, itt is elhagyható.
Azt hiszem, ennek kulcsa abban rejlik, hogy a **stopCount()** függvény **if** alatti rendelkezls-opcióját, ahol a **t** másodszor szerepel, csak azután hajthatja végre a program, miután a **timedCount()** már meghívásra került; így a **t** változót ott az értékével [**=setTimeout(”timedCount()”,1000)**] definiáltuk, és nem volt más fügvény, ami felülírta volna. Mint azonban láttuk, habár ekkor belső változóként definiáljuk, a második függvény, mint külső változót, képes törölni [**clearTimeout(t);**] a parancssor úbvóli futtatására nézve.

A **clearTimeout()** függvénnyel tehát leállíthatjuk azon **setTimeout()** függvények futását, melyeket a hozzájuk tartozó változó révén a paraméterben megjelöltünk [pl. itt a **t=setTimeout(”timedCount()”,1000)**-ot].

További példák
Első példánkban egy másik egyszerű időzítési módot mutatunk be:

```
<html>
<head>

<script type="text/javascript">
function timedText()
{
var t1=setTimeout("document.getElementById('txt').value='2
seconds!'",2000);
var t2=setTimeout("document.getElementById('txt').value='4
seconds!'",4000);
var t3=setTimeout("document.getElementById('txt').value='6
seconds!'",6000);
}
</script>

</head>
<body>

<form>
<input type="button" value="Display timed text!" onclick="timedText()"
/>
<input type="text" id="txt" />
</form>

<p>Click on the button above. The input field will tell you when two,
four, and six seconds have passed.</p>

</body>
</html>
```

Mint látjuk, a weboldalon kezdetben egy **Display timed text!** feliratú gombot, egy szövegbeviteli mezőt és egy bekezdést jelenítünk meg.
Hogyha megnyomjuk a gombot, annak **onclick** attribútuma meghívja a **timedText()** fügvényt.
Ez először a **t1** formális változó értékeként, a **setTimeout()** függvénnyel kiíratja a **txt** azonosítójú szövegmezőbe a **2 seconds!** feliratot, a parancs kiadásától számított második másodpercben. Vagyis a gomb megnyomása után két másodperccel megjelenik a szöveg. Ugyanígy a gombnyomától négy és hat

másoodperc múlva a **t2** és **t3** változókba írt **setTimeout()** függvények felülírják a szövegmező tartalmát (**4 seconds!** ill. **6 seconds!**).
Ezután nem változik a helyzet; hogyha azonban a gombot mégegyszer megnyomjuk, akor 2000ms elteltével a felirat ismét **2 seconds!**-ra vált, azután **4**-re, stb.. Hogyha valamit beírtunk eredetileg a szövegmezőbe, az a legutolsó, **6**-os kijelzéshez hasonlóan, a gomb megnyomása után **2** másodperccel eltűnik, átadva helyét az első (**t1** szerinti) feliratnak.

Második példánkban egy digitális órát készítünk időzített események segítségével:

```
<html>
<head>

<script type="text/javascript">
function startTime()
{
var today=new Date();
var h=today.getHours();
var m=today.getMinutes();
var s=today.getSeconds();
// add a zero in front of numbers<10
m=checkTime(m);
s=checkTime(s);
document.getElementById('txt').innerHTML=h+":"+m+":"+s;
t=setTimeout('startTime()',500);
}

function checkTime(i)
{
if (i<10)
  {
  i="0" + i;
  }
return i;
}
</script>

</head>
<body onload="startTime()">

<div id="txt"></div>

</body>
</html>
```

Mint látjuk, az oldal betöltődésekor meghívásra kerül a **startTime()** függvény.
Ez először a **today** változó értékének megadja az aktuális dátumot, majd a **h**, **m** és **s** változókba kiíratja annak óráit, perceit és másodperceit. Ezután az **m** és **s** változókra nézve futtatja a **checkTime()** függvényt, ami ezek **10**-nél kisebb értékei esetén az értékük elé egy-egy **0**-át illeszt be.
Az így összeállított óra-, perc- és másodperc-értékeket azután a **startTime()**-beli parancssor digitálisóra-formátumban kiíratja a **txt** azonosítójú ** ** elembe, a szövegtestbe.
A **setTimeout()** függvény pedig minden fél másodpercben újra meghívja a **startTime()** függvényt, vagyis önmagát is.

Íme, egy próbálkozás az óra esetlegesen egy-számjeyűvé tételére és a kétszeri gombnyomások kiíratására (egyelőre befejezetlen):

```
<html>
<head>
```

```
<script type="text/javascript">
var clock_is_on=0;

function selector()
{
if (clock_is_on==0)
  {
  function startTime();
  }
else
  {
  function alert();
  }
}

function startTime()
{
var clock_is_on=1;
var today=new Date();
var h=today.getHours();
var m=today.getMinutes();
var s=today.getSeconds();
document.getElementById('txt').innerHTML=h+":"+m+":"+s;
t=setTimeout('startTime()',500);
}

function alert()
  {
  alert("The clock's already running!");
  }
}

function stopClock()
{
if (clock_is_on==true)
  {
  clearTimeout(t);
  clock_is_on=0;
  }
else
  {
  alert("You have already stopped the clock!");
  }
}

</script>

</head>
<body>

<form>
<input type="button" value="Start clock!" onclick="selector()" />
<input type="button" value="Stop clock!" onclick="stopClock()" />
</form>
```

```
<div id="txt"></div>

</body>
</html>
```

25 perces időzítőt készíteni (JS-vizsgafeladat!)

VII. JavaScript – Saját objektumok beállítása

A JS-objektumok az JS-ben tárolt információk (pl. rendelkezések) rendszerezésére szolgálnak.

Néhány gyors példa

Első példánkban közvetlenül hivatkozunk egy saját objektumunkra:

```
<html>
<body>

<script type="text/javascript">
personObj=new Object();
personObj.firstname="John";
personObj.lastname="Doe";
personObj.age=50;
personObj.eyecolor="blue";

document.write(personObj.firstname + " is " + personObj.age + " years
old.");
</script>

</body>
</html>
```

Mint látjuk, először definiáljuk a **personObj** nevű, saját objektumunkat, majd ezen objektumon belül különféle változókat (**firstname, lastname, age, eyecolor**) definiálunk.
Ezután pedig kiíratunk egy (szöveges) string-et, mely a **personObj.firstname** és a **personObj.age** változókra mutató közvetlen hivatkozást tartalmaz. A kimenet:
John is 50 years old.

Hogyha egyszerűen csak a **firstname** és **age** változókra hivatkozunk:

```
document.write(firstname + " is " + age + " years old.");
```

akkor a böngésző nem ír ki semmit; tehát az objektumon belüli változók csak közvetlenül (azaz az objektum nevével együtt) hivatkozhatók.

Második példánkban egy objektum-sablont (=object template) mutatunk be:

```
<html>
<body>

<script type="text/javascript">
function person(firstname,lastname,age,eyecolor)
{
this.firstname=firstname;
this.lastname=lastname;
this.age=age;
this.eyecolor=eyecolor;
}

myFather=new person("John","Doe",50,"blue");

document.write(myFather.firstname + " is " + myFather.age + " years
old.");
</script>

</body>
</html>
```

Mint látjuk, először definiáljuk a **person** függvényt, aminek paraméterei a függvényben meghatározott változókhoz lesznek értékként hozzárendelve. Ezeket a hozzárendeléseket tartalmazza a függvénybe írt parancssor.
A függvény alatti sorban definiáljuk a **nyFather** objektumot, mint a **person** függvény ("John","Doe",50,"blue") paraméterekre vonatkozó értékét. Ekkor a **person** függvény a parancssorába írt változókhoz hozzárendeli az előbbi paraméterként megadott értékeket, majd ezeket a változókat a **myFather** objektumba tartozónak tekinti.
Ezt mutatja az utolsó, **document.write** rendelkezés eredménye is, ahol a **myFather** objektum-névvel hivatkozunk a **person** függvény változó-értékeire, melyeket a **myFather** defifiálásakor paraméterként adtunk meg. A megjelenő szöveg:
John is 50 years old.

JavaScript – objektumok
Mint ismertetőnk korábbi szakaszaiban láttuk, a JS számos beépített (alapértelmezett) (pl. **string**, **date**, **array** stb.) objektummal rendelkezik. Ezeken túl azonban mi is definiálhatunk objektumokat.

Az objektum egy speciális adat-fajta, melyeket jellemzők és módszerek segítségével kezelhetünk.

Vegyük például azt az esetet, mikor egy személyt (ill. a rá vonatkozó adatokat) akarunk objektumként kezelni! A személy jellemzői, vagyis a név, magasság, tömeg, életkor, bőr- és szem-szín stb. emberről emberre változók. Ezek egyúttal tehát megfeleltethetők JS-változóknak is, melyek az objektum, vagyis az adott személy jellemzésére valók. Az objektumokra módszerek is vonatkoznak, melyek alatt az objektumokon végrehajtható akciókat értjük. Az ember akciói lehetnek pl: evés(), alvás(), munka(), játék() stb.

Jellemzők = változók
Egy JS-objektum valamely jellemzőjére/változójára való hivatkozás mondattana:
objektumNév.jellemzőNév

Az objektumok jellemzőit egyszerűen az értékük megadásával definiáljuk (a közönséges változókhoz hasonlóan). Hogyha tehát pl. a **personObj** objektumot már létrehoztuk, ahhoz a **firstname, lastname, age** és **eyecolor** jellemzőket/változó-értékeket a következőképpen rendelhetjük hozzá:

```
personObj.firstname="John";
personObj.lastname="Doe";
personObj.age=30;
personObj.eyecolor="blue";

document.write(personObj.firstname);
```

Mint látható, a fenti kód kimenete a
John
felirat lesz.

Módszerek = függvények
Az objektumok kezelésére módszereket, azaz bizonyos rendelkezésekből és az azok végrehajtásakor felhasználandó paraméterekből felépülő függvényeket definiálhatunk.

A módszerek/függvények meghívásának mondattana:
***obejektumNév.módszerNév*()**

A módszer végrehajtásához szükséges paramétereket (illetve az azokat szolgáltató változókat) a zárójelbe írjuk.

Az előbbi mondattan szerint tehát pl. a **sleep()** függvényt következő kóddal hajthatjuk végre („hívhatjuk meg") a **personObj** objektumon:

```
personObj.sleep();
```

Saját objektumok készítése
Új objektumok definiálására két lehetőségünk is van:

1. Objektum közvetlen beállítása
A következő kóddal közvetlenül definiálhatjuk az objektumot és annak négy jellemzőjét:
```
personObj=new Object();
personObj.firstname="John";
personObj.lastname="Doe";
personObj.age=50;
personObj.eyecolor="blue";
```

Ugyancsak egyszerűen rendelhetünk módszereket a **personObj** objektumhoz; pl. a következő kóddal az **eat()** módszert rendeljük hozzá:
```
personObj.eat=eat;
```

2. Objektum létrehozása konstruktorral
A következő kódban szereplő függvény egy objektumot hoz létre, azaz konstruktorként (=constructor) viselkedik:
```
function person(firstname,lastname,age,eyecolor)
{
this.firstname=firstname;
this.lastname=lastname;
this.age=age;
this.eyecolor=eyecolor;
}
```
A függvényen belül különböző dolgokat rendelhetünk a **this.*jellemzőNév*** általános alakú jellemzőkhöz. Az ezek elnevezésében szereplő **this** megjelölés azért szükséges, mert a **person** objektumot egyszerre több emberre is vonatkozathatjuk, így az egyes jellemzők mindig az éppen tárgyalt személyre vonatkoznak majd. Hogy kire, azt mindig pontosan meg kell adnunk.

Hogyha az objektum-konstruktorként működő függvényünk (itt **person**) készen áll, akkor elkezdhetünk objektumokat készíttetni vele; amikoris a készítendő objektum nevét és jellemzőit kell változó-névként ill. konstruktor-függvény-paraméterekként beírnunk, pl.:
```
var myFather=new person("John","Doe",50,"blue");
var myMother=new person("Sally","Rally",48,"green");
```

A konstruktor-függvényen belül módszereket is hozzárendelhetünk az objektumhoz, pl.:
```
function person(firstname,lastname,age,eyecolor)
{
this.firstname=firstname;
this.lastname=lastname;
this.age=age;
this.eyecolor=eyecolor;

this.newlastname=newlastname;
}
```
Mint tudjuk, a módszerek az objektumokhoz kapcsolt függvények; így már csak fel kell írnunk a **newlastname()** függvényt:
```
function newlastname(new_lastname)
{
this.lastname=new_lastname;
}
```

A **newlastname()** függvény definiálja egy új vezetéknevet (=last name), és hozzárendeli az adott személyhez. A JS a **this** használata révén felismeri, melyik személyről van szó, így fel is írhatjuk: `myMother.newlastname("Doe")`

VIII. JavaScript – Összefoglalás

JavaScript – Összefoglalás

A fentiek során megtanultuk, hogyan lehet a HTML-oldalakat a JS-tel dinamikusabbá és interaktívabbá tenni. Eseményekre való reagálásként, elétrő helyzet-variánsok kezelésére, űrlapok ellenőrzésére stb. különféle rendelkezéseket írtunk. Elsajátítottuk a JS beépített objektumainak használatát, valamint újak készítését és felhasználását is.

További információkat JavaScript példáink között és JavaScript kézkönyvünkben találhatunk.