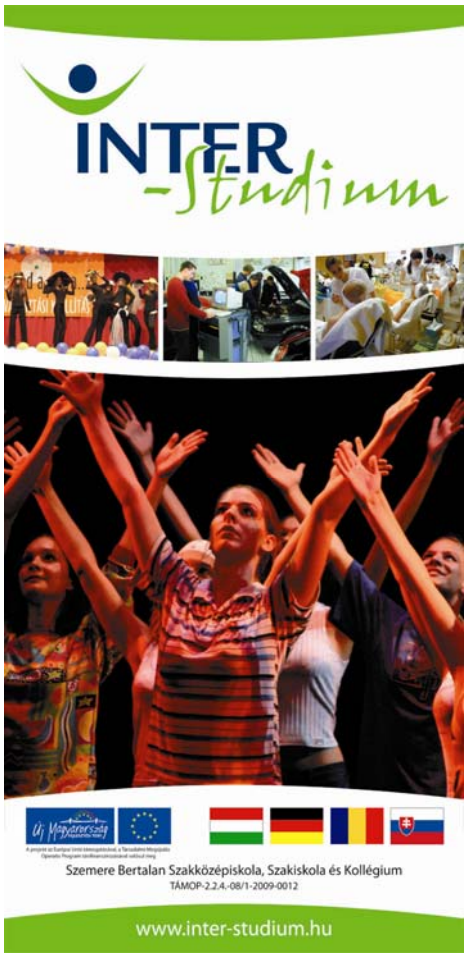
Webszerkesztés, a web programozás alapjai

2. modul

CSS és JavaScript programozás

Az egész életen át tartó tanulás fejlesztése az intézmények közötti nemzetközi együttműködéssel

TÁMOP-2.2.4.-08/1-2009-0012



Szemere Bertalan Szakközépiskola, Szakiskola és Kollégium

Szerkesztette: Ember László

Lektorálta: Molnár Gábor

A kiadvány az „INTER-STUDIUM - Az egész életen át tartó tanulás fejlesztése az intézmények közötti nemzetközi együttműködéssel” című, TÁMOP-2.2.4.-08/1-2009-0012 számú projekt keretében készült.

A projekt az Európai Unió támogatásával, a Társadalmi Megújulás Operatív Program társfinanszírozásával valósul meg

- 2010 -

TARTALOMJEGYZÉK

1. Mi a CSS?

- A CSS a Cascading Style Sheet rövidítése, mely annyit tesz 'Egymásba ágyazott stíluslapok'
- A CSS a weblapok (vagy hasonlóan strukturált HTML, XHTML dokumentumok) stílusának, kinézetének meghatározására szolgáló stílusleíró nyelv
- A CSS egy szabvány melynek specifikációját a W3C konzorcium felügyeli

1.1 Története

A CSS létrejöttének oka a HTML leíró nyelv hiányossága volt, miszerint a HTML főleg a dokumentum tartalmának leírásáért volt felelős, annak formázásáért már kevésbé. A növekvő igények miatt a HTML 3.2-es verziójába már beépültek formázó TAG-ek, de ezek használata bonyodalmas volt és semmiképp nem célratörő. Szükség volt egy a HTML dokumentumtól független formázási lehetőségre, melyekre több megoldás is született. A legjobb megoldásnak mégis a CSS tűnt.

Az első verzió előkészületei 1994-1996 közé tehetők. Ezeket főleg két ember végezte, név szerint Håkon Wium Lie és Bert Bos. Az ő vezetésükkel 1996 decemberében megjelent a CSS1 azaz az első hivatalos verzió. 1998-ban a második verzió is elkészült CSS2 néven. A CSS3 azaz a harmadik verzió fejlesztései pedig még a napokban is folynak.

A CSS olyannyira felülkerekedett a HTML formázási lehetőségein hogy a HTML 4.0 szabványból már ki is kerültek ezek a formázást segítő TAG-ek. Így például a jól megszokott FONT tag is érvénytelenített státuszú lett.

1.2 Feladata

A CSS segítségével a következő feladatokat oldhatjuk meg

- Az elkészített stílusokat egy állományba tudjuk összegyűjteni (.css)
- Az így elkészített stílusokkal több különböző weblapot tudunk felruházni (egy CSS több weblap)
- Egy elkészített weboldal többfajta megjelenését tudjuk prezentálni egyetlen hivatkozás megváltoztatásával (egy weblap több CSS)
- Tudunk igazodni a weboldal megnyitását végző kliens számítógép paramétereihez (pl. felbontás, böngésző, nyelv)
- Interaktívabbá tudjuk tenni a már meglévő weblapjainkat az egér és billentyűzet eseményekre történő stílusváltoztatással
- Sok időt tudunk megtakarítani, mivel a formázásokat változtatás esetén jó esetben csak egy helyen kell módosítani

1.3 Formázási hierarchia

A stílusok elkészítésénél és a formázásnál azokat a hierarchiákat kell figyelembe vennünk melyek a HTML dokumentumra egyébként is érvényesek. Minden egyes elem valamely szülőtől származik és ezek az elemek örökölhetik a szülők stílusát, formázását.

pl.: `<body><h1><p></p></h1></body>`

A <P> bekezdés mint látjuk a hierarchia alján található és szülőeleme a <H1> címsor és a <BODY> is azaz maga a dokumentum.

Ezektől a hierarchiáktól elérni egyedi azonosítók, jelölők, és osztályok segítségével lehet a CSS-ben. Ezek a későbbiekben kifejtésre kerülnek.

1.4 A HTML korlátai, és azok átlépése CSS segítségével

A HTML leíró nyelv határa lényegében ott található, amikor is a weboldalunk tartalmát már felépítettük, és az abban található elemek alapvető sorrendjét, hierarchiáját beállítottuk. A szülő gyermek kapcsolatokat kialakítottuk.
(pl. `...<table><tr><td><img></td></tr></table>...`)

A HTML dokumentumok készítése során, a formázásra nem sok lehetőség van. Az úgynevezett HTML elemek egyes tulajdonságai bár állíthatóak, és ezek segítségével tudjuk módosítani a weboldal kinézetét, de ezek tárháza szegényes, és olykor az igények ezen bőven túlmutatnak.

Ilyen HTML alapú formázási lehetőség például a rendezés (`ALIGN`) mely sok elemhez meghatározható tulajdonság. Ezek segítségével az elemek balra, jobbra, középre, esetleg sorkizárttá rendezhetők. Azonban ennek segítségével se tudjuk az elemeket bizonyos mértékben balra vagy jobbra helyezni az oldalon.

Pl.: `left:50px;`

Vagy ugyanígy járhatunk az egyes elemek külső és belső margóinak beállításával, melyhez a HTML nyelvben valljuk be kevés lehetőséget kapunk, azonban a CSS segítségével szinte az összes elemre beállíthatjuk ezeket a tulajdonságokat.

Pl.: `padding-bottom:5px;`

A már említett FONT elem tulajdonságai például kimerülnek a betűtípus, a méret és a szín tulajdonságok beállítási lehetőségével. A CSS a szövegek formázására ennél jóval több lehetőséget kínál.

Pl. a kiskapitális szöveg: `font-variant:small-caps;`

CSS segítségével még a jól megszokott balról-jobbra, fentről-lefelé megjelenési struktúrát is megbonthatjuk az egyes elemek pozíciójának konkrét megadásával, vagy a rétegek kezelésével, és ezáltal egyedi dizájnt alakíthatunk ki.

Pl.: `position: absolute;`

2. A CSS létrehozásának módjai

Három féle módon illeszthetünk stílust a weboldalunkhoz. Ezeket a stíluslapokat különböző néven látták el, melyekre ezeken a neveken fogunk hivatkozni:

- Inline Style– elembe ágyazott, vagy más néven egy elemhez rendelt stílus *egy konkrét HTML elemhez rendeljük hozzá a HTML kódban*
- Internal Stylesheet– belső vagy beágyazott stíluslap *a HTML kód fejlécében (HEAD) helyezzük el*
- External Stylesheet– külső stíluslap *egy külső állományban tároljuk, melyre hivatkozunk a HTML kódban*

Ezek prioritása a fenti lista szerint épül fel, tehát az Inline stílusnak van a legnagyobb prioritása, és az External stíluslapnak van a legkisebb prioritása a három lehetőség közül.

2.1 External Stylesheet létrehozásának módja

Létrehozzuk a stílusokat és formázásokat tartalmazó állományt, melyet *.CSS* kiterjesztéssel mentünk el. Erre az állományra fogunk hivatkozni a HTML kódjában a következőképpen:

```
<link rel=”stylesheet” type=”text/css” href=”allomany.css”>
```

Ezt a hivatkozást a HTML kód fejlécében, tehát a HEAD részében kell, hogy elhelyezzük. A három paraméter jelentése:

- REL: a két dokumentum közti kapcsolatot írja le. Jelen esetünkben ez külső stíluslapot jelöl.
- TYPE: a csatolt külső állomány MIME típusa. Jelen esetünkben ez *text/css*, mely szövegfájlra utal.
- HREF: a külső állomány elérési útja. Jelen esetünkben ez az *allomany.css* mely a weblapunkkal megegyező mappában helyezkedik el, mivel nem adtunk meg az eléréshez szükséges más hivatkozást, és így relatív hivatkozásként értelmeződik. Ha az állományunk pl. egy mappával fentebb helyezkedik el akkor hivatkozhatunk rá a *href=”../allomany.css”* paraméterrel.

Ha olyan állományra hivatkozunk, mely nem található a HTML nem fog hibát üzenni,a lap betöltődik és stílusok hiányában a lap eredeti formátumában fog megjelenni.

A CSS állományban helyezzük el a formázáshoz szükséges kiválasztókat, és a hozzájuk tartozó tulajdonságokat, melyeknek szintaktikáját a későbbiekben kifejtjük, de lássunk egy példát:

```
td{
     padding-left:10px;
     margin-left:5px;
     background-color:#5656FF;
}
```

Ilyen és ehhez hasonló elemeket fog tartalmazni a CSS állományunk.

2.2 Internal Stylesheet létrehozásának módja

Az Internal Stylesheet abban különbözik az előző megoldástól, hogy a fenti példához hasonló elemeket nem egy külső állományban helyezzük el, hanem a HTML oldalunk fejlécébe (HEAD) írjuk.

A fejlécbe írt beágyazott stíluslapokat a <STYLE> és a </STYLE> HTML elemek közé kell írnunk a következőképpen:

```
<HEAD>
…
<STYLE>
  td{
       padding-left:10px;
       margin-left:5px;
       background-color:#5656FF;
  }
</STYLE>
…
</HEAD>
```

A <STYLE> tagok közé írt elemek szintaktikája megegyezik az External Stylesheet használatakor kötelező szintaktikával, tehát a CSS nyelv szintaktikájával.

Arra figyeljünk, hogy ha hivatkozunk külső stíluslapra is, és belső stíluslapot is használunk, akkor a <LINK> elemet a <STYLE> elé helyezzük el, különben a külső stíluslap felülbírálja a belső stílusainkat.

Azonban normál esetben, ha használjuk mindkét lehetőséget az egyes objektumok tulajdonságai a következőképp alakulnak:

- A külső stíluslapon megjelölt, de a belsőn nem megjelölt tulajdonságok öröklődnek
- A külső és a belső stíluslapon is megjelölt tulajdonságok közül a belső stíluslapon megjelölt tulajdonság lesz az érvényes
- A belső stíluslapon megjelölt, de a külső stíluslapon nem említett tulajdonság egyértelműen a belső stíluslapon megjelölt formában lesz érvényes

3. Inline style létrehozásának módja

Ahogy azt említettük, a legmeghatározóbb és prioritásban is elől álló típusa CSS-ben az Inline style. Az egyes HTML elemekhez konkrétan feltüntetett formázási tulajdonságok felülbírálják a külső vagy belső stíluslapon meghatározottakat.

Használata:

Az egyes HTML elemek paraméterlistájának valamely részében feltüntetjük a STYLE kulcsszót melynek értékeként soroljuk fel az adott elemhez tartozó tulajdonságokat és azok értékeit. Ügyeljünk arra, hogy a felsorolt tulajdonságokat és értékeket továbbra is a szintaktika szabályainak megfelelően kell feltüntetnünk, még akkor is, ha jelen esetben egy sorba írjuk az összest.

Példa:

```
<TD COLSPAN=”2” STYLE=”padding:40px;border:1px solid red;”>
```

A fenti példában látszik, hogy a TD elemhez tartozó HTML attribútumok használhatóak továbbra is az Inline style használata mellett. Jelen esetben pl. a COLSPAN cellaegyesítés. A STYLE attribútum után az idézőjelek közé soroljuk fel a kívánt tulajdonságokat. Arra figyeljünk, hogy ha használunk a CSS leiratában is idézőjelet, akkor más típust használjunk, mint amit a STYLE paraméternél használtunk.

Pl.:

```
<TD COLSPAN=”2” STYLE=”background-image:URL(’kutya.jpg’)”>
```

4. A CSS szintaxisa, kijelölők, tulajdonságok, értékek

A CSS szintaktikai szabálya a következő módon épül fel:

Szükségünk van egy úgynevezett SELECTOR-ra, magyarul szelektor vagy kijelölő. Ez határozza meg mely elemekre fognak érvényesülni a felsorolt tulajdonságok. Ezek a kijelölők sokféleképpen épülhetnek fel, melyeket később sorolunk fel.

A kijelölő után kapcsos zárójelek { } között soroljuk fel a tulajdonság-érték párosokat, mindegyiket egyenként pontosvesszővel lezárva.

Ez a szintaxis a beágyazott és a külső stíluslapra vonatkozik. Az Inline stílusnál a szelektorok és a kapcsos zárójelek nem használhatók.

Általános példa:

```
SZELEKTOR {Tulajdonság1:érték; Tulajdonság2:érték;}
```

Konkrét példa:

```
H3 {color:yellow; background-color:#444F55;}
```

Ezeket a blokkokat természetesen lehet, és érdemes is a szerkesztés során úgy formázni, hogy az olvashatóbb, átláthatóbb legyen.

```
H3
{
     color:yellow;
     background-color:#444F55;
}
```

A kijelölőket megadhatjuk felsorolásként is, azaz egy blokkhoz tartozhat több kijelölő is. Ezeket vesszővel elválasztva kell megadnunk.

```
H3, P, H1
{
     font-size:11px;
}
```

A tulajdonságok mindegyike beépített tulajdonság, melyek a nyelv részei. Ismeretük szükséges a formázások elvégzéséhez, azonban a tulajdonságok legtöbbször az angol megfelelőjükkel azonos, ezért elsajátításuk könnyű.

Minden tulajdonsághoz meghatározott értékek tartozhatnak.

Az értékek a következő típussal rendelkezhetnek:

- beépített érték, konstans
  - színek (red, blue, yellow, stb.)
  - rendezések (center, right, left, stb.)
  - pozíciók (absolute, relative, stb.)
  - szegélyek (dashed, solid, dotted, stb.)
  - stb.
- numerikus érték
  - font-weight:300;

- numerikus érték konkrét mértékegységgel *(itt ügyeljünk arra, hogy az érték és a mértékegység között ne legyen szóköz!)*
  - `left:50px;`
  - `font-size:2.5em;`
  - `width:50%;`
- szöveges érték
  - `background-image:url('paper.gif');`
- színkód
  - `color:#f546f5;`

Vannak esetek, amikor egyes tulajdonságokhoz több értéket is rendelhetünk azokat szóközzel elválasztva. Ilyen pl. mikor a keretek meghatározásakor egyszerre adhatjuk meg azok méretét típusát és színét:

```
border:1px solid black;
```

Vagy ilyen például mikor a felső, jobb oldali, alsó, és bal oldali margókat egyszerre határozzuk meg egy elemre:

```
margin:10px 20px 30px 40px;
```

Ezeknél a lehetőségeknél ügyelnünk kell arra, hogy a sorrend kötött. A CSS nem képes felismerni az értékek felsorolásakor azok rendeltetését.

A szintaktikai szabályokhoz tartozik még a COMMENT-ek, azaz a megjegyzések elhelyezésére vonatkozó szabályok. A CSS-ben a /* */ jelölők közé írva helyezhetünk el megjegyzést a szövegben. És ahogy a példa is mutatja, egyes tulajdonság-érték párosok is így elrejthetők a böngészők számára.

```
TD
{
     /*ez egy cella formázása*/
     background-color:green;
     /* color:red;*/
}
```

5. Szelektorok, kijelölők

Hogy az egyes blokkok tulajdonságai mely HTML elemekre vonatkoznak, azokat a kijelölőkkel tudjuk meghatározni. Ezekre a következő megoldások léteznek:

5.1 Elem neve alapján történő meghatározás

Konkrétan megjelöljük, azt a HTML elemet melyre a tulajdonságokat akarjuk beállítani. Ekkor az összes, a dokumentumban található hasonló elemre érvényesül a formázás.

```
TD{color:green;}   vagy   TABLE{margin:0px;}
```

5.2 ID (azonosító) alapján történő meghatározás

A HTML kódban az egyes elemek tagjaiban elhelyezhetünk egy ID attribútumot melynek értékére később a CSS állományban hivatkozunk a # jel segítségével. A szintaxis a következő:

HTML kód: `<TABLE ID="fontos">`
CSS kód: `#fontos{margin:1px;}`

Ekkor a formázás csak a 'fontos' azonosítóval ellátott elemekre lesz érvényes. Példánkban a feltüntetett TABLE elemre.

Habár az ID alapú kijelölést egyedi esetekre szoktuk használni, amennyiben ez az azonosító több elemre is be van állítva, úgy azokra mindre érvényes lesz a formázás.

5.3 CLASS (osztály) alapján történő meghatározás

Akárcsak az ID alapú kijelölésnél, itt is a HTML kódban helyezünk el egy jelölőt. Az elem attribútumai közé írunk egy CLASS attribútumot melynek értékére a CSS állományban hivatkozhatunk a 'pont' írásjel segítségével a következő módon:

HTML kód: `<H3 CLASS="szines">`
CSS kód: `.szines{color:red;}`

A formázás ekkor minden olyan elemre érvényes lesz melynek osztálya `'szines'`, azaz attribútumai között megtalálható a `CLASS='színes'`. Példánkban egy H3 címsort ruháztunk fel az osztály tulajdonsággal, de ha a dokumentumban található más elem hasonló osztály elnevezéssel (pl.: <TD CLASS="színes">), akkor arra is érvényes lesz a formázás. Hogy ezt elkerüljük lehetőségünk van konkrét HTML elemek osztályaként megjelölni egy-egy blokkot a CSS állományban.

```
H3.szines{color:red;}
```

Ekkor már csak a H3 címsorokra lesz érvényes a formázás, és azok közül is csak arra melyben osztályként a 'szines' osztály van megjelölve.

5.4 Elhelyezkedés alapján történő meghatározás

Ilyenkor az alapján történik a kiválasztás, hogy az adott HTML elem hol helyezkedik el a hierarchiában. Nézzünk két példát:

```
TD P{padding-bottom:3px;}  vagy
DIV TABLE{border:1px solid black;}
```

Az első esetben a formázás csak azokra a P elemekre vonatkozik melyek valamely TD elemben lettek elhelyezve a HTML dokumentumban. A második esetben pedig csak azokra a TABLE elemekre vonatkozik, melyek valamely DIV elembe lettek ágyazva.

Ezek segítségével például szépen elkülöníthetjük azokat az elemeinket, melyeket táblázatban használunk vagy azon kívül.

5.5 Vegyes meghatározás

Lehetőségünk van a már megismert kijelölési módok kombinálására. Ekkor az egyes technikákat felhasználva tudunk bonyolultabb kiválasztást eszközölni.

1. példa:

```
.szines TD{color:green;}
```

Azokra a TD elemekre vonatkozik, melyek olyan elemen belül helyezkednek el a hierarchiában, amelyek a 'szines' osztályba tartoznak.

Pl.: <TABLE CLASS="szines"><TR><TD>

2. példa:

```
#fontos .szines{color:green;}
```

Azokra az elemekre vonatkozik melyek a 'szines' osztályhoz tartoznak és olyan elemen belül helyezkednek el a hierarchiában melynek az azonosítója 'fontos'.

Pl.: <TABLE ID="fontos"><TR><TD CLASS="színes">

És így tovább. A lehetőségek szinte határtalanok.

5.6 Pszeudo osztályok, és elemek szerinti meghatározás

Léteznek beépített osztályok is a CSS-ben, és ezeket is választhatjuk kijelölőnek, kettősponttal a megadott elem után írva azt. Ilyenek például a hivatkozásra vonatkozóak:

```
A:hover{color:green;}      A hivatkozás felett áll az egér
A:active{color:red;}       Aktivált hivatkozás
A:link{color:black;}       Még nem látogatott hivatkozás
A:visited{color:yellow;}   Látogatott hivatkozás
```

De ilyenek például a szövegtípusú elemek pszeudo elemeinek formázása is:

```
P:first-letter{color:red;} A bekezdés első betűjének formázása
P:first-line{color:red;}   A bekezdés első sorának formázása
```

5.7 Attribútum szerinti meghatározás

Ha nem akarjuk osztályhoz kötni a kiválasztást, és mégis külön akarjuk választani ugyanolyan típusú elemek formázását, akkor jó megoldás lehet az attribútum szerinti formázás. Sokszor használjuk például ezt az INPUT elem formázásánál, melynek több különféle típusa van, függetlenül attól, hogy mind INPUT HTML tag.

```
input[type="text"]{font-size:14px;}
```

Ez a formázás csak azokra az INPUT elemekre vonatkozik, melyeknek létezik type=”text” attribútuma.

6. Háttér, szöveg, betűtípus formázás CSS-ben

6.1 A HTML elemek háttérének beállításához szükséges tulajdonságok

6.1.1 A háttér színe: `background-color`

A tulajdonság beépített színazonosítót, RGB-vel meghatározott értéket vagy színkódot kaphat értékül. Lássunk erre három példát:

```
TD{background-color:yellow;}
TD{background-color:RGB(10,20,30);}
TD{background-color:#FF54A6;}
```

6.1.2 A háttérben elhelyezett kép (háttérkép): `background-image`

Értéke az URL metódus segítségével adható meg, mely metódusnak szöveges formában adjuk meg a kép helyét:

```
TD{background-image:URL('kiskutya.jpg');}
```

6.1.3 A háttérkép ismétlődése, és pozíciója

A háttérkép ismőtlédését a következő tulajdonság határozza meg:

```
background-repeat
```

A tulajdonság értékei a következők lehetnek:

`TABLE{background-repeat:repeat;}`	(ismétlődik mindkét irányban)
`TABLE{background-repeat:no-repeat;}`	(nem ismétlődik a kép)
`TABLE{background-repeat:repeat-x;}`	(csak vízszintesen ismétlődik)
`TABLE{background-repeat:repeat-y;}`	(csak függőlegesen ismétlődik)

A háttérkép pozícióját pedig a:

```
background-position
```

tulajdonság állítja. Értéke két egymástól szóközzel elválasztott konstans lehet, melyek közül az első a függőleges pozíciót `(top, center, bottom)`, a második pedig a vízszintes pozíciót `(left, center, right)` hivatott állítani. Így például egy háttérképet az elem középére így helyezhetünk:

```
TABLE{background-position:center center;}
```

a bal alsó sarokba pedig így:

```
TABLE{background-position:bottom left;}
```

6.1.4 Háttérkép csatolása az oldalhoz: `background-attachment`

A tulajdonság megadja, hogy a csatolt háttérkép az oldal gördítésekor kigördülhet-e az oldalon, vagy fix maradjon, függetlenül az oldal terjedelmétől és pozíciójától.

`BODY{background-attachment:fixed;}`	(A háttérkép mozdulatlan marad)
`BODY{background-attachment:scroll;}`	(A háttérkép kigördülhet)

6.2 A szövegek és betűtípus formázása

A szövegek formázására rengeteg lehetőségünk van. A következőkben a teljesség igénye nélkül sorolunk fel pár példát a fontosabbak és hasznosabbak közül, mely példákban a P bekezdés elemre állítjuk a tulajdonságokat.

- Szöveg színe: `P{color:#45466A;}`
  értéke a már megismert színértékek valamelyike lehet
- Igazítás: `P{text-align:right;}`
  értéke lehet: center, left, right, justify (sorkizárt)
- Sortávolság: `P{line-height:4px;}`
  értéke lehet: %-os érték a jelenlegi betűmérethez képest, konkrét mérték, vagy numerikus érték mely az aktuális betűmérettel szorzódik fel (pl.: másfelenél: 1,5)
- Betűköz: `P{letter-spacing:14px;}`
  értéke az eredeti betüköztől való eltérést jelzi
- Behúzás: `P{text-indent:20px;}`
  értéke lehet konkrét mérték, vagy %-os érték a szülőelem méretéhez képest
- Betűmódosítás: `P{text-transform:uppercase;}`
  értéke lehet: `uppercase` *(nagybetűsít),* `lowercase` *(kisbetűsít),* `capitalize` *(szavak első betűje nagy)*
- Kinézet: `P{text-decoration:underline;}`
  értéke lehet: `underline` *(aláhúzott),* `overline` *(feléhúzott vonal),* `line-through` *(áthúzott)*
- Szóköz: `P{word-spacing:20px;}`
  értéke az eredeti szóköztől való eltérést jelzi

- Betűtípus: `P{font-family:Arial,Helvetica,sans-serif;}`

  értéke egy betűtípus pontos neve, vagy egy betűcsalád szabványos elnevezése
- Betűméret: `P{font-size:120px;}`
  értéke konkrét mérték, %-os érték a szülőelem betűméretéhez képest, vagy egy beépített konstans (`x-small`, `small`, `medium`, `large`, `x-large`, `xx-large` *stb.)*
- A betű stílusa: `P{font-style:italic;}`
- Félkövérség: `P{font-weight:100;}`
  értéke lehet a `bold`, `bolder`, `lighter` *konstansok egyike, vagy egy érték 100-900-ig 100-as léptékkel mely a félkövérség mértékére utal*

7. Listaelemek, táblázatok

7.1 Listaelemek formázása:

A listák gyakran használatos elemek a HTML oldal készítése során. Sokszor lehet szükségünk arra, hogy egy listának az elemeit ne a beépített listaelem jelölők vezessék be, hanem saját egyéni jelölőket válasszunk, vagy pedig a beépítettek közül is valamelyik speciálisra van szükségünk. Ezek beállítására a következő tulajdonságok adottak a CSS-ben:

```
list-style-image
```

Segítségével meg tudunk határozni egy képet, melyet a CSS a listaelemek jelölésére fog használni. Használata hasonló a background-image tulajdonsághoz.

Pl.: `UL.erdo{list-style-image:URL('fa.jpg');}`

Ha nincs a tarsolyunkban megfelelő kép a listaelemek jelölésére, akkor választhatunk a beépített elemek közül is:

```
list-style-type
```

Pl.: `UL.fontos{list-style-type:circle;}`

A tulajdonság értékei a következők lehetnek:

`circle`	körök
`disc`	telített körök
`square`	négyzet
`armenian`	örmény számozás
`decimal`	számozott
`decimal-leading-zero`	számozott bevezető nullákkal ellátva
`georgian`	grúz számozás
`lower-alpha`	kisbetűs felsorolás
`lower-greek`	kisbetűs görög ABC szerint
`lower-latin`	kisbetűs latin ABC szerint
`lower-roman`	kisbetűs római számozás szerint
`upper-alpha`	nagybetűs
`upper-latin`	nagybetűs latin ABC szerint
`upper-roman`	nagybetűs római számozás szerint

7.2 Táblázatok formázása:

Lehetőségünk van úgy fejleszteni weboldalakat, hogy a táblázatokat teljes mértékben elhagyjuk. Ezeket TABLELESS technológiáknak hívják, és a táblázatok helyett általában a DIV és SPAN elemek segítségével építik fel a dizájnhoz szükséges szerkezetet, kinézetet.

Viszont, ha mi mégis táblázatokkal szeretnénk fejleszteni, akkor azok formázására is időt kell fordítanunk, mivel a táblázatok bonyolultsága elég nagy. Egy egyszerű táblázat is legalább három típusú elemből áll: TABLE, TR, TD. Egy komolyabb táblázat ennek sokszorosát is tartalmazhatja, és ahhoz, hogy ezeket ne kelljen egyesével formáznunk, jól elkészített szelektorokra lesz szükségünk a stíluslapon.

Mivel a táblázatban lévő elemek hierarchiája állandó, hiszen a `TABLE` külső elemben a sorokra vonatkozó `TR` elem található mindig és azon belül találhatóak a `TH` vagy `TD` cellákat meghatározó elemek, ráadásul mindegyikből annyi ahány cellát jelenítünk meg, ezért a helymeghatározásos szelektorok nagyon jól használhatóak a táblázatok formázásánál.

```
TABLE.fontos TD
{
     background-color:red;
     color:white;
}
```

Példánk esetében a 'fontos' osztályba tartozó táblák cellái lesznek formázva.

De ugyanilyen hasznos lehet, ha a táblázat minden elemére állítunk be értékeket egyetlen szelektor felsorolással:

```
TABLE, TH, TD
{
     border:2px dashed yellow;
}
```

Ha olyan táblázaton próbáljuk ki a fenti beállítást, amelyben valamely cella teljesen üres akkor látni fogjuk, hogy ezekhez a cellákhoz is megjelenik a szegély. Ennek beállítására az `empty-cells` tulajdonság szolgál, melynek értéke vagy `show (megjelenít)` vagy `hide (elrejt)`.

Például így kell elrejteni az üres cellák szegélyét:

```
TABLE
{
     empty-cells:hide;
}
```

A nem üres cellák szegélyei viszont mindenféleképpen megjelennek, ráadásul mindegyiknek saját szegélye van. Ha szeretnénk, hogy minden egyes cella csak egy közös szegéllyel rendelkezzen, akkor használjuk a következő tulajdonság-érték párost:

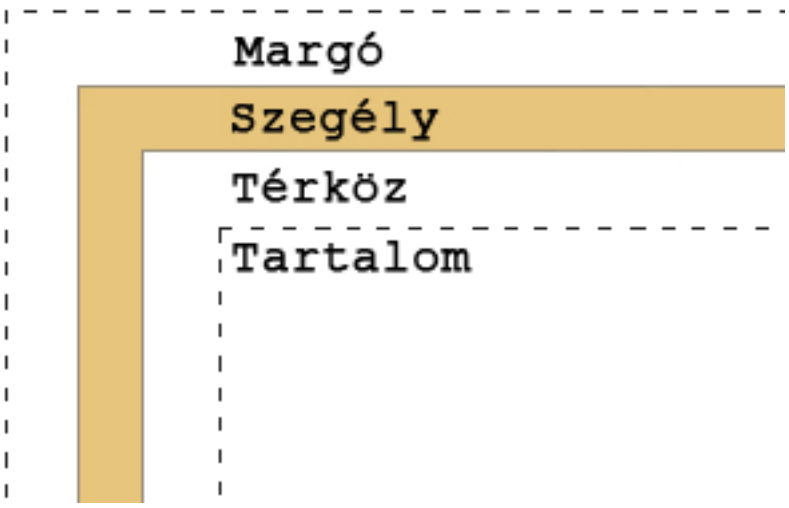
```
TABLE
{
     border-collapse:collapse;
}
```

A táblázatban lévő értékek igazítása vízszintes és függőleges irányban is lehetséges. Ezekre a következő tulajdonságok állnak rendelkezésünkre:

`text-align`	*vízszintes igazítás, értéke lehet* `left`, `right`, `center`
`vertical-align`	*függőleges igazítás, értéke lehet* `top`, `bottom`, `center`

8. Szegély, körvonal, margó, térköz

Ahhoz, hogy megértsük a HTML oldal elemeire beállítható szegélyeket, térközöket, és margókat, először meg kell értenünk, hogyan épül fel CSS-ben az egyes elemek szerkezete. A tényleges tartalom köré milyen sorrendben és módon épülnek rá az egyes tulajdonságok. A lenti ábra ezt hivatott szemléltetni:



A HTML elemek felépítése a CSS számára

Vegyük sorba ezeket a tulajdonságokat.

8.1 Szegély

Egy elem szegélyének háromfajta tulajdonságát állíthatjuk be. A színét, a stílusát, és a vastagságát. Ezeket a tulajdonságokat be tudjuk egyszerre is állítani, de akár egyesével is külön-külön a négy oldalra. Például a szín beállításához használhatjuk a `border-color` tulajdonságot de ha egyesével szeretnénk akkor a

```
border-left-color
border-right-color
border-top-color
border-bottom-color
```

kulcsszavakat válasszuk, melyek sorrendben a bal, jobb, felső, és alsó szegély színét állítják be. Például az alsó keret vörös színre állítva:

```
TD{border-bottom-color:red;}
```

A szegély vastagságánál és stílusánál ugyanez a helyzet. A `border-width` és `border-style` tulajdonságokba a jól ismert szavak közbeékelésével az egyes oldalakat állíthatjuk be. Például `border-top-width` (felső szegély vastagsága), vagy `border-left-style` (bal szegély stílusa).

A szegély vastagságát `thin`, `medium`, és `thick` konstansokkal beállíthatjuk előre definiált méretekre, de konkrét értéket írva pontosan is meghatározhatjuk.

```
DIV{border-left-width:5px;}
```

A szegély stílusánál nem csak ennyi lehetőség közül választhatunk. Kilenc fajta beépített stílus van. Ezek sorban a következők:

`none:`	nincs szegély
`dotted:`	pontozott
`dashed:`	szaggatott
`solid:`	egybefüggő
`double:`	duplavonallal ellátott

És vannak még az úgynevezett 3D-s azaz háromdimenziós hatást keltő stílusok melyek a `groove`, `ridge`, `inset`, és `outset`. Ezek kinézetére hasonló szegélyszín mellett érdemes mind a négy beállítást kipróbálni. Lássunk egy példát is, mely a cella mind a négy szegélyét szaggatottra állítja.

```
TD{border-style:dashed;}
```

Ha tudjuk, hogy be szeretnénk állítani a szegély mindhárom tulajdonságát egyszerre, akkor használjuk az összevont beállítást, ekkor csak a `border`, vagy `border-left`, `border-right`, `border-top`, `border-bottom` valamelyikét válasszuk és kötött sorrendben értékül adjuk neki szóközzel elválasztva a vastagságát, a stílusát, és a színét. Például a bal oldali szegély `3px` vastagra, pontozottra és sárgára állítva:

```
TD{border-left:3px dotted yellow;}
```

Ha már megszoktuk ezt a sorrendet, akkor sajnos el is felejthetjük, mert a következő tulajdonságnál, név szerint a körvonalnál hasonló összevonás esetén a sorrend sajnos a szín, stílus, méret lesz. Bár fejlesztéseink során sokkal kevesebbet fogunk találkozni a körvonal használatával, mint a szegélyekével. A körvonal mindazonáltal hasznos lehet egy-egy elem kiemelésére az oldalon. A körvonal a szegély köré vont vonal melynek stílusa pont ugyanazon értékeket veheti fel mint a szegély. Például egy kék színű egybefüggő `4px` vastag körvonal a bekezdésen így állítható be:

```
P{outline:blue solid 4px;}
```

8.2 Margók és térközök

A margók és a térközök beállítása bár más-mást jelent, használatuk pontosan ugyanúgy működik. Hasonlóan a szegélyekhez, itt is állíthatjuk az összes margó és térköz méretét, de állíthatjuk egyesével is a négy oldalt. Margók esetében ezt a `margin` és a `margin-left`, `margin-top`, `margin-right`, `margin-bottom`, térközök esetében pedig a `padding`, valamint a `padding-left`, `padding-top`, `padding-right`, `padding-bottom` tulajdonságokkal állíthatjuk be.
Például egy `DIV` elem bal oldali margóját `5px` jobb oldali térközét pedig `10px` méretűre állítani így tudjuk:

```
DIV
{
     margin-left:5px;
     padding-right:10px;
}
```

Ha egyszerre akarjuk beállítani a négy oldalt, akkor ügyeljünk a sorrendre. Az egyes értékeket szóközzel elválasztva a következő sorrendben kell megadni:

felső – jobb oldali – alsó – bal oldali

Például:

```
TD
{
     margin:10px 20px 30px 40px;
     padding:35px 15px 25px 30px;
}
```

felső margó 10px, jobb margó: 20px, alsó margó:30 px, bal margó: 40px és hasonlóan a térköz esetében.

9. Méretezés, kijelzés vezérlés, elhelyezés

Az elemek méretezésénél vegyük figyelembe a már megtanult struktúrát, ahogy egy elem felépül. Nézzük meg újra az ábrát:

Egy elem szélessége és magassága beállításakor a CSS csak az elem tartalmának szánt helyet méretezi. Egy elem végleges valós méretét a hozzá tartozó térköz, szegély és margó mérete is meghatározza.
Nézzük meg hogyan kell állítani egy elem méretét.

Szélesség: `width`
Magasság: `height`

Például egy `DIV` elem `200px` szélesre és `300px` magasra állítva:

```
DIV{width:200px;height:300px;}
```

Azonban hogy ennek a tényleges mérete mekkora a megjelenéskor a fent felsorolt tényezőktől függ. Például ha egy `DIV` szélessége `200px`, a hozzá tartozó margó `10px`, a térköz `50px` és a szegély `2px`, akkor a tényleges szélessége nem `200px` lesz hanem 200+20+100+4 azaz `324px`. Vegyük észre, hogy a margó, térköz és szegély értékeket kétszer számoltuk, hiszen a bal és jobb oldalon is hatást fejtenek ki. Ugyanígy működik a magasság esetében is. A margók, térközök, és szegélyek felső és alsó értéke is beleszámolódik a végleges méretbe.

Már tudjuk mekkora 'helyet' foglal el az oldalunkon egy-egy elem, most nézzük meg hogyan tudjuk ezeket az elemeket elrejteni és megjeleníteni, valamint teljesen eltüntetni az oldalról.

A megjelenésért két tulajdonság a felelős. A `visibility` és a `display`. Az első inkább a láthatóságot a második pedig a megjelenést kezeli. A láthatóság két értéket képviselhet, vagy látható vagy nem. Ezek a `visible` és `hidden` konstansokkal érhetők el. Például egy fent említett `DIV` elemet el tudunk rejteni a következő módon:

```
DIV{visibility:hidden;}
```

Viszont, ez a módszer bár elrejti az oldalról az elemet, nem szabadítja fel annak helyét. A fent kiszámolt helyigénye megmarad, az oldal szerkezetét továbbra is befolyásolja. Ahhoz, hogy az oldalról teljesen eltűnjön, ki kell kapcsolni a megjelenítését. Erre való a már említett `display` tulajdonság. Például a `DIV` elem megjelenítésének kikapcsolása:

```
DIV{display:none;}
```

Ha mégis meg akarjuk jeleníteni az elemet, akkor két típus közül választhatunk.

```
DIV{display:inline;}          DIV{display:block;}
```

A kettő között az a különbség, hogy még az `inline` úgy helyezi el az elemet, hogy a megjelenítés után nem tör sort, a `block` beállítás megteszi azt. Az `inline` hasznos lehet olyan HTML elemeknél, melyek automatikusan sort törnek, de mi nem szeretnénk ezt. Például a `FORM` tag vagy az `LI` tag. A `block` pedig pont az ellenkező esetben, azoknál a `HTML` elemeknél, amelyek alapból nem törnek sort. Ilyen például a `SPAN` vagy az `A` tag.

Ha úgy döntünk, hogy megjelenítjük az elemeinket, fontos hova helyezzük el őket. Az elemek normál esetben statikus pozícióval rendelkeznek melyek automatikusan számolódnak ki. Az elemek pozíciójának megváltoztatásához a `position` tulajdonságot kell új értékkel felruháznunk, majd utána a `left`, `right`, `top`, `bottom` tulajdonságokkal állíthatjuk be az elem pontos helyét. Ezen tulajdonságok értékei az elem megadott szélének távolságát jelentik a viszonyított elemhez képest. Például a `left:50px`; azt jelenti, hogy az elem bal széle `50px-re` helyezkedjen el a viszonyított elem bal szélétől. Vagy pedig lehet `auto` érték melynél a böngésző dönti el az egyes szélek helyzetét, ha tudja. Az, hogy éppen mihez viszonyítunk a `position` tulajdonság értéke dönti el. Lássuk ezen lehetőségeket:

```
DIV{position:fixed;bottom:20px;}
```

A fixed érték esetén az elem helyzete a böngészőablak szélétől lesz számítva, függetlenül attól, hogy az oldalt elgördítették-e vagy nem. Jól használható pl. egy logo állandó megjelenítéséhez az oldal egy bizonyos pontján.

```
DIV{position:relative;left:-20px;}
```

A `relative` érték esetén az elem helyzete relatívan az eredeti pozíciójától lesz számítva. Mint láthatjuk a példából a pozíciók negatív értékkel is rendelkezhetnek.

```
DIV{position:absolute;top:30px;}
```

Az `absolute` érték esetén az elem helyzete relatívan az első olyan szülőobjektum szélétől lesz számítva amelynek átállítottuk position tulajdonságát másra mint statikus (`static`). Ha nincs ilyen, akkor maga a dokumentum lesz ez a szülő objektum.

Néha elkerülhetetlen hogy az egyes elemek fedésbe kerüljenek. Addig-addig állítgatjuk azok helyzetét, hogy az egyik eltakarja a másikat, vagy annak egy részét. Hogy ilyenkor melyik elem kerül mégis „feljebb” a `z-index` tulajdonság határozza meg. Úgy kell elképzelni az elemeket, mintha külön rétegekre helyeznénk el őket. Minél nagyobb a `z-index` értéke annál magasabb rétegbe helyezkedik el. Tehát egy kisebb `z-index` értékkel rendelkező elemet elfed.

```
DIV{position:absolute; left:40px; z-index:10;}
```

10. Képek, szövegek igazítása

Az elemek igazítására már tanultunk módszereket, most foglaljuk ezeket össze, valamint nézzünk meg egy új lehetőséget.

Egy elemen belül, a képek szövegek igazítását a már megismert text-align és vertical-align tulajdonságokkal tudjuk jobbra, balra, és középre helyezne, vízszintesen valamint függőlegesen.

Második lehetőségünk hogy a már megtanult margó tulajdonságot kihasználva helyezünk el egy elemet a szülő objektum valamely részére. A margónak értékül adhatunk egy úgynevezett auto értéket, mely a böngészőre bízza az adott margó értékének kiszámítását. Ezt kihasználva pl. helyezzünk minden irányból középre egy elemet:

```
DIV{margin:auto;}
```

csak vízszintesen középre:

```
DIV{margin-left:auto;margin-right:auto;}
```

vagy jobbra úgy, hogy a bal oldali margót állítjuk automatikusra:

```
DIV{margin-left:auto;}
```

Harmadik lehetőségünk az előzőekben tanult pozíció (`position`) meghatározása. Ha ügyesen használjuk a `left`, `right` stb. tulajdonságokat, az elem a kívánt helyen jelenik majd meg.

Negyedik lehetőségünk egy új tulajdonság megismerését igényli. Ez pedig a `float`. A `float` tulajdonság lényegében az elemek igazodását jelenti. A megjelenített elemek a böngészőablak méretétől függően megpróbálnak a megadott irányba csoportosulni. Az irány kétfajta lehet,

vagy bal: `IMG{float:left;}`
vagy jobb: `IMG{float:right;}`

Ha több elemre is beállítjuk egymás után ezeket a tulajdonságokat, akkor sorban egymás utána próbálnak az elemek igazodni a megadott irányba. Néha viszont szükség van arra, hogy a soron következő elemhez már ne igazodjanak a float tulajdonsággal ellátott elemek. Ekkor használjuk a clear „utasítást” mely „megtisztítja” az elem adott oldalát.

Az elem jobb oldalához nem engedélyezett az igazodás:
```
P{clear:right;}
```
Az elem bal oldalához nem engedélyezett az igazodás:
```
P{clear:left;}
```
Az elem egyik oldalához sem engedélyezett az igazodás:
```
P{clear:both;}
```

11. Átlátszóság

Bár az átlátszóság nagyon hasznos, és sok esetben jól mutat a weboldalunk bizonyos elemeire aktiválva, sajnos még nem CSS szabvány. Jelenleg két tulajdonság is van a CSS-ben amely ezt hivatott állítani, ráadásul ezeknek értéktartománya is eltér.

Az első az `opacity:érték` melynek értéke 0.0 és 1.0 közé eshet, úgy, hogy 0.0 a teljesen átlátszó a 1.0 pedig az egyáltalán nem átlátszó elemet eredményezi.

A másik ilyen tulajdonság a `filter:alpha(opacity=érték)` melynél az érték 0 és 100 közé esik hasonló értelmezésben, miszerint a 0 a teljesen átlátszó a 100 pedig az egyáltalán nem átlátszó elemet eredményezi.

A második megoldás az Internet Explorer böngésző számára értelmezhető üzenet, az első pedig pl. a firefox böngésző számára. A CSS3 szabványban az első változat lesz az elfogadott.

Ha azt akarjuk, hogy vélhetően mindegyik böngésző értelmezni tudja majd a beállításainkat, átlátszóság esetén mindkét paramétert egyszerre használjuk, és egyenértékű értékekkel ruházzuk fel.
Pl.:

```
IMG
{
opacity:0.3;
filter:alpha(opacity=30);
}
```

Mint példánkban is látjuk a tulajdonságokat egy IMG azaz egy kép elemre állítottuk be. Ez nem véletlen. Ezeket az átlátszósággal kapcsolatos effektusokat főleg képekre, DIV elemekre állítjuk be. Két okból is. Először is mivel nem biztos, hogy minden böngésző minden esetben képes az átlátszóságot kezelni, nem szerencsés szövegekre beállítani az effektust, mivel rossz esetben olvashatatlanná válik a szöveg. Másrészről pedig a képek, vagy a DIV és egyéb blokk jellegű elemek hátterébe helyezett képek átlátszóvá téve jól mutathatnak. Arra ügyeljünk, hogy pl. egy DIV elemet átlátszóvá téve nem biztos, hogy a benne megtalálható gyermek elemek is átlátszóvá válnak. Ezt a böngészők különbözőképp kezelik!

12. Teszt

Komplex teszt végrehajtása a MOODLE e-learning keretrendszerben automatikus értékeléssel.
Az értékelés típusa: pontszámarányos értékelés ötfokozatú skálán értékelve, érdemjegyekkel.

13. JavaScript története, fejlődése

A JavaScript története egészen 1996-ig nyúlik vissza, szorosan összekapcsolódva az ismert Netscape böngésző második verziójának megjelenéséhez. A programnyelv ebben a böngészőben jelent meg először. A nyelvet az akkor a Netscape-nél dolgozó Brendan Eich nevű fejlesztő készítette. Eredetileg Mocha majd később LiveScript néven futott, és csak ezután kapta meg "végleges" nevét, a JavaScriptet. Bár elméletileg volt köze a névadásnál a Java-hoz, főleg marketing okokból, nagyon fontos hogy a JavaScript-nek semmi köze a Java-hoz. Vagy ahogy sok-sok JavaScript oktató könyv kezdi: **JavaScript != Java.**
A nyelv létrejöttének oka, hogy szükség volt egy olyan eszközre, mellyel a megjelenített HTML, XML stb. oldalak programozhatóvá válnak. Az egyes elemek kliens, avagy szerver oldalon módosíthatóak lesznek az oldal betöltése után is.
A nyelv alapja az úgynevezett ECMA Script szabvány, amely sok szkriptnyelvnek az alapja. Annak változataira épülve és kiegészítve a böngészőben hasznos objektumok kezelésével alakultak ki az egyes változatok. Ezeknek az objektumoknak a leírására való a DOM (Document Object Model). Nézzük, hogy épülnek fel az egyes verziók:

- JavaScript 1.0

  1996-ban jelent meg a Netscape 2.0 böngészőben, valamint később az Internet Explorer 3.0-s verziójában. A korai ECMA Script szabványra és a DOM Level 0-ra épült a képelemek kivételével.

- JavaScript 1.1

  A Netscape 3.0 használta. Alapja a korai ECMA Script és a már teljes DOM Level 0.

- JavaScript 1.2

  1997 júniusában jelent meg a Netscape 4.0-ás verziójától. Az ECMA Script szabványra és a DOM kiterjesztett változatára épült (DOM Level 0 + Proprietary DOM vagy Layers DOM), mely leírja hogy az egyes pozícionált elemek és gyermek elemek stílusát hogyan érhetjük el és változtathatjuk meg.

- JavaScript 1.3

  1998 október. A Netscape 4.06-tól valamint az Internet Explorer 4.0-tól. A továbbfejlesztett ECMA Script szabványra és persze a DOM kiterjesztett verziójára épült.

- JavaScript 1.4

Csakis a Netscape Server változathoz készült. (Server side JavaScript)

- JavaScript 1.5

2000 november. Az ECMA Script harmadik verziójára épül valamint a DOM W3C-re mely már teljes egészében leírja az elemek kezelését, elérését, valamint az egyes elemek, objektumok létrehozását, törlését, módosítását az oldal betöltődése után. Ezt a verziót lényegében az összes ma is használt elterjedt böngésző támogatja. A Netscape 6.0-ás verziója, az Internet Explorer 5.5-től napjainkig (ezekben a böngésző a Microsoft már JScript-nek hívja), az ekkor megjelent Mozilla Firefox 1.0, az Opera 6.0-tól napjainkig, a Safari 3.0-tól napjainkig, és a Google Chrome.

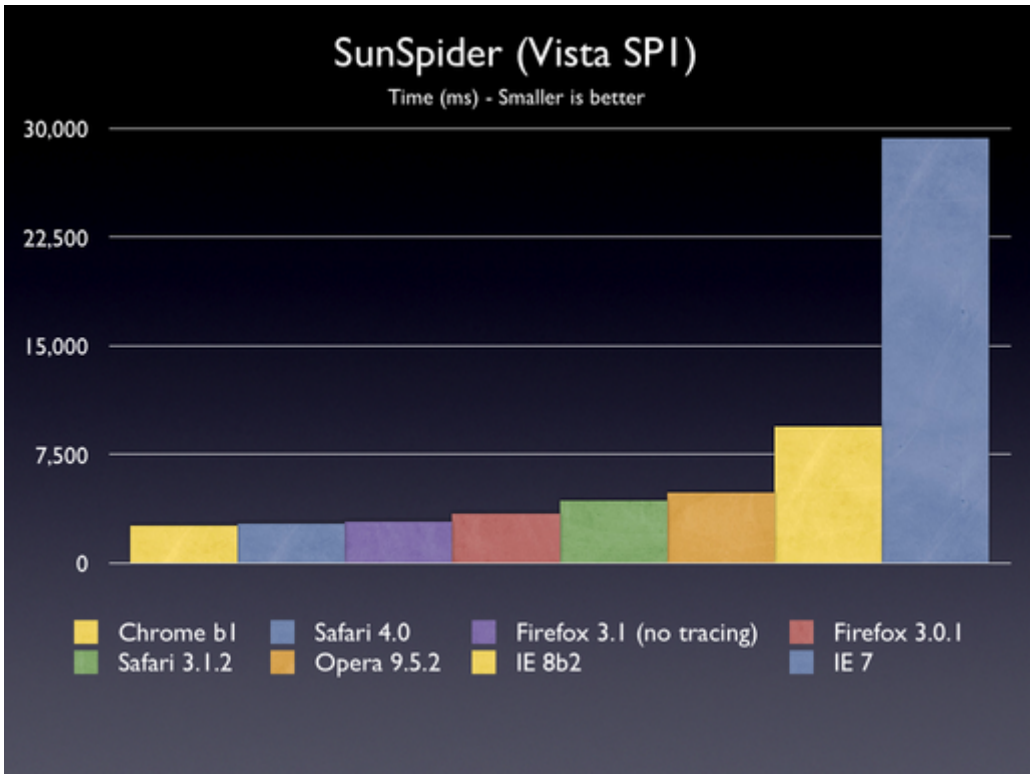
- JavaScript 1.6, 1.7, 1.8, 1.8.1, 1.8.2, 1.9

Ezek a verziók mind csak az 1.5-ös valamilyen fejlesztései melyek csak a Mozilla Firefox új verzióban jelentek meg vagy fognak megjelenni. Sorban az 1.5, 2.0, 3.0, 3.5, 3.6, 4.0. Ennek oka egyszerű. A nyelv fejlesztője Brendan Eich ahogy a fejlődésen is végigkövethető először a Netscape-nél dolgozott, majd részt vett a Mozilla Foundation megalapításában, és a Mozilla fejlesztője lett, ahol napjainkban is tevékenykedik.

- JavaScript 2.0

A Mozilla által tervezett jövőbeli kiadás.

A JavaScript programnyelv elhanyagolhatatlanná vált. Aki szeretne weboldalakat fejleszteni, és azokat dinamikus oldalakká varázsolni, elkerülhetetlen, hogy megismerje és jól használja a nyelvet. Ezt még jobban megerősíti az a tény, hogy az AJAX (asynchronous JavaScript and XML) kifejlesztésével újra központba került a JavaScript nyelv.
Nem véletlen, hogy sok böngésző lassúságát, vagy éppen gyors működését a JavaScript feldolgozó motor okozza. Majdnem mindegyik weboldalon található valamilyen kisebb, nagyobb szkript. A böngészőnek az oldal megjelenítésekor ezeket a programokat fel kell dolgoznia. Ennek sebessége kihat az oldal betöltődési idejére. A böngészők versenyében egyértelműen azok az alkalmazások látszanak kiugróan magas eredményeket elérni, akik újabb és újabb JavaScript motort használva ezek feldolgozási idejét csökkentik. Itt egy régebbi diagram ennek bemutatására:



forrás: ejohn.org

14. Objektumorientáltság, objektumok

A JavaScript a programozónak jó néhány beépített objektumot biztosít, amelyek mindegyikének számos *elemváltozója* és *elemfüggvénye* van. Az objektumok elérést biztosítanak a HTML dokumentumunk szinte minden fontosabb entitásához, ugyanis a fontosabb, pl. a felhasználói bevitelt támogató HTML tag-ekre most már programból is hivatkozhatunk, azaz a tag-ekkel definiált gombokat, szövegbeviteli sorokat vagy ablakokat immár JavaScript objektumokként érhetjük el.

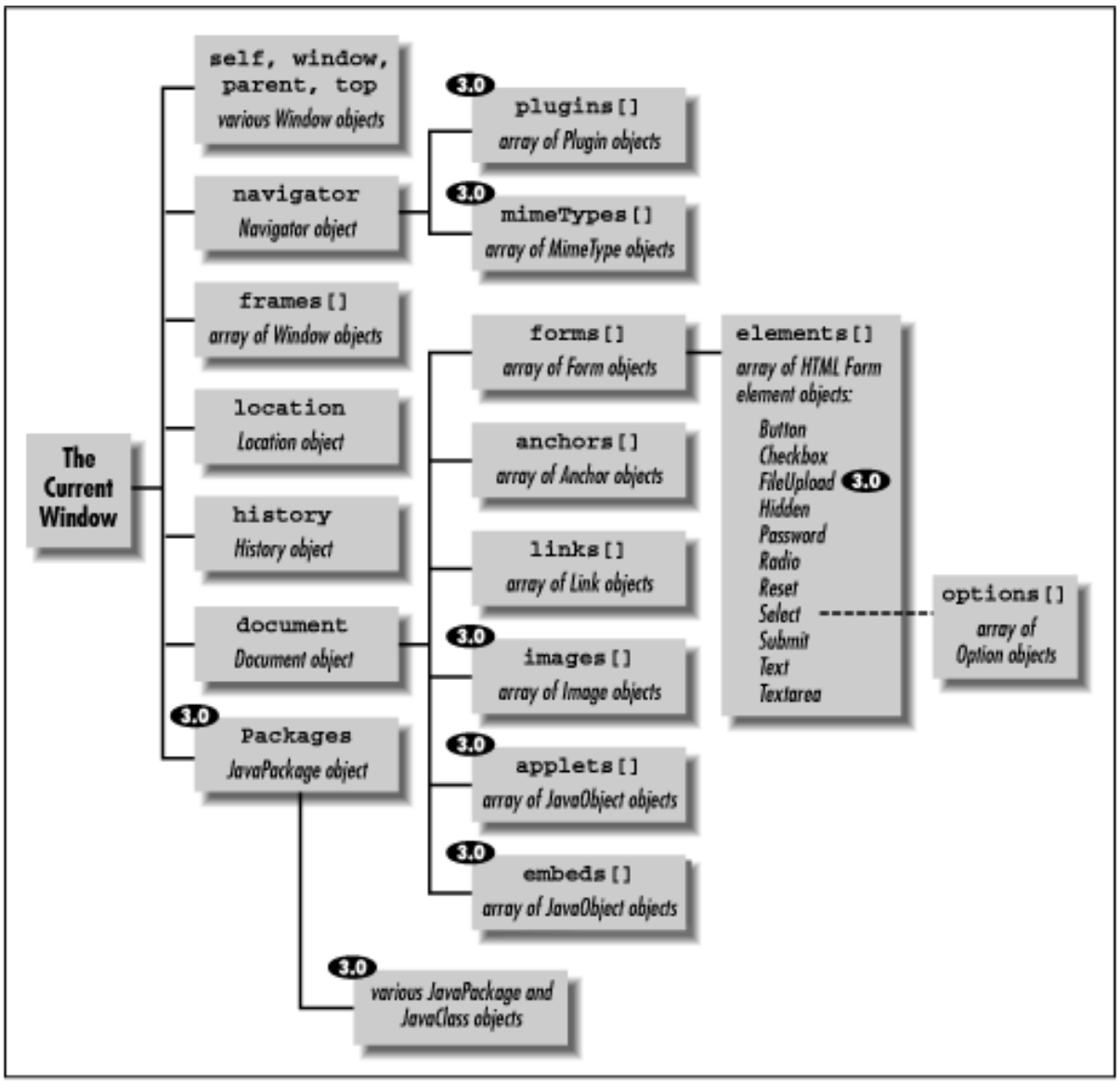
Ebben a tananyagrészben a következőkről lesz szó:

- Melyek a JavaScript beépített objektumai, azok elemváltozói és elemfüggvényei?
- Hogyan tegyük alkalmassá a jól ismert HTML tag-eket arra, hogy JavaScript utasításokat hajtsanak végre bizonyos események bekövetkezése esetén?
- Hogyan nyerjünk ki értékeket az űrlapok elemeiből, például szövegbeviteli ablakokból, egymást kiváltó ún. rádiógomb-menükből és igen/nem válaszadásra („kipipálásra”) alkalmas kapcsolókból (checkbox)?
- Hogyan hozzunk létre és hogyan kezeljünk stringeket (string-objektumokat)?

- Melyek azok a programozó által definiált gombok, amelyek JavaScript függvények hívására alkalmasak, azaz „függvényélesztő” eseményeket tudnak előidézni?
- Hogyan tudjuk a matematikai függvényeket és konstansokat elérni?

14.1 A JavaScript objektumainak típusai

Az 1. ábra tünteti fel a JavaScript által támogatott objektumokat.



1. ábra: A JavaScript objektumai

14.2 A JavaScript előre definiált objektumainak használata

Néhány objektumot, mint például a Date és a Math , a JavaScript fordító eleve tartalmaz, így ezeknek semmi közük a HTML tag új paramétereihez, míg mások kapcsolatban vannak bizonyos HTML tag-ek paramétereivel.

Például a szövegbeviteli sor objektum (<input type=”text” size=20>) lehetővé teszi, hogy egy szövegbeviteli (input) sor tartalmára hivatkozhassunk.

Az objektumok azonosításának érdekében két dolgot tettek:

Először is egy új name paraméter került számos HTML tag-be azért, hogy ezekre az objektumokra egyértelműen hivatkozhassunk.

Például a következő HTML tag:

```
<input name=”mezoNev” type=”text” size=20>
```

Nem csupán egy *szövegbeviteli* sor, hanem egy szövegbeviteli *objektum* is.

Másodszor, a JavaScript a HTML írójának megengedi, hogy az objektumok elemváltozóit is elérje. Ezek az elemváltozók az objektumok belső adatstruktúráját valósítják meg.

Például a document nevű objektumnak van egy bgColor nevű elemváltozója, ami az aktuális dokumentum háttérszínét adja meg.

14.2.1 Ízelítőül (és kissé előre szaladva) két érdekes objektum

Checkbox (kapcsoló) objektum

Elemváltozók

`checked`	Egy igaz vagy hamis értékkel tükrözi ez a tulajdonság, hogy a `checkbox` be van-e kapcsolva.
`defaultChecked`	A `checkbox` tulajdonsága, amely igaz vagy hamis értékével azt adja meg, hogy volt-e a kapcsolónak `CHECKED` attribútuma.
`form`	Az űrlap neve, amelyben a `checkbox` található.
`name`	A `NAME` attribútum értéke.
`type`	A `TYPE` attribútum értéke, ez itt „`checkbox`”.
`value`	A `VALUE` attribútum értéke.

Elemfüggvények

`blur()`	Elveszi a beviteli fókuszt a `checkboxról`.
`click()`	A `checkboxra` kattintást szimulál, de nem hívja meg az `onClick` eseménykezelőt.
`focus()`	A `checkbox` megkapja a beviteli fókuszt.
`handleEvent()`	Meghívja a megadott típusú esemény kezelőjét.

Eseménykezelők

`onBlur`	A megadott eseménykezelő akkor fut le, ha `Blur` esemény következik be, azaz a `checkbox` elveszti a beviteli fókuszt.
`onClick`	A megadott eseménykezelő akkor fut le, ha `Click` esemény következik be, azaz a felhasználó rákattintott a `checkboxra`.
`onFocus`	A megadott eseménykezelő akkor fut le, ha `Focus` esemény következik be, azaz a `checkbox` megkapja a beviteli fókuszt.

Button (gomb) objektum

Elemváltozók

`form`	Az űrlap neve, amelyben a `button` található.
`name`	A `NAME` attribútum értéke.
`type`	A `TYPE` attribútum értéke.
`value`	A `VALUE` attribútum értéke.

Elemfüggvények

`blur()`	Elveszi a beviteli fókuszt a `buttonról`.
`click()`	A `buttonra` kattintást szimulál, de nem hívja meg az `onClick` eseménykezelőt.
`focus()`	A `button` megkapja a beviteli fókuszt.
`handleEvent()`	Meghívja a megadott típusú esemény kezelőjét.

Eseménykezelők

`onBlur`	A megadott eseménykezelő akkor fut le, ha `Blur` esemény következik be, azaz a `button` elveszti a beviteli fókuszt.
`onClick`	A megadott eseménykezelő akkor fut le, ha `Click` esemény következik be, azaz a felhasználó rákattintott a `buttonra`. Ha az eseménykezelő `false` értékkel tér vissza, a böngésző nem küldi el a `formot`.
`onFocus`	A megadott eseménykezelő akkor fut le, ha `Focus` esemény következik be, azaz a `button` megkapja a beviteli fókuszt.
`onMouseDown`	A megadott eseménykezelő akkor fut le, ha `MouseDown` esemény következik be, azaz a felhasználó rákattint a

	`button`ra. Ha az eseménykezelő hamisat ad vissza, a gomb nem nyomódik le.
`onMouseUp`	A megadott eseménykezelő akkor fut le, ha `MouseUp` esemény következik be, azaz a felhasználó felengedi az egér gombját a `button` felett. Ha az eseménykezelő hamisat ad vissza, az `onClick` eseménykezelő nem hívódik meg.

15. Adattípusok

A JavaScript változói és konstansai alapvetően három adattípust használhatnak:

- stringek: karakterláncok
- numerikus értékek: egész és valós számok
- boolean: igaz vagy hamis logikai értékek

A **stringek** karakterláncok, amelyeket egyszeres (’) vagy kétszeres (”) idézőjelek között kell megadnunk. Stringek pl. ”Jancsika fut”, ”115,234” vagy a ”4+2=9” is. Számokat stringként is tárolhatunk, és ennek ellenére végezhetünk velük matematikai műveleteket.

Ha a böngésző egy ”Gyerünk\nÚszni!” karakterlánccal találkozik, azt két sorban jeleníti meg, mivel a stringben \n újsor karakter található.

1. táblázat. **Szövegformátum vezérlő kódok**

Kód	Jelentése
\n	Új sor
\t	Tabulátor karakter
\r	Kocsi vissza
\f	Lapdobás
\a	Hangjelzés
\b	Backspace

Nyilván hibásnak kell tekintenünk a következő kódot, hiszen az idézőjelek nincsenek párban:

```
AStringem = ”Hibás’
```

A **numerikus** értékek (számok) mind egész (*integer*), mind pedig valós (*floating point* vagy *real*) számok lehetnek.

Például 0, -55 és 59 egész típusú, míg 25.43232, -0.3456 és 3.8E-6 valós számok.

A **boolean** típusú (logikai) adatok csak *true* (igaz) illetve *false* (hamis) értéket vehetnek fel.

15.1 Saját változó definiálása

A JavaScriptben nem szükséges a változók típusát megadnunk. A változónevek az angol ABC betűjével (a-z vagy A-Z) illetve aláhúzás karakterrel '_' kell, hogy kezdődjenek, majd betűk, számjegyek és újabb aláhúzás karakterek bármilyen kombinációja szerepelhet a névben.

- `atmenetiValtozo`
- `TalalatokSzama`
- `Ennyi_nap_van_a_honapban`

A fentiek helyes változónevek. Hogyan lesznek ezek valóban változók?

```
var atmenetiValtozo = -34.56;
var TalalatokSzama = 45;
Ennyi_nap_van_a_honapban = 30;
```

Ezek szerint a var kulcsszó használható a változók definiálásakor, de nem kötelező. Nem lehetünk azonban biztosak ekkor abban, hogy az Ennyi_nap_van_a_honapban változót korábban definiáltuk-e, vagy ez csupán egy var nélküli definiálás.

Inkább használjuk a var kulcsszót! Persze értéket nem feltétlenül kell azonnal hozzárendelnünk a változóhoz:

```
var ittCsupanDefinialunkErteketNemAdunk;
```

Mivel adattípust sem adtunk meg, így ez egy homályos változódeklaráció.

A var kulcsszó használata egy függvénytörzsben egyébként is kötelező, ha már van ugyanolyan nevű globális változó, mint amilyet a függvényben használni szeretnénk.

16. Változók

A JavaScript-ben a változókat kétféleképpen hozhatjuk létre. Vagy értékadással hozzuk létre, vagy pedig a var kulcsszó használatával.

Amikor a var kulcsszóval hozzuk létre a változót, nem kötelező neki kezdőértéket adni. Például:

```
var szoveg;
```

Ekkor a változónak, mint bizonyára sejtettük még nem lesz értéke, de típusa lényegében már lesz, méghozzá a definiálatlan típus (undefinied). Ha értékadással hozzuk létre a változót, akkor rendelkezni fog értékkel is és az abból következtetett típussal is. Például:

```
szoveg=”Kis Pista vadászni ment”;
```

A változó neve szoveg, értéke „Kis Pista vadászni ment”, típusa pedig string.

A var kulcsszó használata mellett több változót is felsorolhatunk, valamint ekkor is adhatunk meg kezdőértéket a változónak méghozzá többféle módon.

```
var a=0;
var a,b=0.56;
var a,b=0,c=0;
var a,b=c=0;
var tomb=[];
var tomb=[6,89];
var tomb=new Array();
var barmi=new Object();
```

A JavaScript nem erősen típusos nyelv ezért a változók típusa akár menet közben is változhat a programunk futása során. Nézzük meg a következő példát:

```
var s=”Helló”;
document.write(s);
s=6+4;
document.write(s);
s=s+1;
document.write(s);
```

A végeredmény a képernyőn: Helló1011

Az s változó az első értékadás után string típusú változóként jött létre. Majd amikor az újabb értékadást végrehajtottuk rajta már numerikus értéket képviselt. Ezután már végrehajthattunk rajta aritmetikai műveletet is.

Mi lehet egy változó értékadásnál a jobb oldalon:

Numerikus érték:	`var a=5.4;`
Szöveg:	`var a=”Lajos”;`
Logikai érték:	`var a=true;`
Tömb:	`var a=new Array(5,6,7);`
Objektum:	`var a=document.urlap;`
Függvény értéke:	`var a=osszead(a,b);`

Függvény maga: `var a=function (){ alert('Helló');}`

Az utolsó értékadás érdekes megoldás. Ha kiíratjuk az a változó értékét `document.write(a);` akkor ezt kapjuk: `function(){ alert('Helló');}` Ha viszont hivatkozunk rá a változó nevével és a függvény híváskor megszokott két zárójellel a(); akkor az `alert` metódus jut érvényre és a függvény végrehajtódik

17.Operátorok

Az operátorok olyan programnyelvi elemek melyek elvégzik a műveletek végrehajtását. Több csoportba sorolhatjuk őket aszerint, hogy milyen műveleteket végeznek. Első ilyen csoport az aritmetikai operátorok, azaz a matematikai műveletekhez szükséges operátorok.

Mielőtt felsorolnánk az aritmetikai operátorokat, mindenekelőtt kell megemlíteni a értékadó utasítás operátorát, magát az egyenlőség jelet. Az egyenlőségjel kétoperandusú operátor, mely a bal oldalt álló, értéket tárolni képes operandusba betölti a jobb oldalán álló kifejezés értékét.
Pl.: `x=5;`

17.1 Aritmetikai operátorok

A példák miatt említsük meg hogy az "a" változónk jelenlegi értéke 5.

	operátor	példa	új eredmény
Összeadás:	+	a=a+4	9
Kivonás:	-	a=a-3	2
Szorzás:	*	a=a*4	20
Osztás:	/	a=a/2	2.5
Maradékszámítás:	%	a=a%4	1
Inkrementálás:	++	a++	6
Dekrementálás:	--	a--	4

Bizonyos kifejezésekben lehetőségünk van több operátor összevonására:

```
a=a+4;          a+=4;
a=a-4;          a-=4;
a=a*4;          a*=4;
a=a/4;          a/=4;
a=a%4;          a%=4;
```

A következő csoport az összehasonlító operátorok. Ezeket az operátorokat a logikai kifejezésekben használjuk. Segítségükkel olyan kifejezéseket készíthetünk, melynek eredménye a logikai IGAZ vagy HAMIS lesz.

A példák miatt említsük meg hogy az "a" változónk jelenlegi értéke 10.

	operátor	példa	eredmény
Egyenlő:	==	a==10	igaz
Pontosan egyenlő:	===	a==="10"	hamis
	(értéket és típust is vizsgál)		
Nem egyenlő	!=	a!=10	hamis
Nagyobb mint:	>	a>10	hamis
Kisebb mint:	<	a<10	hamis
Nagyobb vagy egyenlő:	>=	a>=10	igaz
Kisebb vagy egyenlő:	<=	a<=10	igaz

Létezik a JavaScript-ben 3 operandusú operátor is. Ez a feltételes operátor „`? :`” melynek szintaktikája a következő:

```
(feltétel)?érték1:érték2
```

Ha a zárójelben leírt feltétel `IGAZ` akkor a kettőspont előtti érték lesz a kifejezés értéke, ha pedig `HAMIS` akkor a kettőspont utáni érték lesz a kifejezés értéke. Például:

```
var uzenet=(kor<18)?”Kedves diáktársunk”:”Kedves felnőtt”;
```

18. Logikai operátorok

Lássuk ezeket:

A példák miatt említsük meg hogy az "a" változónk jelenlegi értéke 20, „b” változónk jelenlegi értéke pedig 15.

	operátor	példa	eredmény
Logikai ÉS kapcsolat:	&&	(a>=5) && (b<=20)	igaz
Logikai VAGY kapcsolat:	||	(a<5) || (b==20)	hamis
Tagadás (egyoperandusú):	!	!(a>5)	hamis

Két, ÉS kapcsolattal összekötött kifejezés értéke csakis akkor lesz IGAZ, ha mindkét a kapcsolatban résztvevő kifejezés értéke egyenként is IGAZ volt.

Két, VAGY kapcsolattal összekötött kifejezés értéke csakis akkor lesz HAMIS, ha mindkét a kapcsolatban résztvevő kifejezés értéke egyenként is HAMIS volt. Tehát ha bármelyik a kettő közül IGAZ, a végeredmény is IGAZ lesz.

A tagadás mindig ellenkezőjére állítja az eredeti értéket. IGAZ-ból HAMIS lesz és fordítva, HAMIS-ból IGAZ.

Bitszintű műveletekhez szükséges operátorok (az érték kettes számrendszerbeli alakjának bitjein hajtódik végre a művelet):

Bitenkénti tagadás:	~
Bitenkénti VAGY:	|
Bitenkénti ÉS:	&
Bitenkénti kizáró vagy (XOR)	^
Bitenkénti eltolás balra:	<<
Bitenkénti eltolás jobbra:	>>

19. String operátorok

String típusú értékek, azaz szövegek összefűzésére is használunk operátort. Ekkor hasonlóan az aritmetikai összeadáshoz a + jelet használjuk, de abban az esetben, ha a két operandus egyike string, a végeredmény a két érték összefűzéséből álló karaktersorozat lesz.

```
”alma”+”körte” = ”almakörte”
”46” + ”57” = ”4657”
”46” + 57 = ”4657”
46 + 57 = 103
”Helló”+(5>4) = ”Helló true”
```

20. Típuskényszerítés

A JavaScript esetében könnyű az átmenet a változó típusok között. Ha a feldolgozó kiértékel egy kifejezést, akkor az alábbi szabályok szerint jár el:

Tevékenység	**Eredmény**
szám és szöveg összeadása	a szám a szövegbe épül
logikai érték és szöveg összeadása	a logikai érték a szövegbe épül
szám és logikai érték összeadása	a logikai érték a számba épül

21. Elágazások

JavaScript-ben az elágazások hasonlóan sok más programnyelvhez valamilyen feltételtől függő utasítások végrehajtását jelentik. Ehhez a JavaScript-ben, két utasítás áll rendelkezésünkre.

Az első az `IF` utasítás. Segítségével kétirányú elágazást hozhatunk létre a programunkban, vagy egymásba ágyazásuk esetén akár több irányút is. Az utasítás szintaktikája a következőképpen néz ki:

```
if (feltétel) {
     …utasítások…
}
```

Ez a normál eset. Ekkor a feltételben kiértékelt kifejezés IGAZ volta esetén végrehajtódnak a blokkjelek {} közti utasítások, HAMIS esetén pedig azokat átugorja a program, tehát nem kerülnek végrehajtásra. Ez bár kétirányú elágazás, HAMIS feltétel esetén nem történik semmi az elágazáson belül. Ha szeretnénk, hogy HAMIS feltétel esetén is hajtsunk végre utasításokat használnunk kell az ELSE kifejezést, mely használatakor az IF utasítás szintaktikája a következőképp változik:

```
if (feltétel) {
     …utasítások…
}else {
     …utasítások…
}
```

Az első blokkjelek közti utasítások IGAZ feltétel esetén a második blokkjelek közti utasítások pedig HAMIS feltétel esetén hajtódnak végre. Például:

```
if (kor<20){
     uzenet=”Csak 20 év felettiekre vonatkozik a kedvezmény”;
} else {
     uzenet=”A kedvezmény érvényes”;
}
```

A blokkjeleket elhagyhatjuk, de csak akkor, ha tudjuk, hogy pontosan egy utasítás áll a feltételünk vagy az ELSE utasítás után.

```
if (kor<20)
     uzenet=”Csak 20 év felettiekre vonatkozik a kedvezmény”;
else
     uzenet=”A kedvezmény érvényes”;
```

Ha több IF utasítást akarunk egymásba ágyazni, akkor hasznos lehet az IF ELSE IF szerkezet használata melynek szintaktikája a következő:

```
if (feltétel1){
     …utasítások1…
} else if (feltétel2) {
     …utasítások2…
} else if (feltétel3) {
     …utasítások3…
} else {
     …utasítások4…
}
```

Ekkor az 'else if' kifejezés az 'egyéb esetben ha' mondatnak felel meg és újra két irányra bontja szét a programunk végrehajtását.

A második utasítás melyet megemlítünk az elágazások témakörében a SWITCH lesz. A SWITCH utasítás segítségvel egy vagy többirányú elágazást hozhatunk létre. Főleg akkor szoktuk használni, ha az IF ELSE utasítás már nem elegendő a feladat elvégzéséhez, vagy az azzal való megoldás túl bonyolult lenne.

A SWITCH utasítás szintaktikája a következő:

```
switch (kifejezés){
     case érték1:
          …utasítások1…
     break;
     case érték2:
          …utasítások2…
     break;
     default:
          …utasítások3…
}
```

Elsőnek a zárójelek közé írt kifejezés értékelődik ki. Itt általában egy változót szoktunk szerepeltetni, melynek aktuális tartalma fogja eldönteni a program további haladási irányát. A kifejezés értékét összehasonlítja a program az egyes esetek (case) értékeivel sorban. Ha valamelyik eset (case) értéke megegyezik a kifejezés értékével akkor az abba foglalt utasítások végrehajtódnak, majd tovább halad a program. A break; utasítás azért szükséges az egyes esetek lezárásaként, hogy a program ne haladjon tovább a következő ágra. Ha pl. az első eset értéke megegyezne a kifejezés értékével, és az első eset végén nem használnánk a break; utasítást, akkor a program végrehajtása a kettes esettel folytatódna. Ha egyik eset értéke sem felel meg a kifejezés értékének, akkor a program a default; ágat célozza meg végrehajtásra, már amennyiben szerepel ilyen a switch-ben, ugyanis használata nem kötelező. Ha nem írtunk ilyen ágat akkor a program továbbhalad a switch utáni első utasításra. Nézzünk egy konkrét példát:

```
switch (a+b)
{
     case 4: uzenet="az összeg 4";
          break;
     case 5: uzenet="az összeg 5";
          break;
     case 6: uzenet="az összeg 6";
          break;
     default: uzenet="az összeg nem 4, nem 5 és nem 6";
}
```

22. Ciklusok, eljárások, tömbök

22.1 Ciklusok

Ha ugyanazon, vagy hasonló programrészleteket ismételnünk kell a programunkban akkor valószínű azokat egy jól szervezett ciklusba illeszthetjük. Négyféle ciklusfajtát ismer a JavaScript. Kettőt a `while` utasítás kettőt pedig a `for` utasítással írhatunk le. Vegyük őket sorba. A `for` utasítás olyan esetekben használható hatékonyan, amikor tudjuk, hogy a ciklus hányszor fog végrehajtódni, vagy legalábbis ez az érték valahol megtalálható lesz a program végrehajtódása közben.

Szintaktikája a következő:

```
for (változó=kezdőérték ; feltétel ; változó=változó+érték){
     ...utasítások...
}
```

A változót ciklusváltozónak hívjuk. A ciklusváltozó felvesz egy kezdőértéket, a feltételben a ciklus megvizsgálja minden egyes lépés előtt hogy a ciklusváltozónk értéke még megfelel-e, ha igen, akkor végrehajtja az utasításokat majd növeli a ciklusváltozó értékét a megadott érték-kel. A feltételt ciklusban maradási feltételnek hívjuk, hisz amíg igaz addig fogja végre hajtani a ciklus az utasításokat. Lássunk egy konkrét példát, egytől tízig a számok összege:

```
var ossz=0;
for (var i=1 ; i<=10 ; i++){
     ossz=ossz+i;
}
```

Másik fajta felhasználása a for ciklusnak, amikor segítségével bejárjuk egy tömb összes elemét vagy egy objektum összes tulajdonságát.

```
for (változó in objektum){
     ...utasítások...
}
```

A ciklus pontosan annyiszor fog végrehajtódni ahány elemű a tömb vagy ahány tulajdonsága van az objektumnak. A változó maga pedig tömb esetén felveszi az index értékét, azaz, hogy hányadik elemnél járunk, objektum esetén pedig a tulajdonság nevét fogja magába hordozni. Például így tudjuk kiíratni a document objektum összes tulajdonságát:

```
for (a in document){
     document.write(a);
}
```

A while utasítással megvalósított ciklusok lényege, hogy előre nem tudjuk hány lépést kell végrehajtania a ciklusnak.

```
while (feltétel)
{
     ...utasítások...
}
```

A `while` utasítás után szereplő feltételünk lesz a ciklusban maradási feltétel, azaz amíg a feltétel igaz a ciklus futni fog. A `for` ciklushoz hasonlóan ez is egy elöl tesztelős ciklus, azaz minden lépés előtt megvizsgálja a feltételt és csak akkor hajtja végre az utasításokat ha a feltétel még mindig IGAZ.
Ennek a ciklusnak létezik hasonló de hátul tesztelős változata is. Ez pedig a `do while`.

```
do{
     ...utasítások...
}while (feltétel)
```

Ekkor az utasítások egyszer biztosan végrehajtódnak majd minden lépés után a feltétel kiértékelődik, és ha igaz akkor lép újra az utasításokra a program.

22.2 Tömbök

A tömbök, azaz a több értékkel rendelkező változók létrehozása:

```
var változónév = new Array();
```
vagy
```
var változónév = new Array(érték1, érték2, érték3);
```
vagy
```
var változónév = [érték1, érték2, érték3];
```

a létrehozás után a tömb egyes elemeire a `változónév[sorszám]` -al hivatkozhatunk. Például:

```
var tomb = new Array(70, 60, 98, 77, 48, 56);
document.write("Én 19"+tomb[1]+"-ban születtem");
```

Arra ügyeljünk, hogy mint sok más programnyelvben a JavaScript-ben is az indexelés 0-tól kezdődik, tehát a fenti példában az "Én 86-ban születtem" fog megjelenni.

Ugyanígy kezeljük, ha pl. értéket akarunk adni valamelyik elemnek:
```
tomb[0]=120;
```

Ha olyan elemnek adunk értéket, amely még nem létezik a tömbben, akkor automatikusan létrejön. Például egy 3 elemű tömb esetén melyben van 0., 1., és 2. elem, ha értéket adunk a 6.-nak akkor lesz 0.,1.,2. és 6. elemünk, tehát 4 elemű tömbünk lesz.

A fenti példánál maradva: `tomb[6]=110;`
ekkor a tömbünkbe a következő elemek lesznek: 70, 86, 98, 110
amelyek sorszáma 0,1,2,6

A tömbök elemeire nemcsak egyesével kell hivatkoznunk, hanem néha végig kell járnunk azokat. Erre való a már tanult `FOR IN` szerkezet.

Előző példánkra alapozva:

```
for (srsz in tomb)
{
  document.write("Én "+tomb[srsz]+"-ban születtem<br>");
}
```

22.3 Eljárások, függvények

Eljárásokra és függvényekre akkor van szükség, amikor valamely megoldást, programrészletet szeretnénk többször is végrehajtani. Az ezekbe megírt programrészleteket hívhatjuk kisebb programoknak is, vagy ismertebb néven alprogramoknak. Eljárásokat és függvényeket a `function` kulcsszóval deklarálunk a JavaScriptben.

```
function eljárásnév(paraméter1, paraméter2....)
{
     ...utasítások...
}
```

A zárójelek közti paraméterek száma lehet 0 is de abban az esetben is ki kell írnunk a zárójelet. Az eljárásra a nevével hivatkozhatunk. Az eljárásban lévő utasítások csak akkor hajtódnak végre, ha az eljárást meghívta valamely másik alprogram vagy maga az alprogram, vagy egy esemény. Például:

```
function koszones(){
     document.write("JÓ NAPOT");
}
```

Ezzel deklaráltunk egy eljárást. Erre valahol hivatkoznunk kell majd a következőképpen:

```
koszones();
```

Ha használunk paramétereket akkor figyeljünk arra, hogy a paraméterek mint változók, csak az adott alprogramban használhatók. Nevét úgy adjuk meg, hogy az ne ütközzön semelyik főprogrambeli globális változó nevére mert az félreértésekhez vezethet.

```
function osszead(a,b)
{
     document.write(a+b);
}

szam1=5;
osszead(szam1, 66);
```

Ennek eredménye 71 lesz mivel az alprogramban az "a" változó értéke a szam1 változótól ered, azaz 5, a "b" változó értéke pedig a 66 literálból ered.

Az eljárások onnantól válnak függvénnyé, hogy eredményt közölnek, azaz van visszatérési értékük. Mivel van visszatérési értékük, így a hívás helyén is értéket képviselnek és szerepelhetnek pl. értékadó utasítás jobb oldalán. A visszatérési értéket a return kulcsszó után írjuk. Arra ügyeljünk, hogy a return utasítás azonnal visszalépteti a programot a hívás helyére, tehát az alprogramban utána lévő utasítások már nem hajtódnak végre. A fenti példa újra csak most függvénnyel (az ÜZENET már nem fog kiíródni a képernyőre):

```
function osszead(a,b)
{
return a+b;
document.write("Üzenet");
}


szam1=5;
document.write(osszead(szam1, 66));
```

23. Teszt

24. Alapobjektumok, függvények

24.1 Főbb objektumok a JavaScriptben:

`Array`	tömbkezelés
`String`	szövegkezelés
`Math`	matematikai objektum
`Date`	dátumkezelés (külön témakörben foglalkozunk vele)
`RegExp`	reguláris kifejezések kezelése

Fontosabb függvények, tulajdonságok:

`Math` objektum

Pl: `Math.PI;` `PI` értékével tér vissza

Kerekítések:
`Math.round(a);` szabályos kerekítés
`Math.ceil(a);` felfelé kerekítés
`Math.floor(a);` lefelé kerekítés

Hatványozás: `Math.pow(a,b);` a számot b hatványra emeli

Gyökvonás: `Math.sqrt(a);` a számból gyököt von

Véletlenszám: `Math.random();` egy véletlenszámot generál 0 és 1 között, ha más intervallumból szeretnénk szorozzuk fel és kerekítsük

Maximum kiválasztás: `Math.max(a,b,...);` kiválasztja a paraméterek értékeiből a legnagyobbat

Minimum kiválasztás: `Math.min(a,b,....);` kiválasztja a paraméterek értékeiből a legkisebbet

`String` objektum:

```
var s="Ez egy szöveg";
```

Karakter egy pozíción: `s.charAt(pozíció);` a pozíción lévő karaktert adja vissza az `s` változó értékéből
Karakter cseréje: `s.replace(eredeti szöveg, új szöveg);`
Az s értékében az eredeti szövegrészletet az újra cseréli és az új szöveget adja vissza

Megjegyzés: az eredeti szöveg lehet reguláris kifejezés is, és ahogy sok függvénynél itt is sokkal hatékonyabban lehet azzal megoldani a feladatokat.

Szöveg feldarabolása: `s.split(elválasztó, maximum elemszám);`
Az `elválasztó` alapján szétbontja az s értékét. Ha megadjuk a maximum elemszám értékét, akkor annál több elemet nem hoz létre a függvény. Ha elhagyjuk, akkor az egész szöveget feldarabolja. A függvény értéke a szövegrészek lesznek vesszővel elválasztva.

Kisbetűsítés: `s.toLowerCase();`
Nagybetűsítés: `s.toUpperCase();`

`Array` objektum:

```
var tomb=new Array();
```

Elem hozzáadása a tömb végére:	`tomb.push(új elem);`
Elem hozzáadása a tömb elejére:	`tomb.unshift(új elem);`
Utolsó elem lekérdezése és eltávolítása:	`tomb.pop();`
Első elem lekérdezése és eltávolítása:	`tomb.shift();`
Elemek sorrendjének megfordítása:	`tomb.reverse();`
Elemek rendezése:	`tomb.sort();`
Tömb részletének lekérdezése:	`tomb.slice(mettől, meddig);`

A második paraméter elhagyható, ilyenkor az első paraméter értékétől a tömb végéig kapjuk meg az elemeket.

24.2 Reguláris kifejezések

A reguláris kifejezések karaktersorozatok. Olyan maszkok melyek speciális paraméterek alapján el tudják dönteni egy szövegről hogy mely elemei hasonlítanak a maszkra. A reguláris kifejezések segítségével összehasonlításokat végzünk.

A reguláris kifejezés elemei:

A reguláris kifejezéseket legegyszerűbben úgy írhatjuk le, ha a // jelek közé írjuk őket. A / jelek után már csak a módosító paramétereket tüntethetjük fel. Ezek:

`g` Globális keresés. Az összes lehetőséget megkeresi.
`i` Kis-nagybetűre nem érzékeny keresés.
`m` Többsoros keresés.

A // jelek közé írható elemek:

`szöveg`	A szöveget egy az egyben keresi
`[xyz]`	A szögletes zárójelben felsorolt karakterek, vagy tartományok meglétét keresi
`[^xyz]`	A kalapjel után álló karaktereken, vagy tartományokon kívüli értékeket keresi
`[víz|vas|vág]`	A lehetőségek valamelyikét keresi

`.`	A pont helyettesít bármilyen karaktert. Ha tényleg a pont karaktert keressük tegyük elé a \ jelet. \.
`\w`	Alfanumerikus karaktereket keres beleértve az aláhúzó jelet is
`\W`	A nem alfanumerikus karaktereket keresi
`\d`	Számokat keres
`\D`	A nem számokat keresi
`\s`	Az úgy nevezett whitespace karaktereket keresi. Szóköz, új sor, tabulátor stb.

\S	A nem whitespace karaktereket keresi
\b	Szó elején vagy végén keresi a karaktersorozatot
\B	A karaktersorozatot a szóban keresi, nem az elején vagy a végén
\n	Új sor karaktert keres
\r	A kocsivissza karaktert keresi
\t	A tabulátor karaktert keresi

Módosítók:

+	Minimum egy találatot keres
*	Nulla vagy több találatot keres
?	Nulla vagy egy találatot keres
{h}	h hosszú sorozatát keresi a karaktereknek
{h,}	minimum h hosszú sorozatát keresi a karaktereknek
$	A szöveg végén keres
^	A szöveg elején keres. Nem szabad összetéveszteni a [] közti kalapjellel

Példák a szövegkezelő MATCH függvény segítségével bemutatva:

Email cím ellenőrzése:

```
var regkif=/\w[\w|.]*@\w[\w|.]*\.\w+/;
document.write(email.value.match(regkif));
```

TAJ számhoz hasonló 000-000-000 formátum ellenőrzése:

```
var regkif= /\d{3}-\d{3}-\d{3}/;
document.write(email.value.match(regkif));
```

Bonyolultnak tűnhet első ránézésre, de ha végignézzük többször egyesével, akkor rájövünk a nyitjára.

25. Események

Az események olyan elemek a JavaScriptben melyeket a böngésző vagy a felhasználó aktivál. Ha megtörténik egy ilyen esemény, a JavaScript észleli azt. Mindegyik a JavaScript által ismert eseményhez állíthatunk be valamilyen műveletet, melyet csak akkor fog végrehajtani a program, ha az esemény ténylegesen be is következik. Az eseményfigyelőket és az események hatására végrehajtódó utasítást (legtöbbször egy eljáráshívást) a HTML elemek attribútumai közé helyezzük el.
Ekkor lényegében azt is meghatározzuk, hogy melyik HTML elem eseményeit figyeljük. Például egy űrlap gombján figyeljük az egérkattintást:

```
<input type="button" onclick="alert('Helló');">
```

Az eseményeket csoportba bonthatjuk az alapján, hogy mivel kapcsolatos.

25.1 Egér események

`onmouseover`	az egérkurzor az aktuális elem felett áll
`onmouseout`	az egérkurzor az aktuális elem felett állt és elmozdult róla
`onmousemove`	az egér megmozdult
`onmousedown`	az egér gombja le lett nyomva az elem felett állva
`onmouseup`	az egér gombja el lett engedve az elem felett állva
`onclick`	kattintottak az egérrel az elemre
`ondblclick`	duplát kattintottak az egérrel az elemre

25.2 Billentyűzet események

`onkeypress`	egy gombot lenyomtak, vagy nyomva tartottak
`onkeydown`	egy gombot lenyomtak
`onkeyup`	egy gombot felengedtek

26. Dokumentum események

`onload`	az aktuális oldal vagy kép betöltődött
`onunload`	az oldalt elhagyta a felhasználó
`onerror`	hiba történt a dokumentum vagy a kép betöltődésekor

26.1 Egyéb események

`onabort`	a kép betöltése megszakítva
`onblur`	az elemről lekerült a fókusz
`onfocus`	az elemre rá került a fókusz
`onchange`	az elem értéke megváltozott
`onresize`	az elemet átméretezték (pl. ablak)
`onselect`	a szöveget kijelölték

Ezeket az eseménykezelőket felhasználva tehetjük az oldalunkat dinamikusabbá, interaktívabbá. Nézzünk néhány példát:

Az oldal betöltődésekor üdvözöljük a felhasználót:

```
<body onload="alert('Üdvözöljük weblapunkon!');">
```

Az oldal elhagyásakor elköszönünk:

```
<body onunload=”alert(’Látogasson el hozzánk máskor is!’);”>
```

Kép cseréje ha az egér kurzor felette áll:

```
<img src=”kutya.jpg” onmouseover=”this.src=’macska.jpg’;”>
```

27. Dátum és idő kezelése

A dátum és idő kezelésére a JavaScript-ben egy beépített objektum áll a rendelkezésünkre. Ez a `DATE` objektum. Segítségével lekérdezhetjük az aktuális dátumot és időt, hozhatunk létre új dátumidő típusú változókat, és műveleteket végezhetünk velük.

Az egyik legegyszerűbb művelet az aktuális dátum és idő lekérdezése. Ehhez a `DATE` objektumot kell csak meghívni, mint metódust, a következőképpen:

`document.write(Date());` így pl. kiíratjuk az aktuális dátumot

Ugyanez változóval:

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

Ekkor a változó már egyben dátum típusú is lesz, és tartalmazni fogja az aktuális dátumot és időt. Hozhatunk létre úgy is dátum típusú változót, hogy annak értékét mi határozzuk meg:

```
var d=new Date(dátum szövegesen megadva)
var d=new Date(év, hónap, nap, óra, perc, másodperc,
ezredmásodperc);
var d=new Date(másodperc) 1970.01.01-től eltelt másodpercek
száma
```

példák a fenti sorrendet alapul véve:

```
var d=new Date("2010/03/10 13:00:33");
var d=new Date(2003,0,20,12,10,33,0);
```
ahol a második paraméter a hónap 0-tól számozódik

```
var d=new Date(1034555444045);
```

A dátum típusú változókkal a beépített metódusok segítségével tudunk műveletek végezni, valamint azok értékeit megváltoztatni, lekérdezni.

Értékek lekérdezése:

```
var d=new Date();
```

`d.getFullYear();`	az évet adja vissza négy számjeggyel
`d.getMonth();`	a hónapot adja vissza 0-11-ig
`d.getDate();`	a hónap napját adja vissza 1-31-ig
`d.getDay();`	a hét napját adja vissza 0-6-ig
`d.getHours();`	az órát adja vissza 0-23-ig
`d.getMinutes();`	a percet adja vissza 0-59-ig
`d.getSeconds();`	a másodpercet adja vissza 0-59-ig
`d.getTime();`	1970. január 1-től eltelt másodpercek számát adja vissza

Például az aktuális dátum kiíratása az ismert formátumban, pontokkal elválasztva:

```
var d=new Date();
document.write(d.getFullYear()+”.”+(d.getMonth()+1)+”.”+d.getD
ate());
```

Értékek beállítása:

```
var d=new Date();
```

`d.setFullYear(év);`	az évet állíthatjuk be
`d.setMonth(hónap);`	a hónapot állíthatjuk be 0-11-ig
`d.setDate(nap);`	a hónap napját állíthatjuk be 0-31-ig
`d.setHours(óra);`	az órát állíthatjuk be 0-23-ig
`d.setMinutes(perc);`	a percet állíthatjuk be 0-59-ig
`d.setSeconds(másodperc);`	a másodpercet állíthatjuk be 0-59-ig
`d.setTime(másodperc);`	1970. január 1-től eltelt másodpercekkel megadva állíthatjuk be a dátumot számát adja vissza

Az értékeket származtathatjuk a változó eredeti értékéből is. Így használjuk az érték lekérdező és beállító metódusokat is. Például a d által tárolt dátumhoz hozzáadunk 3 napot.

```
d.setDate(d.getDate()+1);
```

vagy 52 percet:

```
d.setMinutes(d.getMinutes()+52);
```

Csalóka lehet, hogy a `setMinutes` paramétere így átlépheti a megadott 59-es határértéket, de a JavaScript szerencsére ezeket lekezeli, és automatikusan átállítja az új dátum összes értékét, például óra, nap, hónap stb. váltásokat.

Ehhez a témakörhöz kapcsolódik az időzítés témaköre. Sokszor előfordul, hogy egy eseményt, utasítások végrehajtását, csak bizonyos idő múltán szeretnénk végrehajtani, vagy bizonyos időközönként. A `setTimeout` utasítás segítségével ezek nagyon egyszerűen megoldhatók. Szintaktikája:

```
setTimeout(kódrészlet, ezredmásodperc);
```

Ha a JavaScript ehhez az utasításhoz ér automatikusan elkezdi visszaszámlálni az ezredmásodperc paraméterben megadott időt és ha az idő lejár a háttérben végrehajtja a kódrészlet paraméterben megadott utasításokat mely általában egy függvényhívás.

Például egy 10 másodperces késleltetéssel megjelenő felugró ablak amiben üdvözöljük a felhasználót:

```
function udv()
{
     alert('Üdvözöllek');
}
setTimeout('udv()',10000);
```

Időközönként végrehajtott utasításokat úgy tudunk vele megoldani, hogy a meghívott függvényben újra aktiváljuk az időzítőt, így újra és újra meghívódik a függvény a beállított időközönként. Például egy visszaszámláló készítése:

```
<body onload='szamol();'>
<script>
var szamlalo=10;
function szamol(){
     document.dobozform.doboz.value=szamlalo;
     if (szamlalo!=0){
          szamlalo--;
          setTimeout('szamol()',1000);
     }else{
          alert("BUMMMMMM!");
     }
}
</script>
<form name="dobozform">
<input type="text" name="doboz" readonly="readonly">
</form>
</body>
```

Az oldal betöltődésekor meghívódik a `szamol()` függvény, és amíg a számláló nem lesz nulla újra és újra meghívja magát pontosan 1 másodperces időközönként.

28. Űrlapkezelés

Amikor egy űrlapot készítünk weboldalunkra a HTML elemek segítségével, azokat az elemeket fogjuk kiválasztani az adatok bekéréséhez, amelyek a legjobban passzolnak az adott érték típusához. Példádul jelszavaknál jelszómező, választási lehetőségeknél rádiógomb, és így tovább. Azonban nagyon sok olyan adat van, melyekhez nem tudunk külön elemet megjelölni és egyszerűen szöveges adatként kezeljük őket. Ilyenek például a felhasználónevek, email címek, dátum adatok stb. A JavaScript nagyon jó eszköz lehet arra, hogy az űrlapadatokat mielőtt átadjuk egy programnak feldolgozásra, még megvizsgáljuk és validáljuk, azaz eldöntsük, hogy számunkra az adatok megfelelőek, avagy nem.

Először is, hogyan tudjuk megoldani, hogy az űrlap elküldése előtt valamely alprogramunk még megvizsgálja az adatokat:

```
<form name=”urlap” method=”post” onsubmit=”return vizsgal();”>
```

Ez esetben a meghívott alprogram IGAZ vagy HAMIS visszaadott értékétől függ, hogy az űrlap végül elküldésre kerül vagy nem.

Mit vizsgálunk:

Az egyes űrlapelemek vizsgálatához el kell érnünk azok értékeit JavaScriptből. Vagy azonosítókkal látjuk el őket és az alprogramon belül annak segítségével találjuk meg,

```
<script>
function vizsgal(){
      var elem=document.getElementById("doboz");
     …
}
</script>
<form name="dobozform" onsubmit=”return vizsgal();”>
<input type="text" id="doboz" name="barmi"
readonly="readonly">
</form>
```

Az elem *nevű változó egy objektum lett mely a szöveges dobozunkra hivatkozik.*

vagy mivel az űrlapunkon úgyis több elem lesz, a másik lehetőség, hogy átadjuk a vizsgal() függvényünknek magát az űrlapot és annak elemeit pedig a nevükre hivatkozva érjük el.

```
<script>
function vizsgal(urlap){
      var elem=urlap.email;

}
</script>
<form name="dobozform" onsubmit=”return vizsgal(this);”>
<input type="text" id="doboz" name="email"
readonly="readonly">
</form>
```

Az elem *nevű változó most is egy objektum lett mely a szöveges dobozunkra hivatkozik.*

Már tudjuk mit vizsgáljunk, nézzük meg hogyan vizsgáljuk. A szöveges beviteli mezők értékét a value tulajdonsággal tudjuk lekérdezni. Legfontosabb vizsgálat, hogy a mezőt üresen hagyták-e.

```
function vizsgal(urlap){
     var elem=urlap.email;
     if ((elem.value=””) || (elem.value=null)){
          return false;
     }else{return true;}
}
```

A bevitt szöveg hosszát és általában egy sztring hosszát a `length` tulajdonság adja meg. Ne engedjünk 6 karakternél rövidebb szöveget elküldeni:

```
function vizsgal(urlap){
     var elem=urlap.email;
     if (elem.value.length<6){
          return false;
     }else{return true;}
}
```

Bonyolultabb értékek vizsgálatánál hasznos lehet az `indexOf()` a `lastIndexOf()` és a `substr()` objektummetódus. Az `indexOf(keresett szöveg, kezdőpozíció)` és a hasonló paraméterekkel rendelkező `lastIndexOf` a kezdőpozíciótól kezdve keresi meg a keresett szöveg előfordulását a változó értékében. A különbség hogy az `indexOf` az első a lastIndexOf az utolsó előfordulást adja vissza. Ha nem található meg a keresett szöveg, akkor mindkettő 1-el tér vissza.

```
var s=”Kispajtás”;

document.write(s.indexOf(„s”));          eredmény: 2
document.write(s.lastIndexOf(„s”));      eredmény: 8
```

A `substr(kezdőpozíció,hossz)` a kezdőpozíciótól, hossz hosszan kivágja a szövegrészletet és azt adja vissza értékül.

```
document.write(s.substr(3,3));          eredmény: „paj”
```

Például email cím egyszerű ellenőrzése ezek segítségével:

```
function vizsgal(urlap){
     var elem=urlap.email;
     if (elem.value.indexOf("@")<1)
     {
          return false;
     }
     else if (elem.value.indexOf(".")<3)
     {
          return false;
     }
     else if
(elem.value.indexOf("@")>elem.value.lastIndexOf("."))
     {
          return false;
     }else {return true;}
}
```

Láthatjuk, hogy ez a megoldás sem teljes, de egyfajta megoldásnak megfelel.

Dátumok ellenőrzését is megoldhatjuk. A legegyszerűbb mód, ha a beírt dátumot megpróbáljuk átkonvertálni dátum típusra, és ha ez sikerül, akkor elfogadjuk a dátumot. A számunkra elfogadott dátumformátum a 2010.03.14 ezért a felhasználó által beírt értékeket szétvágjuk a substr segítségével majd a JavaScript által ismert 2010/03/14 formában megpróbáljuk átváltani, és ha sikerül, akkor helyes dátumot adott meg a felhasználó.

```
function vizsgal(urlap){
     var ev=urlap.email.value.substr(0,4);
     var ho=urlap.email.value.substr(5,2);
     var nap=urlap.email.value.substr(8,2);
     var d=new Date(ev+"/"+ho+"/"+nap);
     if (isNaN(d)){
          return false;
     }
     return true;
}
```

Az `isNaN` függvény IGAZ értéket ad vissza, ha az átadott paraméter definiálatlan.

Ha szeretnénk a felhasználónak jelezni, melyik űrlapelem a hibás egyszerűen a CSS segítségével változtassuk meg a háttérszínét:

```
urlap.email.style.backgroundColor='yellow';
```

Hasznos lehet még egy számláló készítése mely mutatja, hogy hány karaktert ütöttünk le például egy TEXTAREA elemben.

```
function szamol(area){
     document.getElementById("szamlalo").value=(area.value.leng
th);
}
</script>
<form name="dobozform" onsubmit="return vizsgal(this);">
     <input type="text" id="szamlalo" name="szamlalo">
     <textarea name="uzenet" onkeyup="szamol(this)"></textarea>
</form>
```

29. Dinamikus tartalom.

29.1 Tartalom váltása a lap újratöltése nélkül

Egy HTML oldal betöltésekor a böngésző az adott HTML kódot értelmezve megjeleníti az abban felsorolt elemeket a megadott sorrendben és kialakítja az oldal szerkezetét. Ha a betöltés után az oldal tartalma nem változik, akkor statikus lapnak nevezzük. Sok weboldalnál szerver oldali kódok döntik el a HTML kód felépítését, de a végeredmény a böngészőben akkor is egy statikus weblap lesz. Akkor nevezünk egy weboldalt dinamikusnak, ha a tartalma, szerkezete változik a felhasználó beavatkozására, vagy bármilyen más esemény hatására.

Mivel JavaScript-ből elérjük a HTML elemeket, mint objektumokat, lehetőségünk van azok értékeit is a kódból megváltoztatni. Nézzünk meg pár ilyen megoldást:

A kép elem

A kép elem egyik legfontosabb attribútuma az src, azaz a kép elérési útja. Ennek a paraméternek a változtatásával dinamikusan tudjuk az adott „kép” - vagy inkább nevezzük a kép tárolójának – forrását megváltoztatni.

```
<img src=”viragok.jpg” onmouseover=”this.src=’viragok1.jpg’;”
onmouseout=”this.src=’viragok.jpg’;”>
```

Bővebb tartalom megváltoztatásához használjuk a DIV elemeket, vagy úgyis mondhatnánk dobozokat.

A DIV elem

A DIV elemeknek van egy tulajdonságuk, mellyel a saját belső tartalmukra hivatkozhatunk. Ez az `innerHTML`. Ennek értékét megváltoztatva magát a DIV tartalmát változtathatjuk meg. Nézzünk egy egyszerű példát a használatára:

```
<script>
function mutat(sorszam){
  var elem=document.getElementById("cikkdoboz");
  switch (sorszam){
      case 1:elem.innerHTML="<p align='justify'>Ez az első cikk szövege
                             lesz, melyben HTML elemeket is
                             feltüntethetünk, hogy formázzuk a megjelenő
                             szöveget.</p>";
      break;
      case 2:elem.innerHTML="<p align='justify'>Ez a második cikk szövege
                             lesz, ide is írhatunk HTML elemeket akárcsak
                             az elsőhöz.</p>";
      break;
  }
}
</script>

<a href='#' onclick="mutat(1);">Első cikk</a>
<a href='#' onclick="mutat(2);">Második cikk</a>

<div style="width:300px;border:1px solid black;background-color:#ae43f5;"
id="cikkdoboz"></div>
```

A működése egyszerű. A HTML elemek közé elhelyezünk egy kitüntetett DIV elemet. Adunk neki egy azonosítót. Példánk esetében ez `cikkdoboz`. Beállíthatunk neki egy kis stílust a megtanult CSS segítségével, majd valamilyen esemény hatására - jelen esetben két különálló szövegre való kattintás – meghívunk egy eljárást, mely megkeresi a kitüntetett elemet, és a sorszámtól függően változtatja meg az `innerHTML` tulajdonság értékét. A lap tartalma, ezáltal dinamikusan fog változni.

CSS segítségével

A CSS-ből tanultak alapján tudjuk, hogy egy HTML elem egyik tulajdonsága a megjelenése. Ezt a `display` CSS tulajdonság tárolja. Egy elem stílusát JavaScriptből pedig az elem `style` mezőjén keresztül érjük el. Például egy `doboz` azonosítóval ellátott elem display CSS mezőjét így érjük el:

```
var elem=document.getElementById(„doboz”);
elem.style.display=”none”;
```

Ügyeljünk arra, hogy az értékek idézőjelbe kerüljenek. Jelen esetünkben a doboz megjelenését kikapcsoltuk. Ez alapján már elkészíthető a fenti példához hasonló, cikkek dinamikus váltását megvalósító kód. Lássuk ezt:

```
<script>
var maxcikk=3;
function mutat(sorszam){
  for (var i=1; i<=maxcikk ; i++){
    var elem=document.getElementById("cikk"+i);
    if (i==sorszam){
         elem.style.display="block";
    }else{
        elem.style.display="none";
    }
  }
}
</script>

<a href=# onclick="mutat(1);">Első cikk</a>
<a href=# onclick="mutat(2);">Második cikk</a>
<a href=# onclick="mutat(3);">Második cikk</a>

<div style="display:none;width:200px;background-color:yellow;"
              id="cikk1">Első cikk szövege</div>

<div style="display:none;width:200px;background-color:olive;"
              id="cikk2">Második cikk szövege</div>

<div style="display:none;width:200px;background-color:blue;"
id="cikk3">Harmadik cikk szövege</div>
```