JS2 – A Beginning JavaScript könyv gyors jegyzetei

Kódolási zsinórmérték:

- szellős kód
- változókat ott definiálni, ahol először használjuk,
- csak a szükséges mennyiségű, helyi változó,
- rendszeres ellenőrző kód-részletek
  [„Trükkös Iván”: rendszeresen iktassunk be ellenőrző-feltételeket (pl. objektumok támogatásának vagy változók létezésének ellenőrzésére), hogy a kód hibás futására idejekorán fény derüljön, ill. ennek megakadályozására.]
- console.log(); parancs és Firebug konzol alkalmazása [alert(); helyett]!
- cca. 50 sornál hosszabb kód esetén elkülönítés új dokumentumba!
- Feltétlenül közöljük a felhasználóval, hogyha egy link vagy eseménykezelő új ablakra vagy lapra lép, az esetleges félreértések elkerülésére!
- Ne továbbítsunk formokat, iránytísunk át oldalakat a felhasználó tudta nélkül (automatikusan)!
- Kerüljük az olyan funkciók programozását, melyek a böngészőkben eleve megtalálhatók, mert azok általában többféle beviteli eszközzel, könnyebben használhatók! (Pl. „Vissza” gomb.)
- Ne „animáljuk” túl az oldalt, a statikusan vagy CSS-oldalról elérhető hatásokat ill. a böngésző alapértelmezett funkcióit használjuk inkább!
- Rendszeresen ellenőrizzük a böngészőben (a JS kikapcsolásával), hogy az oldal JS-támogatás nélkül is használható-e!
- Az elrejtett tartalmat is le kell töltenie a felhasználónak; takarékoskodjunk a letöltendő adat-mennyiséggel!
- Inkább meglévő adatstruktúrákat dinamizáljunk, és ne magával a JS-tel generáljuk a tartalmat, menüket, egyéb passzív vagy aktív elemeket!
- A .focus(); ill. .blur(); módszereket csak a legszükségesebb esetben alkalmazzuk, mert megbontájk a formok tabulálásának alapértelmezett hieracrhiáját, ami a kitöltést nehezíti!

1. fejezet

JS-mondattan:

- A weboldalakba írandó JS-et a <script> tag tartalmazza:
  <script type="text/javascript">
  // Your code here
  </script>
- Hogy a kódot véletlenül se hajtsa végre a böngésző, a HTML comment tagbe kell foglalni:
  <script type="text/javascript">
  <!--
  // Your code here
  -->
  </script>
- A HTML legújabb és legszigorúbb változata, az XHTML szerint a kommenteléshez a CDATA kommentelési mondattant kell alkalmaznunk:
  <script type="text/javascript"><!--//--><![CDATA[//><!--
  // Your code here
  //--><!]]></script>
  Emellett az XHTML értelmében a JS-et csakis külön .js fájlban kellene tárolni!
- A // jel utáni sor-rész komment.
- A /* és */ jelek közötti szöveg komment.

Kód-végrehajtás:

- A {} jelek közé írt kód egy tömbként kerül végrehajtásra.
- Az egyes rendelkezéseket pontosvesszőkkel (;) vagy sortörésekkel választhatjuk el.
- A kód (ill. kód-blokkok) végrehajtási sorrendje megegyezik a fentről-lefele – balról-jobbra olvasási iránnyal. Azaz az egyes script tagekben vagy a .js dokumentumban foglalt kódokat ilyen sorrendben olvassa be a böngésző.
- A screen.availWidth ill. screen.availHeight változók a felhasználó képernyőjének szélességét ill. magasságát szolgáltatják (pixelekben).

Függvények:

- A függvények ismételt meghívásával sok manuális kódbeírást spórolhatunk meg. A függvények/módszerek tehát bizonyos feladatok, eljárások tárolására valók.
- Függvények definiálása: function *függvény-név*(*paraméterek*) {*kód-blokk*;}
- Függvények meghívása (a helyzettől függő paraméterekkel): *függvény-név*(*paraméterek*);

Objektumok

- Az objektumoknak (objects, melyek a JS-nyelv „főnevei") vannak jellemzői (properties, mint „melléknevek"), módszerei (methods, mint „igék") és eseményei (events).
- Az Objektum-orientált Programozás (OOP) a JS és a webes programozás lényege.
- Az ún. mag (core) objektumok előre megadottak, implementációtól függetlenek. Sok programozást spórolhatunk meg a használatukkal.
- A saját objektumok megadásához az ún. class-t használjuk.

2. fejezet

Adat-típusok:

- Az adatok (data) a JS információi, melyeket a típusuknak (pl. String/Number/Boolean) megfelelően kezel a nyelv.
- A null adat-típus az adat hiányát, az undefined pedig az adat (pl. változó-érték) meghatározatlanságát jelzi.

Stringek:

- A string-adatokat idézőjelekbe (vagyis ún. delimiterek közé) kell írni. Az idézőjel tetszőlegesen lehet egyszeres vagy kétszeres, a lényeg, hogy az adott elem elején és végén is ugyanolyan legyen. Idézetek vagy aposztrofok egymásba ágyazásakor a bennfoglaló idézőjel és a bent lévő eltérő legyen! Jó megoldás az is, hogyha ún. escape sentence-eket használunk (kódolt karaktereket) az idézőjelek helyett, pl. \' ill. \".
- Az ASCII-betűk kódja: \x*ff*, ahol ff hexadecimális szám.
- Az Unicode-betűk kódja \u*ffff*, ahol az utolsó négy karakter két hexadecimális számnak felel meg.

Műveletek:

- A műveletekkel (operators) módosíthatjuk az adatokat. Az alkalmazott műveletek rendűsége és zárójelekkel való csoportosításuk a matematika szabályainak megfelelően történik.

Változók:

- A kód áttekinthetősége érdekében a változókat lehetőleg a program elején definiáljuk, ill. esetleges (újra)definiálásokat a függvényeken belül végezhetünk.
- A JS esetfüggő; a változó-nevek pedig számmal nem kezdődhetnek.

Adat-típusok és átalakításuk:

- A prompt módszer/függvény a window objektumra vonatkozik. Két paramétere a megjelenítendő szövegre és a szövegmező alapértelmezett értékére utal: window.prompt(”szöveg”,”érték”);
- A prompt-ként bevitt adatokat a program string-ként kezeli, így a + művelet nem összeadást, hanem hozzáfűzést jelent akkor is, ha a bevitt adat szám. A többi művelet azonban nem kétértelmű, így azokat szám-tartalmú stringre is matematikai műveletként hajtja végre a program; felismerve, hogy a string számként is értelmezhető. (Ellenkező esetben a művelet értéke NaN!)
- A typeof(); módszer stringként visszaadja a paraméterébe írt változó értékének típus-megnevezését (number/string/boolean/array stb.).
- A JS-et rákészszeríthetjük, hogy a bevitt adatot számként értelmezze. Ennek módjai:
  - A Number() függvény paraméterébe írt értéket megpróbálja számmá alakítani.
  - A parseFloat() függvény elkezdi beolvasni a stringet, egészen addig, míg egy nem-szám karakterhez ér. Az eddig beolvasott számokat értelmezi és visszaadja. Fontos, hogy a string számmal, vagy + ill. – jellel kezdődjön!
  - A parseInt() függvény a parseFloat()-hoz hasonlóan működik, de a képezhető számnak csak az egészértékét veszi fel (a tizedeseket elhagyja).

Összetett adat-típusok: az array-k és objektumok.

- Az összetett adat-típusok:
  - Az objektumok, melyet lehetnek alapértelmezettek vagy a felhasználó által készítettek.
  - Arrray-k, melyek egy vagy többféle más típusú adatot tartalmaznak.
- Alapértelmezett JS-objektumok pl.: String, Date, Math.

A string objektum:

- Stringek deifiniálási módszerei:
  - Implicit (=járulékos) módszer: egy változóhoz szöveges értéket rendelünk; pl. var változónév = ”érték”;
  - Explicit (=direkt) módszer: a String konstruktorral ténylegesen létrehozzuk a stringet, mint objektumot: var változónév = new String (”érték”);
    Az explicit meghatározás szabatosabb, általánosabb érvényű.
- A string objektumokra vonatkozó módszerek közül kettő: indexOf() és substring().
- A változó-név.indexOf(”keresett szöveg”); paranccsal a keresett szöveg karakterének helyszámát adja vissza a módszer (függvény). A string első karakterének helyszáma 0. A szóközök is karakternek számítanak.
- A változó-név.substring(kezdő-index,vég-index); módszer a kezdő-indexű karaktertől a vég-indexű előttiig terjedő al-stringet veszi fel értékként.

A Date objektum:

- A JS-ben nincs primitív (implicit) dátumadat-típus, ezért a dátum-adatok csak expliciten, azaz objektumként definiálhatók [new Date() objektum].
- Hogyha pl. a dátumnév.setDate(100) módszerrel egy dátumhoz túl nagy értéket rendelünk, a JS a dátum többi elemét is megváltoztatja, az eredeti dátum és a megadott érték által kifejezett időtartamnak megfelelően.

A Math objektum:

- A Math objektumot csak implicit módon, azaz a használatával „definiálható”, s a string és Date objektummal ellentétben nem tárol adatot.

Az Array objektum:

- Az array-elemeket névvel is elnevezhetjük, pl.:
  var array-név = new Array();
  array-név [”elem_neve”] = ”érték”;

Feltétel-vizsgálatok és feltételes rendelkezések:

- Az array-k elemeinek ABC-sorrendbe állítása, valamint a stringek közötti egyenlőtlenségi relációk során a JS az őket alkotó karakterek ASCII-számait veszi alapul!
- A feltétel-vizsgálatokkor szöveges értékek esetén az ASCII-számok alapján dönt a program. Ha az ABC-sorrendnek megfelelő, esetfügetlen összehasonlításra van szükségünk, akkor az értékeket célszerű a toUpperCase() vagy toLowerCase() függvényekkel egységes betű-esetűvé alakítani. (47.)
- Hogyha objektumok (pl. explicit módon definiált stringek) közt állíptunk meg relációkat, azok magukra az objektumokra, s nem azok értékére vonatkozólag kerülnek kiértékelésre. Ezért ha a változókat ovjektumként definiáltuk, akkor értékeik összehasonlításához a változó.valueOf() módszert kell rájuk alkalmazni. (47-48.)
- Logikai műveletek a feltétel-vizsgálatokban: a feltételek teljesülésének egymás közötti, logikai viszonyának megfelelően megállípíthatjuk, hogy a felhasználótól kapott adat értelmes-e,m illetve a korábbi választásoknak megfelelően csökkenthetjük a további alternatívák számát. (48.)
- Az értékek nem-szám (NaN) jellegének ellenőrzésére az isNaN(érték) függvény szolgál, melynek értéke true, ha az érték NaN, és false az ellenkező esetben. Objektumok értékeinek vizsgálatához ne felejtsük el haszálni a valueOf() módszert! (49.)
- A break; parancs az adott (pl. if else) feltételes rendelkezés kód-blokkjának végrehajtását leállítja, és a következő kód-blokkra téríti át a programot.
- Akkor, hogyha egy feltétel után az ÉS (&&) logikai művelet áll, a vizsgálat csak akkor folytatódik, hogyha az ÉS-feltétel teljesült. Egyébként a program nem lép be a ciklusba.
- A switch feltételes rendelkezés olyan, mint a tirisztor. Hogyha az egyik case teljesül, akkor az összes következő parancsot végrehajtja a program; vagyis a case kulcsszavakat többé nem veszi

figyelembe. Ezért kell a break; paranccsal kilépni a ciklusból az adott esetben végrehajtásra kerülhető összes parancssor végén.

- A for ciklust alkalmazzuk, ha a kód-részletet bizonyos számú alkalommal kell futtatni.
- A while ciklust alkalmazzuk, hogyha egy feltétel teljesülése esetén akarjuk futtatni.
- A do while ciklust, hogyha a feltétel teljesülésének ellenőrzése előtt egyszer mindenképpen le akarjuk futtatni a kódot.
- A break; utasítással a teljes cillusból kilépünk, úgy, hogy az többé nem indul el. A continue; parancs a ciklust az adott helyen leállítja majd elölről újraindítja.

Ciklusok:

- A változó-név.length() módszer segítségével megállapíthatjuk egy array elem-számát vagy egy egyszerű változó karaktereinek számát.
- A while ciklus a kód végrehajtása előtt ellenőrzi a feltétel teljesülését (és true esetén végrehajtja azt), míg a do while ciklus a kódot először lefuttatja, majd a végén ellenőrzi a feltételt, és míg azt érvényesnek találja, megismétli a kód futtatását.

3. fejezet

DHTML

- A DHTML nehezen és drágán szervizelhető/fenntartható, korlátolt kompatibilitású, sok kód-ismétlést igénylő weboldal-dinamizálási megoldás volt, ami inkompatibilitás esetén az oldalt használhatatlanná tette. Ezért mára teljesen kikopott, felváltotta a JS.

A JS mint behavior layer:

- A webhelyek rétegei:
  - Behavior layer: a felhasználó egyes akciói esetére előírt reakciók (JS, ActionScript, Flash)
  - Presentation layer: a weboldal forma- és színvilága (CSS)
  - Structure layer: a weboldal szöveges és képi struktúrája (XHTML)
  - Content layer: a weboldalban felhasznált szöveges, képi, multimédiás- és számadatok összessége (XML, adatbázis, médiaeszközök).
  - Business logic layer (back end / logistics): a szerver által fogadott és küldött adatok kezelési módja.
- A szakszerű weboldal-fejlesztés lényege e rétegek elkülönített, de mégis együttműködő formában történő létrehozása.
- A DHTML fájlokban a content és behavior rétegek elválaszthatatlanul össze voltak keverve, meggátolva ezzel az egyszerű szervizelést.

Objektum-felismerés: a böngésző-függőség ellenszere

- If feltétel-vizsgálatokkal megnézzük, hogy a böngésző ismeri-e azt az objektum-típust, amit az adott JS-kódolás jelent. Hogyha igen, akkor a feltételes rendelkezés végrehajtásra kerül; ellenkező esetben a rákövetkező kód-variánst tartalmazó feltételes rendelkezést vizsgálja meg a program. A böngésző-függő objektumok, melyek létezését keressük: IE=document.all; Netscape=document.layers; Mozilla, Opera, Safari=document.getElementById && document.createTextNode.
  A vizsgálat kizárásos alapon is történhet, pl. egy ehhez tartozó if-feltétel:
  if (!document.getElementById || !document.createTextNode){return;}
  Ebben az esetben a program meg sem vizsgálja a további feltételeket; tehát már semmi esetre sem a Mozilla-típusú böngészőknek szánt kódot fogja beolvasni.

Fokozatos erősítés (progressive enhancement)

- Az újabb és újabb funkcionalitások futtatása előtt ellenőrizzük, hogy a felhasználó programja támogatja-e azokat; hogyha nem, akkor is tökéletesen kell működnie az oldalnak (az adott szinten).
- Funkcionalitási szintek:
  - Mondattanilag helyes XHTML dokumentum, benne az összes szöveges, kép (alt attribútum!) stb. tartalommal.
  - Külső CSS dokumentum a megjelenés, olvashatóság/áttekinthetőség javítására és egyszerűbb egérérintési hatások létrehozására.
  - JS hozzáadása:
    - A JS a window objektum onload eseményekor kezd futni.
    - Ellenőrzi, hogy a felhasználó támogatja-e a W3C DOM-ot.
    - Ellenőrzi a felhasználandó objektumok létezését, és felruházza őket az előírt funkcionalitással.

Hozzáférhetőségi alapszabályok:

- A JS teljes mellőzése nem lehet megoldás a kezelési/megjelenítési problémákra, ezért a következő elvek szerint dolgozzunk:
  - A web-dokumentum JS-tel és anélkül látható tartalma legyen megegyező.
  - Azokat az elemeket, melyek csak a JS működése esetén fontosak (pl. a JS működését segítő feliratokat, útmutatókat), magával a JS-tel generáltassuk.
  - A JS-funkciók a beviteli eszköztől függetlenek legyenek.

    - A linkeken és formokon kívül más nem interaktív (statikus) elemeknek ne adjuk funkciót! Pl. ha egy header-re klikkelve le kell nyílnia egy fülnek, akkor a billentyűzetet használó egyén nem tudja azt kijelölni a TAB gombbal. Ha azonban a header egyben link is, már működik így is.
    - Felhasználói interakció nélkül ne irányítsuk át a böngészőt más oldalra és ne továbbítsunk formokat. Egyes böngészők a vírusok hasonló viselkedése miatt tiltják ezt, ill. a téves adatközlések elkerülésére is fontos.
- A stílus-kialakításnál ügyelni kell az átméretezhetőségre (nagyító-gombok) ill. tekintettel kell lenni a színvakokra és a kevéssé kontrasztos (pl. fekete-fehér) megjelenítő-eszközökre.

Good coding practices

- Nevezéktan:
    - A JS esetfüggő.
    - Az elnevezések nem kezdődhetnek számmal, nem tartalmazhatnak szóközt és nem egyezhetnek az alapértelmezett objektum-nevekkel (pl. for, while, if).
    - A név bármilyen hosszú lehet, de minél rövidebb, annál könnyebben kezelhető.
    - A nevek lehetőleg legyenek képszerűek, informatívak.
    - A többtagú nevek tagjait aláhúzással vagy az első utáni tagok nagybetűvel kezdésével különíthetjük el (camelCase). Utóbbi előnyösebb, mert a programozásban elterjedt és a IDE-ek felismerik; miáltal egy kattintással kijelölhetők.
    - Ügyeljünk az I, l és 1 betűk/számok összekeverhető voltára!
- Kód-formázás (indentálás); voltaképpen mindegy, hogy tabulátorokat vagy kettőzött szóközöket használunk a beugratásra ill. hogy a kapcsos zárójeleket új sorba írjuk-e, a lényeg, hogy a kód áttekinthető legyen. (Pl. ha az adott programban a TAB túl nagy távolságot jelent, azt lejjebb kell venni vagy szóközöket kell használni helyette.) A JS-esek (közösségileg) a kezdő {-t az eredeti sorban hagyják, míg a PHP-sok új sorba teszik.
  A sorokat célszerű 80 karakternél rövidebben írni!
- Általános rendszabály: egy JS-fájlba tartozó kód átlagos mérete (maximum) 50 sor legyen! Ha ennél több (kb. százas nagyságrendű), akkor a webhely dokumentum-struktúráját átalakítva, új dokumentumot kell nyitni.
- Megfelelő (funkcionalitásra vonatkozó) kommenteléssel a kód emberi olvasását és javítását könnyíthetjük.

Függvények:

- A függvényeket a function kulcsszóval definiáljuk. A neve után zárójelbe kerülhet a tetszőleges számú és nevű paramétere/argumentuma.
- A PHP-val ellentétben a JS-nél nem lehet alapértelmezett értékeket megadni a paramétereknek; ez megkerülhető, ha az függvény elején megvizsgáljuk értéküket egy-egy if-fel, és ha az = = null, akkor hozzájuk rendeljük az ilyenkor szükséges értéket.
- Ha a függvény végrehajtása során elér a return false; rendelkezéshez, akkor a program kilép a függvényből.
- Bármilyen más változó vagy objektum nevét írva a return szó után, a függvény annak értékét szolgáltatja a folyamat végén.
- A függvény-íráskor szem előtt tartandók:
    - A függvények egy-egy feladat(-fajta) sokszori végrehajtására valók.
    - Az egyes feladat-csoportokat külön-külön függvényekbe célszerű elkülöníteni.
    - Korlátlan számú és típusú (string, objektum, változó, array) paraméterrel rendelkezhetnek.
    - A függvény definiálásakor a paramétereknek már létezniük kell (nem lehet őket helyben létrehozni). Meglétüket il. a függvényre vonatkozó alapértelmezésüket if feltételes rendelkezésekkel ill. terner operátorokkal (műveletekkel) biztosíthatjuk.
    - A függvény-nevek megválasztásakor törekedjünk az érthetőségre és egyediségre, nehogy az egyes .js fáljokba írt függvények felülírják egymást.
- Sok JS-függvény kezelésekor érdemes mindegyiket külön (jellemző névvel ellátott) .js fájlba elkülöníteni, és így egy univerzális JS-eszköztárat kiépíteni.

Gyorsírás terner operátorokkal

- A függvény által kezelt objektumok stb. kapcsán túl sok if il. switch rendelkezést kéne írni. Ezt a terner operátorokkal küszöbölhetjük ki.
- A terner operátorok egy if else feltételes rendelkezés-párosnak felelnek meg.
  Mondattanuk: var változó-név = feltétel ? igaz-érték:hamis-érték; pl.:
  var direction = x < 200 ? -1 : 1;
- A terner operátorok egymásba ágyazhatók, pl.:
  var direction = x < 200 ? (x > 100 ? 0 : -1) : 1;

Változók és függvények érvényessége

- A függvényeken kívül definiált (globális) változók az egész script-re vonatkoznak. Használatukat kerüljük, inkább a függvényeken belül definiáljuk a változókat.
- A függvényeken belül létrehozott (lokális) var objektumok (implicit módon definiált változók) csak az adott függvényen belülre érvényesek. (Pl.: var változó-név = ”érték”;.) Azaz az esetleg már meglévő globális változók értékét csak a függvényen belülre nézv változtathatják meg. Használatuk a többi függvény zavartalan működése érdekében előnyös.
- A függvényekben explicitmódon definiált változók (pl.: változó-név = ”érték”;) a teljes további scriptben érvényesek; azaz mind a függvényen belül, mind a további scriptre nézve felülírják azok esetleges korábbi értékeit.

Objektum-gyorsírás (object literal)

- Mint látjuk, az egymást követő scriptek azonos nevű függvényei felülírhatják egymást. Ennek kiküszöbölésére a függvényeket egy-egy objektumhoz rendelt módszerekként hozhatjuk létre.
- A függvények objektumokba szervezése más programozási nyelvek (pl. C++, Java) ún. programozási osztályaira (programming classes) emlékeztet.
- Az objektumok definiálását az objektum-gyorsírással lerövidíthetjük. Az egyes rendelkezéseket itt sorvégi vesszők választják el.
  Mondattana:
  var objektum-név=
  {
  függvény1:function() {kód1;},
  függvény2:function() {kód2;},
  függvény3:function() {kód3;}
  }
  A második függvény meghívása az oldalbetöltési eseménykor (ahogy az adott objektumra vonatkozó módszerek meghívásánál már megszoktuk):
  window.onload=objektum-név.függvény2;
- Az gyorsírásos objektumon belüli változók ill. array-k mondattana:
  változó-név1:’érték1’,
  változó-név2:’érték2’,
  array-név1:[’első_érték’,’második_érték’,’stb’],
  array-név1:[’első_érték’,’második_érték’,’stb’]

Régi és modern JS-módszerek

- Elavult, kerülendő eljárások:
  - A document.write() parancs túlzott használata.
  - Az objektumok létezése/elérhetősége helyett böngésző-verziók ellenőrzése.
  - Sok új HTML tartalom kiíratása a már meglévő kezelése helyett.
  - „Cégszerű” DOM-objektumok használata (pl. document.all a MSIE ill. document.layers a Netscape Navigator esetében).
  - A <head> rész ill. a <script src=”xxx.js” type=”text/javascript” /> taggel beidézett JS-tömbök helyett a dokumentumtestbe írt JS-darabkák.
  - javascript: linkek események helyett.

- Modern, követendő eljárások:
  - Elkülönített .js dokumentumokra hivatkozás.
  - A felhasználandó objektumok ellenőrzése a böngésző verziók helyett.
  - A kliensoldali programok nélkül is jól működő oldalt lássuk el aktív bővítményekkel, és ne a szkriptek megírása után készítsük el a statikus tartalmat tartalék-képpen!
  - Rövid, zárt kódot írjunk, jellemző és egyedi elnevezésekkel, mellőzve a globális változókat.
  - A kód legyen jól áttekinthető, karbantartható („szellős”).
  - A kód tartalmazza az értelmezéséhez szükséges kommenteket.

4. fejezet

HTML-anatómia

- A webes tartalmak nagyrészt HTML ill. XHTML formátumúak (del lehet szó XML-ről és SVG-ről is). Ezt szerveroldali programok (ASP.NET, PHP, ColdFusion, Perl) is generálhatják.
- HTML-alapszerkezet:
  - <html dir="ltr" lang="en"> elem
  - <head> elem
  - <meta http-equiv="Content-Type" content="text/html; charset=utf-8" /> a kódolás értelmezésére, amennyiben azt nem állítottuk be a szerveren.
  - <title> elem
  - <body> elem
- A Document Type Definition (DTD), melyre a DOCTYPE-ben hivatkozunk, megszabja a használható tageket, azok egymásba ágyazhatóságát, lehetséges attribútumait.
- A HTML-elem kezelésének folyamata: a DOCTYPE alapján a böngésző felismeri az adott elemet (és az alapértelmezett megjelenítési/formázási szabályokat társítja hozzá), majd az érvényes attribútumok alapján módosítja a megjelenést; pl. ha az elemre értelmezhető a class attribútum, akkor a böngésző a hozzá tartozható CSS-formázásokra is rákeres, és végrehajtja őket. Hogyha valamely elem vagy attribútum (az adott DOCTYPE-ban) nem értelmezhető, a program nem áll le, hanem kihagyja, vagy pedig felveszi őket a DOM-struktúrába. Az ilye HTML-kód azonban – természetesen – hibás.
- A hibátlan HTML-kódolás előnyei (a hibással szemben):
  - Könnyen javítható.
  - Validálható.
  - Nem okoz váratlan hatásokat.
  - Gond nélkül konvertálható más formárumokba.
- DOM-modell (a HTML-oldala képi kezelési módja):
  - A dokumentumot csomópontok/fülek ágazatásnak tekintjük.
  - Egy leágazás lehet pl.:
    - elem
    - attribútum
    - szöveg
  - A FireBug lenyíló füles mezője a DOM struktúrának felel meg.
  - A sortöréseket (
) egyes böngészők szövegnek, mások elemnek tekintik.
- Elterjedt hiba, hogy a dokumentumban több elem rendelkezik ugyanazon id (unique identifier) attribútummal, ami súlyos működési elégtelenségeket eredményezhet!
  Következtetés: az id attribútumot csak egyetlen elemhez használjuk az adott dokumentumban, a class-t pedig többre.

Régebbi weboldal-dinamizálási módszerek (párbeszédpanelek)

- HTML-szöveg kiíratása a document.write(); módszerrel. Hátránya: a structure és presentation layert keveri és a JS-kód (.js fájlba való) elkülöníthetőségét gátolja.
- A Window objektum is alkalmas visszajelzésre és adat-bevitelre: alert(), confirm() és prompt(). Hátrányuk, hogy a felugrás zavaró és mivel az operációs rendszerekben hibát jeleznek, negatív képzetek társulnak hozzájuk.
- Az alert(változó-név); rendelkezés hasznos javító-eszköz; segítségével mindig követhető, hogy éppen a kód mely szakasza működik (hiszen az OK megnyomásáig a végrehajtás áll). A ciklusoknál azonban óvatosan használjuk, mert lehet, hogy csak hosszas nyomkodás után léphet tovább a program.
  Az alert box értéke az OK és X gomb nyomására egyaránt undefined.
- A függvények true ill. false helyett bármilyen más értéket is felvehetnek (pl. return megfelelo;). A függvény ezen értékét egy változónak adhatjuk: var változó-név=függvény-név();
- A confirm() csak true ill. false értéket vehet fel aszerint, hogy az OK ill. Cancel gombot nyomtuk-e meg. Előnyük, hogy egyszerűen programozhatók és hangjelzéssel járnak.
  A conform box értéke az OK nyomására true, a Cancel és X-re pedig false.

- A prompt() párbeszédpanel értéke az OK megnyomása esetén "" (üres), vagy az előre megadott (preset) ill. egyedieg bevitt érték lehet; a Cancel-re és X-re pedig null.
- A külső .js fájlra csak teljes (kéttagú) <script></script> taggel lehet hivatkozni, a rövidített (<script />) taggel nem. Tehát ez helyes:
  <script type="text/javascript" src="URL"></script>
  ez pedig helytelen:
  <script type="text/javascript" src="URL" />
- Bizonyos ID-jú elem létének ellenőrzése:
  if(!document.getElementById('ID')){return;}
- Bizonyos ID-jú elem értékének beolvasása:
  var változó-név=documet.getElementById('ID').value;
- A felugró ablakok alkalmazásának előnyei:
  - Könnyen értelmezhető és feltűnőbb (hanghatással kiegészített) adatbeviteli lehetőséget jelentenek, mint az (egyszerű) HTML.
  - HTML elemek felhasználása nélkül, igen egyszerűen implementálható.
  - A dokumentum fölött ill. azon kívül jelennek meg, ami szintén kiemeli fontosságukat.
- A felugró ablakok alkalmazásának hátrányai:
  - Az ablakok nem formázhatók és megakasztják a weboldal működését. Emiatt az oldal általános megjelenéséből és működéséből egyaránt kiütnek.
  - A beviteli mező nem a weblap stílusában jelenik meg, ami az oldal egységes dinamizmusérzete ellen hat.
  - Csak a JS bekapcsolása/elérhetősége esetén használhatók.
  - Nem lehet gyorsan és világosan megállapítani, hogy a felugró ablak melyik megnyitott weboldalhoz tartozik. Így egy harmaik fél képes lehet elhalászni a felhasználó által bevitt adatokat. (Ezt phishing-nek nevezik; prompt-fishing, adathalászat.) Oldalunkat az internetes biztonsággal foglalkozó cégek emiatt akár tiltólistára is tehetik.

HTML-dinamizálás a DOM-on keresztül

- A Window mellett a Document objektum kezelésével is dinamizálhatjuk a weboldalakat.
- Eddig a .write() paranccsal írtunk a dokumentumba, ez azonban csak stringeket ad hozzá, a node-ok (elágazások/csomópontok) és attribútumok számát és jellegét nem változtatja. A string hozzáadására a dokumentumban csak a <script> tag helyén kerülhet sor, azaz a script-et nem lehet értelmesen különválasztani a HTML-től.
- A DOM és módszerei lehetővé teszik a módosítandó elemek kijelölését.
- A DOM-elemek elérésének kétféle mondattana:
  - document.getElementsByTagName('P')
    Visszaadja az összes <p> objektum listáját.
    A tag-név{} CSS-kijelölésre emlékeztet.
    A tagName betűit nagy ill. kisbetűvel egyaránt írhatjuk (pl. H2=h2, DIV=div, P=p, stb.), de a nagybetűs változat az alapértelmezett.
  - document.getElementById('id')
    Visszaadja a megadott id attribútummal rendelkező elemek (mint objektumok) listáját.
    Az #id{} CSS-kijelölésre emlékeztet.
- A .osztály-név{} típusú CSS-kijelölésnek nincs JS-analogonja, de ki ügyességgel lehet saját getElementsByClassName() függvényt definiálni az adott osztályú elemek kijelöésére.
- Függvény meghívása az oldal teljes betöltődése után (onload event):
  window.onload=függvény-név;
- Adott elem meghívása előfordulási sorszám szerint (az array-k indexéhez hasonlóan 0-tól kezdődően). Pl. az első bekezdés kijelölése:
  var változó-név=document.getElementsByTagName('p')[0];
- A beágyazott elemek módosításához a kijelölések egymásba ágyazhatók. Pl. a második <p> alatt található első <a> elemet a következőképpen vétethetjük fel:
  var változó-név=document.getElementsByTagName('p')[1].getElemetsByTagName('a')[0];

- Bizonyos elem-típus bizonyos (pl. itt legutolsó) helyszámú elemének felvételéhez az elemekből array-t képezhetünk, mely utóbbi elemeit (pl. a .length property segítségével) egyszerűbben kezelhetjük, pl.:
  var listItems=document.getElementsByTagName('li');
  //az utolsó elem felvétele:
  var lastListItem=listItem[listElements.length-1];
- A .length jellemző és egy (for) ciklus segítségével az összes elemet módosíthatjuk:
  var listItems=document.getElementsByTagName('li');
  for(var i=0; I<listItems.length; i++)
  {
  kód
  }
- Mivel az ID csak egyetlen objektumra vonatkozhat, ennek felvételével is ugyanazt kapjuk:
  var változó-név=document.getEelementsById('id');
- A kétféle módszer ötvözésével az array-szerűen felvett objektumok számát csökenthetjük. Pl. az alábbi definíció csak az eventsList id-jű objektumok alá tartozó <li> elemeket szolgáltatja:
  var events=document.getElementsById('eventsList');
  var eventListItems=events.getElementsByTagName('li');
  for (var i=0; i<eventListItems.length; i++) {kód}

  Ugyanez rövidebben:
  var eventListItems=document.getElementsById('id').getElementsByTagName('li');
  for (var i=0; i<eventListItems.length; i++) {kód}
- Így a két módszerrel a dokumentum minden egyes elemét vagy elemcsoportját elérhetjük.

HTML-elemek felvétele Child, Parent, Sibling, Value alapján

- Ha a befolyásolandó HTML-struktúra nem ismert, általánosabb módszer kell az objektumok elérésére.
- Leírandó az elemek helye a HTML-ben, ill. hogy tartalmaznak-e továbbiakat.
- Az egy szinten lévő elemek (pl. a <head> és <body>) testvérek (siblings).
- Az egymás alatt ill. felett lévő elem (pl. a <h1> és <body>) egymásnak gyermeke (child) ill. szülője (parent).
- Az egyes elemekbe tartozó szöveg,
 stb., habár nem elem, az előbbiekhez hasonló alá- ill. mellérendelt viszonyokkal írható le.
- A dokumentumban levő csomópontok (node-ok) azonosításukra használható sajátságai:
  - nodeType jellemző: megmutatja, hogy az adott node micsoda. Lehet pl. element, attribute, comment, text stb. (12 féle).
    Ha nodeType=1, akkor a node egy element, ha nodeType=3, akkor #text, stb..
  - nodeName jellemző: #text vagy element típusú node típusnevét adja vissza (pl. H2, LI, #text). Mivel a visszaadott érték, mint látjuk, nagy-és kisbetűs is lehet, a további logikai vizsgálatottor célszerű toLowerCase()-szel egységesíteni, pl.:
    if(objektum.nodeName.toLowerCase()= = 'li'){kód}
  - A tagName jellemző csak HTML-elemekre vonatkozik, melyekre értéke megegyezik a nodeName jellemzőével.
    Ezzel szemben pl. az automatikus sortörésekre nézve azonban értéke nem #text, hanem undefined; jelezve, hogy a HTML-nyelvben az adott string nem értelmezhető tagként.
    Hogyha a .getElementsByTagName()[x]; módszer alkalmazásánál nem adjuk meg az [x] helyszámot, akkor a kapott array .tagName jellemzőjének értéke undefined (holott az összes tag ugyanolyan típusú). Hogyha a megadott helyszámú tag nem létezik, a program nem ad értéket, hanem hibával leáll.
  - nodeValue jellemző: a node értékét adja meg; ha a node egy elem, akkor nodeValue=null; ha #text, akkor nodeValue="szöveges tartalom".
- A dokumentum első <p>-ében lévő szöveg megváltoztatása (újradefiniálása):
  document.getElementsByTagName('p')[0].firstChild.nodeValue='Hello World';

Vagyis a 0 sorszámú <p> elembe tartozó első gyermek, azaz #text node értékét vesszük fel, s állítjuk át.

Kijelölés a szülőktől a gyermekek felé

- A szülő-elemek gyermekeit a childNodes jellemzővel érhetjük el:
  - A childNodes felveszi az adott objektum összes elsőrendű al-elemét (gyermekét).
  - Egy adott gyermek-elemet az array-beli sorszámmal (pl. [0]) vagy az item() módszerrel érhetünk el.
  - Az elem-név.firstChild és elem-név.lastChild jellemzők az első és utolsó al-elem felvételének gyorsírásai (melyek egyébként:
    elem-név.childNodes[0] ill.
    elem-név.childNodes[elem-név.childNodes.length-1]).
  - A hasChildNodes() módszerrel ellenőrizhetjük, hogy vannak-e egyáltalán al-elemek egy objektum alatt. Értéke true vagy false.
- A nodeName módszer
- A MSIE-en kívüli böngészőkben az automatikus sortörések is egy-egy #text node-nak minősülnek.

Kijelölés a gyermekektől a szülők felé

- A .parentNode módszerrel történik.
- Példa: egymásba ágyazott <ul id=”ul-id”> és <a id=”a-id”> elemek.
  A szülő elérése:
  var változó-név=document.getElementById(’a-id’);
  alert(változó-név.parentNode.nodeName); = LI
  Ugyanakkor a nagyszülő elérése:
  alert(változó-név.parentNode.parentNode.nodeName); = UL
  A .parentNode sor a végtelenségig folytatható.
- Példa: leellenőrizhetjük, hogy a kérdéses elem egy bizonyos ID-jű elemben található-e. Azaz kijelöljük a kérdéses elemet, majd addig lépkedünk a magasab elemekre, míg el NEM (azaz a while első feltétele előtt: !) érjük a bizonyos ID-jűt, vagy a <body>-t:

```
var kerdesesElem=document.getElementById(’kerdesesId’);
var bennfoglaloElem=kerdesesElem.parentNode;
while(!feljebbLepes != ’bizonyosId’ && bennfoglaloElem != ’document.body’)
{
   bennfoglaloElem=bennfoglaloElem.parentNode;
}
alert(bennfoglaloElem);
```

Kijelölés testvérek között:

- A node previousSibling ill. nextSibling jellemzőjével annak testvéreit érhetjük el.
- Példa: egy lista egyik elemében találhattó az <a id=”a-id”> elem. Ebből az előző és következő listaelemet így érhetjük el:

```
var aTag=document.getElementById(’a-id’);
var apja=aTag.parentNode;
var apjaElőtt=apja.previousSibling;
var apjaUtán=apja.nextSibling;
```

  Ez csak IE-ben működik, hisz a többi böngészőben a listaelemeket #text-nek minősülő automatikus sortörések választják el.
- 104. oldal: a függvények nincsenek felcserélve? – De igen! – Nem, nincsenek!!!
- Példa:<input> elem melletti elem elérése egy paragrafuson belül:

```
var inputTag=document.getElementById(’input-id’);
var spanTag=inputTag.parentNode.getElementsByTagName(’span’)[0];
```

  Azaz a második rendelkezésben az <input> feletti <p> elemben levő első elemet vesszük fel.

- Példa: dátum formátumának ellenőrzése:
  var dateInput=document.getElementById('dateInput-id');
  var formátum=new RegExp("\\d{2}/\\d{2}/\\d{4}");
  var boolean1=formátum.test(dateInput.value);
  Azaz a dateInput field-be beírt kifejezés éertékét egyeztetjük a formátum változóként definiált RegExp objektummal (##/##/####). Ha megegyezik, boolean1=true; egyébként false.
- RegExp objektumok definiálása és használata;
  mint az előző példában láttuk, a RegExp objektumot adatformátumok tárolására és ellenőrzésére használjuk. Az RegExp definiálása után a test() módszerrel ellenőrizhetjük, hogy a paraméterül beírt adat megfelel-e a formátumnak:
  var formatum = new RegExp(???);
  formatum.test(adat);
  Hogyha az adat a formátummal megegyezik, a függvény-érték true (létezik); ellenkező esetben false (!nem létezik).
- Példa: string-változó (értékének) kiíratása span elemen belüli #text objektumként:
  var string1=document.getElementById('string1-id');
  var string-változó="érték";
  string1.firstChild.nodeValue=string-változó;
  Min látjuk, a #text objektum a string1-nek gyermeke!
- Példa: beviteli mezőre fókuszálás:
  var input1=document.getElementById('input1-id');
  input1.focus();
- Példa: <form> adattovábbításának engedélyezése függvény-érték alapján:
  <form action="URL" method="post" onsubmit="return ellenőrzőFüggvény();"></form>
  Ha az ellenőrzőFüggény végértéke true (azaz az utolsó parancs return true;), a form továbbításra kerül (az URL felé), false esetben nem.

HTML-attribútumok kezelése:

- A felvett HTML-elem attribútumait is kezelhetjük, kétféleképpen.
- Régebbi módszer:
  - Az elem-attribútumokat objektum-jellemzőkként kezelhetjük.
  - Első példa: dokumentum első <a> eleme href attribútumának átállítása:
    var firstLink = document.getElementsByTagName('a')[0];
    if (firstLink.href = = 'search.html')
    {
    firstLink.href = 'http://www.google.com';
    }
  - Második példa: kép-attribútumok megadása/felülírása:
    var mainImage = document.getElementById('nav').getElementsByTagName('img')[0];
    mainImage.src = 'forras.jpg';
    mainImage.alt = 'Friss ivóvíz!';
    mainImage.title = 'Forrás';
- Újabb módszer:
  - A getAttribute() ill. setAttribute() módszerekkel felvehetjük ill. beállíthatjuk a paraméterül beírt attribútumot ill. értékét. Ez a módszer bonyolultabb, de formálisan konzekvensebb és a magasabbrendű programozási nyelvekkel kompatibilisabb.
    Az objektum.getAtrribute('attribútum-név') parancs felveszi az adott objektum attribútum-értékét. Ha az attribútum-érték üres (""), akkor a felvett érték is ugyanaz. Ha az attribútum nem létezik, akkor az eredmény null.
    Az objektum.getAtrribute('attribútum-név','attribútum-érték') paranccsal beállíthatjuk a kérdéses attribútum új értékét. Ha az attribútum nem létezik, az eredmény undefined.
  - Példa: link href attribútumának beolvasása (feltételként) és módosítása:
    var firstLink = document.getElementsByTagName('a')[0];
    if (firstLink.getAttribute('href') = = 'search.html')

```
{
   firstink.setAttribute( 'href' , 'http://www.google.com');
}
```

  - Mindezek tekintetében a könyv 108. oldala teljességgel hibásnak bizonyult.
- Bármely attribútum megadható/módosítható (egy-kettő biztonsági okokból nem).
- Egyedi attribútumokat is megadhatunk.
- Az értékeket attribútumokban tárolva sok tesztelést és ciklusokat spórolhatunk meg.
- A HTML-attribútumok és JS-parancsok nevei ütközhetnek, ilyenkor speciális kulcsszavakat használunk. (Egyébként a program hibával leáll.)
  Pl. a <label for=””> attribútumot .for helyett .htmlFor –nak jelöljük (a for ciklussal ütközne).
  Ugyanígy a <h1 class=””> .class helyett .className!

Elemek létrehozása, felülírása, törlése

- Eljárások elemek létrehozására, módosítárása, törlésére:
  - document.createElement('elem-név');
    Egy új, attribútumként megadott típusú elem-node –ot hoz létre.
  - document.createTextNode('string');
    Egy új, attribútumként megadott szöveges tartalommal bíró #text-node –ot hoz létre.
  - node1.appendChild(node2);
    A node1 alá, a meglévők után egy új gyermeket hoz létre (node2).
  - node2 = node1.cloneNode(boolean);
    Létrehozza a node2-t, a node1 másolataként. Hogyha a módszer paramétereként megadott boolean = true, akkor a node2 az eredeti összes gyermekét és attribútumát is tartalmazza, ellenkező esetben nem.
  - node1.insertBefore(node3,node2);
    a node1 alá beszúrja az új node3 gyermeket, a szintén ott lévő node2 elé!
  - node1.removeChild(node2);
    A node1 alól eltávolítja a node2 gyermeket.
  - node1.replaceChild(node3,node2);
    A node1 gyermekeként szereplő node2-t lecseréli a node3-ra!
- Mint látjuk, a createElement() és createTextNode() a document objektumra vonatkozó módszerek, míg a többi bármely node-ra nézve használható.
- Fontos, hogy a node-ok kezelésére olyan HTML-elemeket használjunk, melyek a JS kikapcsolása esetén sem zavarnak senkit, és az oldal használhatóságából sem von le semmit, ha nem működnek.
- A Submit gombok nem formázhatók, a linkek viszont igen, de csak JS-tel lehet őket továbbításra használni. Egy nagyon egyszerű megoldás erre:
  Submit
  Ha azonban a JS ki van kapcsolva, a formot ezzel nem lehet továbbítani!
- Biztosítanunk kell tehát egy egyszerű Submit gombot is, a JS-et nem támogató felhasználóknak:
  1. Vizsgáljuk meg az összes <input> elemet!
  2. Ellenőrizzük, hogy type=”submit” –e!
  3. Ha nem, folytassuk a következő elemmel!
  4. Ha igen, hozzunk létre egy <a> elemet, benne egy #text node-dal!
  5. A #text node értékéül vegyük fel az <input> elem értékét (value)!
  6. A link href-attribútumát is állítsuk be a következőre:
     javascript:document.forms[0].submit()
     [Korrektebb, ha a következő fejezetben megismerendő eseménykezeléssel adjuk az <a>-hoz a submit funkciót!]
  7. Cseréljük le az <input> elemet az <a>-ra!
- Az ehhez szükséges kód:

```
function submitToLinks()
{
  if ( !document.getElementById || document.createTextNode ) {return;}
```

```
    var inputs, i, newLink, newText;
   inputs = document.getElementsByTagName( 'input' );
    for ( i=0 ; i<inputs.length() ; i++ )
    {
       if ( inputs[i].getAttribute( 'type' ).toLowerCase() != 'submit' )
       {
          continue;
          i++
       }
       newLink = document.createElement( 'a' );
       newText = document.createTextNode( inputs[i].getAttribute( 'value' ) );
       newLink.appendChild( newText );
       newLink.setAttribute( 'href' , 'javascript:document.forms[0].submit()' );
       inputs[i].parentNode.replaceChild( newLink , inputs[i] );
       if( i<inputs.length )
      {
          i--
       }
   }
}
window.onload = submitToLinks;
```

- A példa tanulságai:
  - Egyszerre több változó-nevet (var objektumot) is definiálhatunk:
    var elso,masodik,harmadik,stb;
  - Hogyha a ciklusban a return; ill. return érték; parancs szerepel, akkor az leáll (és felveszi az adott értéket); hogyha viszont continue; az utasítás, akkor csak az adott folyamat végrehajtása áll le, maga a ciklus továbblép.
  - Az i++ kifejezés a continue; statement után teljesen felesleges, mert azt a kód-részletet (éppen mivel a continue; után van) a program sohasem hajthatja végre!!!
  - Hogyha a HTML-ben több submit gomb volt, a program hibával leállna, mert az utolsó cseréje miatt az i értéke eggyel nagyobb lesz, mint az új array hossza, tehát a következő input elem helyszáma. Ezért a csere után az i értékét eggyel csökkenteni kell. Erre szolgál az if(i<inputs.length) {i--} kifejezés.
  - A gombokkal ellentétben a linkek nem működnek ENTER-nyomásra. Ezt a problémát kiküszöbölhetjük, ha a Submit gombot eltávolítás helyett csupán elrejtjük, ill. egy üresen hagyott kép-gombot adunk a formhoz (lásd a következő fejezetet).
  - Másfelől a gombok kinézetének megváltozása megzavarhatja a felhasználót, akár biztonsági kockázatot is sejthet a rendkívüli megjelenés miatt. Mivel azonban ma már az ilyen gombok a gyakoribbak, ez a hátulütő elhanyagolható.

A <noscript> helyettesítése

- A HTML-be ágyazott JS-kódok működési elégtelenségeinek kiküszöbölésére a <script> tagek mellett <noscript> tageket is beírhatunk. Az ebben lévő (HTML) tartalom akkor jelenik meg, hogyha a JS nem elérhető.
- Minthogy azonban a HTML-oldalba nem illendő scriptet írni, a <noscript> tag a JS kiesésének korrigálására alkalmatlannak tartandó (annál is inkább, mert a HTML-tartalom megjelenését amúgy sem szabad a behavior layerhez kötni). Így a <noscipt>-be rejthető HTML-hibafeliratok és a tiltott <noscript> tag használata helyett más (DOM) megoldásokat kell alkalmaznunk.
- A <noscript> tag alkalmazása:
  <script type="text/javascript"></script>
  <noscript>A JS nem működik!</noscript>
  A közleménybe ilyenkor érdemes webmesteri elérhetőségeket is írni.
- A modern megoldás: megadjuk a HTML-tartalmat (pl. hibaüzenetet egy <p> tagben), és a JS-tel helyettesítjük vagy töröljük azt (ami tehát csak akkor történik meg, ha a JS lefut).

- Példa: bekezdésbe írt hibaüzenet kezelése:
  A hibaüzenet:
  Az oldal megtekintéséhez a JavaScript engedélyezése szükséges. Ha nem tudja vagy kívánja engedélyezni a JavaScript futtatását, kérjük, keressen fel minket ezen a címen, és megpróbálunk segítseni.
  A vonatkozó script:

```
function noscript()
{
//A felhasználandó <p> elem ill. módszerek létezésének elllenőrzése.
   if( !document.getElementById || !document.createTextNode ){return;}
   var noJSmsg = document.getElementById( ’noscripting’ );
   if( !noJSmsg ){return;}

//A bekezdés elé „Browser test succeeded” tartalmú <h1> beszúrása a hibaüzenet elé.
   var head = document.createElement( ’h1’ );
   var headline = ’Browser test succeeded’;
   head.appendChild( document.createTextNode( headline ) );
   noJSmsg.parentNode.insertBefore( head , noJSmsg );

//A sikeres elenőrzésről szóló <p> beszúrása a hibaüzenet elé (és a <h1> mögé).
   replaceMessage = ’We tested if Your browser is capable of ’;
   replaceMessage += ’supporting the application, and all checked out fine. ’ );
   replaceMessage += ’Please proceed by activating the following link.’ );
   var infoPara = document.createElement( ’p’ );
   infoPara.appendChild( document.createTextNode( replaceMessage );
   noJSmsg.parentNode.insertBefore( infoPara , noJSmasg );

//A hibaüzenet lecserélése a továbblépő – linkre.
   var linkPara = document.createElement( ’p’ );

   var appLink = document.createElement( ’a’ );
   applink.setAtrribute( ’href’ , ’application.aspx’ );
   var linkMessage = ’Proceed to application.’;
   appLink.appendChild( document.createTextNode( linkMessage ) );
   linkPara.appendChild( appLink );
   noJSmsg.parentNode.replaceChild( linkPara , noJSmsg );
}

window.onload = noscript;
```

- A példa tanulsága: nehéz sok szöveget generálni a DOM-on keresztül (a document.write paranccsal pedig nem illendő). Az ehhez hasonlón esetekben, amikor felesleges a kiírandó szöveget változóként definiálni (mert a kiíratáson kívül nem végzünk rajta műveleteket), leginkább az innerHTML módszert használjuk.

A JS rövidítése innerHTML segítségével

- Az innerHTML (nem-standard, de általánosan elterjedt) jellemző segítségével komplett HTML-részleteket szúrhatunk be a JS-tel. A node-okat és gyermekeiket a felhasználó-oldali JS generálja a HTML-tagek értelmezésével.
  Ennek módja: a beszúrandó HTML-kódot stringként (vagyis idézőjelben) egy változóhoz rendeljük, majd a megváltoztatandó/beillesztendő elemhez hozzárendeljük a .innerHTML módszerrel:

```
megvaltoztatandoElem = getElementsByTagName( ’p’ )[0];
beszurandoHTML = ’<p>Egy<br /><hr />bekezdes.</p>’;
megvaltoztatandoElem.innerHTML = beszurandoHTML;
```

- A 116. oldali példában a replaceMessage változó definíciójának második sorában egyenlőség (=) helyett inkrementum (+=) értendő!
- Az innerHTML-lel tehát kiolvashatjuk az egyes elemekben található HTML-tartalmat, és komplett HTML-darabokat cserélhetünk le a weboldalban, ami az Ajax alkalmazásoknál igen gyakori.

DOM-összegzés:

- **Elemek felvétele a dokumentumból:**
  - document.getElementById('id')
    Az adott id-jű elemet objektumként felveszi.
  - document.getElementsByTagName('tagname')
    Array-szerűen felveszi a megadott HTML-elem összes példányát.
- **Elem-attribútumok, node-értékek és egyéb adatok felvétele**
  - node.getAttribute('attribútum')
    A kijelölt objektum megadott attribútumát veszi fel. (Annak megváltoztatására nem alkalmas!)
  - node.setAttribute('attribútum' , 'érték')
    A kijelölt objektum megadott attribútumának értékét állítja be. (Annak felvételére nem alkalmas!)
  - node.nodeType
    A node típusának számát adja vissza. Értéke HTML-elemre 1, #text node-ra 3.
  - node.nodeName
    A node típusának (nagybetűkkel írt) nevét adja vissza. Értéke HTML-elemre pl. TABLE, #text node-ra pedig #text (régebben #textNode).
  - node.nodeValue
    Beolvassa ill. megváltoztatja a node értékét (ill. a #text szövegét).
- **Node-ok relatív felvétele**
  - node.previousSibling
    A megadott node megelőző testvérét veszi fel objektumként.
  - node.nextSibling
    A megadott node-ot következő testvérét veszi fel objektumként.
  - node.childNodes
    A megadott objektum (node) összes gyermekét array-ként felveszi.
    - Az első gyermek-node egyszerű felvételére a
      node.firstChild
      parancs szolgál.
    - Az utolsó gyermek-node egyszerű felvételére a
      node.lastChild
      parancs szolgál.
  - node.parentNode
    A megadott node-ot tartalmazó node-ot veszi fel.
- **Új node-ok definiálása**
  - document.createElement('elem-név')
    Ezzel egy új elem-node-ot definiálhatunk. Az elem típusának nevét (esetfüggetlenül) stringént adjuk meg.
  - document.createTextNode('string')
    Egy #text node-otdefiniál, a string-nek megfelelő szöveges értékkel.
  - node2 = node1.cloneNode('true/false');
    A node1 másolatát veszi fel node2-ként. Ha a függvény-paraméter true, akkor az új node az eredeti összes gyermekét (és attribútumait) is tartalmazza (ellenkező esetben nem).
  - node1.appendChild(node2);
    A node2-t a node1 utolsó gyermekévé teszi.
  - node1.insertBefore(node3 , node2);
    A node3-at a node1 node2-t megelőző gyermekévé teszi.

- node1.removeChild(node2);
  A node1 alatti node2 gyermeket eltávolítja.
- node1.replaceChild(node3 , node2);
  A node1 alatti node2 gyermeket a node3-ra cseréli.
- elem.innerHTML
  Az elembe zárt HTML-tartalmat stringként kiolvassa ill. megváltoztatja [beleértve az összes al-elemet, attribútumaikat és szöveges (#text) tartalmukat].

DOM – általános megoldások

- Az egyes böngészők az automatikus sortöréseket #text node-ként értelmezik, míg mások a HTML tagek részének tekintik. Ezért a .nextSibling felvétel (módszer) használatakor mindig ellenőrizni kellene a nodeType-ot (egy feltételes rendelkezéssel), a #text node-ok kikerülésére.
- Néhány egyszerű eljárással kiküszöbölhetjük ezt a gondot, amivel a kód többi részét nagymértékben egyszerűsítjük. Ezt JavaScript Framework-nek ill. Library-nek nevezzük.
- Saját framework-ünk a DOMhelp objektumból ill. annak módszereiből áll. A DOMhelp-re vonatkozó módszerek olyan függvények, melyek meghívásával a fentihez hasonló, a JS nyelvtanából ill. a böngészők eltérő interpretálásából adódó hibákat/eltéréseket egyszerűen kiküszöbölhetjük. Azaz a DOMhelp objektumra olyan összetett JS-módszerek vonatkoznak, melyek a böngészők egyenetlenségeire nézve (is) kiegészítik a JS alapértelmezéseit.
  A DOMhelp objektum definiálása:

```
DOMhelp =
{
   lastSibling:function(node){…},
   firstSibling:function(node){…},
   getText:function(node){…},
   setText:function(node,szöveg){…},
   closestSibling:function(node,irány){…},
   createLink:function(URL,szöveg){…},
   createTextElm:function(elem,szöveg){…},
   initDebug:function(){},
   setDebug:function(bug){},
   stopDebug:function(){}
}
```

- **A felmerülhető problémák és a hozzájuk tartozó függvények:**
  - **Node utolsó testvérének megkeresése.**

```
lastSibling:function(node)
{
   varTempObj = node.parentNode.lastChild;
   while( tempObj.nodeType != 1 && tempObj.previousSibling != null )
   {
      tempObj = tempObj.previousSibling;
   }
   return ( tempObj.nodeType = = 1 ) ? tempObj : false;
},
```

  - **Node első testvérének megkeresése.**

```
firstSibling:function(node)
{
   varTempObj = node.parentNode.firstChild;
   while( tempObj.nodeType != 1 && tempObj.nextSibling != null )
   {
      tempObj = tempObj.nextSibling;
   }
   return ( tempObj.nodeType = = 1 ) ? tempObj : false;
},
```

- **A node gyermekei közül az első #text típusúba írt szöveg felvétele.**
  A reg.test(tempObj.nodeValue kifejezéssel ellenőrizzük, hogy a tempObj objektum megfelel-e a reg nevű RegExp-ben megadott követelménynek. Vagyis a kifejezés értéke akkor true, hogyha a node értéke nem csak szóközökből áll.

```
getText:function(node)
{
   if( !node.hasChildNodes() ) {return false;}
   var reg = /^\s+$/;
   var tempObj = node.firstChild;
   while( tempObj.nodeType != 3 && tempObj.nextSibling != null ||
   reg.test( tempObj.nodeValue ) )
   {
      tempObj = tempObj.nextSibling;
   }
   return tempObj.nodeType = = 3?tempObj.nodeValue:false;
},
```

- **A node gyermekei közül az első #text típusúba írt szöveg beállítása.**

```
setText:function( node , text )
{
   if( !node.hasChildNodes() ){return false;}
   var reg = /^\s+$;
   var tempObj = node.firstChild;
   while( tempObj.nodeType != 3 && tempObj.nextSibling != null ||
   reg.test( tempObj.nodeValue ) )
   {
      tempObj = tempObj.nextSibling;
   }
   if( tempObj.nodeType = = 3 ){tempObj.nodeValue = txt;}
   else {return false;}
},
```

- **A node (adott irányban) legközelebbi, nem #text típusú testvérének felvétele. [Az irányt a direction paraméter jelzi; ha =1, akkor a következő, ha =-1, akkor a megelőző testvért szolgáltatja a függvény.]**

```
closestSibling:function( node , direction )
{
   var tempObj;
   if( direction = = -1 && node.previousSibling != null )
   {
      tempObj = node.previousSibling;
      while( tempObj.nodeType != 1 && tempObj.previousSibling != null)
      {
         tempObj = tempObj.previousSibling;
      }
   }
   else if( direction = = 1 && node.nextSibling != null )
   {
      tempObj = node.nextSibling;
      while( tempObj.nodeType != 1 && tempObj.nextSibling != null)
      {
         tempObj = tempObj.nextSibling;
      }
   }
   return tempObj.nodeType = = 1?tempObj:false;
},
```

A do while ciklus alkalmazásával egyszerűsíthetjük a kódot:

```
closestSibling:function( node , direction )
{
   if( direction = = -1 && node.previousSibling != null )
   {
      do while( node.nodeType != 1 && node.previousSibling != null)
      {
         node = node.previousSibling;
      }
   }
   else if( direction = = 1 && node.nextSibling != null )
   {
      do while( node.nodeType != 1 &&  node.nextSibling != null)
      {
         node = node.nextSibling;
      }
   }
   return node.nodeType = = 1?node:false;
},
```

- **Megadott helyre mutató szöveges link készítése.**

```
createLink:function( URL , txt )
{
   var tempObj = document.createElement( 'a' );
   tempObj.appendChild( document.createTextNode( txt ) );
   tempObj.setAttribute( 'href' , URL );
   return tempObj;
},
```

- **Megadott szöveget tartalmazó HTML-elem készítése.**

```
createTextElm:function( elm , txt )
{
   var tempObj = document.createElement( elm );
   tempObj.appendChild( document.createTextNode( txt );
   return tempObj;
}
```

- **Programozható JS debug konzol készítése**

Az alert ablakok sokszori nyomkodása helyett egy új -be való kiíratással is követhetjük a kód futását. E -et id-vel látjuk el, így megfelelő CSS-beállítások révén fixen az oldal többi része fölé pozícionálhatjuk.
A konzol három részből áll:

  - Az első ellenőrzi, hogy nem fut-e már egy hasonló konzol (ha igen, akkor a DOMhelp objektum stopDebug módszerével leállítja azt), majd létrehozza, id-vel jelöli az új -et és beilleszti azt a weboldal legelejére:

```
initDebug:function()
{
   if( DOMhelp.debug ){DOMhelp.stopDebug();}
   DOMhelp.debug = document.createElement( 'div' );
   DOMhelp.debug.setAttribute( 'id' , DOMhelp.debugWindowId );
   document.body.insertBefore( DOMhelp.debug , document.body.firstChild );
},
```

  - A második ellenőrzi, hogy a konzol-div létezik-e (ha nem, meghívja az előző módszert), majd a paramétereként megadott bug stringet innerHTML-ként hozzáadja a DOMhelp.debug elemhez:

```
setDebug:function(bug)
{
```

```
    if( !DOMhelp.debug ){DOMhelp.initDebug();}
    DOMhelp.debug.innerHTML += bug + '\n';
},
```

- A harmadik eltávolítja a konzol-div-et a dokumentumból (amennyiben az létezik). Mint látjuk, a módszer eltávolítja a DOMhelp.debug változóval azonosított div-et, majd a DOMhelp objektum vonatkozó értékét null-ra állítja (ellenkező esetben a debug-folyamat újbóli meghívásakor a program megpróbálná a már eltávolított div-be kiírni az adatokat, mivel annak logikai helye (az vonatkozó objektum-érték true állapota) megmaradt. Azaz nemcsak az objektum-érték tartalmát kell törölnünk, hanem annak logikai állapotát is nulláznunk kell:

```
stopDebug:function()
{
    if( DOMhelp.debug )
    {
        DOMhelp.debug.parentNode.removeChild( DOMhelp.debug );
        DOMhelp.debug = null;
    }
}
```

5. fejezet

A Presentation Layer módosítása JS-tel

- Minden HTML-elemnek van style attribútuma, melyet a JS-tel megváltoztathatunk.
- Hogyha a display style-jellemzőt ''-nek (üresnek) vesszük, az megfelel az eredeti beállításhoz való visszatérésnek. Ez azért fontos, mert a JS-esek nem mindig tudják, hogy milyen display mellett kellene megjelenne az adott elemne, és automatikusan block-ot írnak be, ha a display: none; formázást akarják érvényteleníteni.
  Emiatt jobb, ha display helyett lehetőség szerint (ha az elemek által elfoglalt hely-maradvány nem zavaró) visibility-t használunk.
- Elem style-attribútumának beállítása (display:none;color:#ffcc00; formázás):
  - Klasszikus attribútum-beállítás:
    elem.setAttribute('style' , 'display:none;color:#ffcc00;');
    FIGYELEM! Az IE nem támogatja, hogy a .setAttribute() módszernél az attribútum 'style' vagy 'class' legyen; ezért a .style ill. .className JS-jellemzőket célszerű használni!
  - Az objektum style JS-jellemzőjének beállítása:
    elem.style.display = 'none';
    elem.style.color = '#ffcc00';
- **A beállítható JS-style-jellemzők listája**
  A jellemző-nevekben a kötőjel helyett camelCase áll; valamint a float helyett cssFloat-ot írunk!
  - background, backgroundAttachment, backgroundColor, backgroundImage, backgroundPosition, and backgroundRepeat
  - border, borderBottom, borderTop, borderLeft, borderRight, and for each of them style, width, and color
  - color, direction, display, visibility, letterSpacing, lineHeight, textAlign, textDecoration, textIndent, textTransform, wordSpacing, letterSpacing
  - margin, padding (each for top, left, bottom, and right)
  - width, height, minWidth, maxWidth, minHeight, maxHeight
  - captionSide, emptyCells, tableLayout, verticalAlign
  - top, bottom, left, right, zIndex, cssFloat, position, overflow, clip, clear
  - listStyle, listStyleImage, listStylePosition, listStyleType
  - font, fontFamily, fontSize, fontStretch, fontStyle, fontVariant, fontWeight
- A klasszikus attribútum-beállítási eljárás előnye, hogy rövidebb kódot eredményez; hátránya, hogy a JS (behavior) és CSS (presentation) réteg keveredését jelenti.
  Emiatt a következő esetekben a JS-stílus-jellemzőt célszerű használnunk:
  - böngészők CSS-támogatási hibáinak kiküszöbölése
  - elemek dinamikus átméretezése a megjelenítési eltérések megszüntetésére
  - dokumentum-részek animálásához
  - áthúzási (drag-and-drop) műveletek beállítása felhasználói felületeken
- A behavior (JS) és presentation (CSS) layer elkülönítésének egyik módszere: az elem-osztályok (class) dinamikus megváltoztatása.
  - A CSS és JS-fejlesztőnek csupán a class-neveket kell egyeztetniük.
  - Pl. az <elem id="nav"> osztályának megadása/felülírása ("dynamic"-ra):
    var n = document.getElementById('nav');
    n.className = 'dynamic'; vagy n.setAttribute('class' , 'dynamic');
  - **FIGYELEM! Az IE nem támogatja, hogy a .setAttribute() módszernél az attribútum 'style' vagy 'class' legyen; ezért a .style ill. .className JS-jellemzőket célszerű használni!**
  - A class eltávolítható, ha értékét kiürítjük(''):
    elem.className = '';
    A .removeAttribute() módszert is használhatnánk, csakhogy az IE ezt sem támogatja.
  - A HTML-elemeknek több class attribútum-értékük is lehet. A szóközök elhelyezése azonban fontos; pl. class=" első második harmadik" helytelen, mivel az első előtti szóköz felesleges.

- A véletlen szóköz-hozzáadások elkerülésére szolgál a következő kód (lásd DOMhelp), mely ellenőrzi, hogy az attribútum üres-e, és ennek megfelelően szóközzel vagy anélkül adja hozzá az értéket ill. alkalmas az elem-osztály ellenőrzésére is:

```
function cssjs(a,o,c1,c2)
{
  switch(a)
  {
      case ’swap’:  if(!domtab.cssjs(’check',o,c1))
                    {
                        o.className.replace(c2,c1);
                    }
                    else
                    {
                        o.className.replace(c1,c2);
                    }
      break;

      case 'add':   if(!domtab.cssjs('check',o,c1))
                    {
                        o.className+=o.className?' '+c1:c1;
                    }
      break;

      case 'remove': var rep=o.className.match(' '+c1)?' '+c1:c1;
                    o.className=o.className.replace(rep,'');
      break;

      case 'check':  var found=false;
                    var temparray=o.className.split(' ');
                    for(var i=0;i<temparray.length;i++)
                    {
                        if(temparray[i] = = c1)
                        {
                            found = true;
                        }
                    }
      return found;
      break;
  }
}
```

- A módszer paraméterei és használatuk:
  - a = a class attribútumon elvégzendő akció; lehetséges értékei:
    - ’swap’ = egy adott éték lecserélése egy másikra,
    - ’add’ = új érték hozzáadása,
    - ’remove’ = érték eltávolítása,
    - ’check’ = egy érték meglétének ellenőrzése.
  - o = az objektum, melynek class attribútumát kezeljük.
  - c1 = eredeti class-érték
  - c2 = új class-érték (csak a swap akciónál szükéges)
- A kezelendő class-étéket egyetlen paraméterként adhatjuk meg, így a későbbiekben egyszerűen lecserélhetjük másra.
- Ha egy webhelyen sok class-t kell kezelnünk (hozzáadnunk ill. elvennünk), célszerű az erre szolgáló módszert egy külön JS-fájlba elkülöníteni.
- A DOMhelp.getText(node) módszerrel felvehetjük egy node szöveges tartalmát.

- A DOMhelp.cssjs('check', …..) értéke létezik/IGAZ, ha a megadott class-érték már megtalálható, és HAMIS/nem létezik, hogyha nem.

- **Segítség a CSS-designhoz**
  - Az elemek felvétele JS-tel a DOM révén sokkal egyszerűbb, mint a CSS-kijelölések. Pl. míg CSS-sel csak az adott elemek gyermekei irányába értelmezhetők érintési (:hover) hatások, addig a JS-tel a szülők felé is (a .parentNode-dal).
  - A JS-tel id és class attribútumokat kezelhetünk, tartalmat generálhatunk, style és link elemeket kezelhetünk, vagyis komplett CSS-eket adhatunk az oldalhoz ill. távolíthatunk el.
  - Legegyszerűbb, hogyha a <body> elemhez vagy a dinamizált elemhez adunk valamilyen class-t, majd ennek a segítségével a benne lévő elemekhez már specifikus CSS-kijelöléseket írhatunk, arra az esetre, ha a JS működik. Az eredeti (fő-osztállyal nem specializált) kijelölések pedig a JS futása nélküli állapotra vonatkoznak.
  - Ha sok elemhez sok efféle práhuzamos formázást kellene írnunk, egyszerűbb, ha a kétfése (ereseti és specifikus) formázást külön CSS-be írjuk, és csak a megfelelőt töltetjük be a felhasználóval, egy JS-tel generált link taggel. Így a dinamizált és eredeti (statikus) oldal formázási utasításai elkülönülnek, és a felhasználónak csak azt kell letöltenie, amelyik szükséges.
  - Példa egy CSS-link JS-tes hozzáadására:
    ```
    var newStyle = document.createElement( 'link' );
    newStyle.setAttribute( 'type' , 'text/css' );
    newStyle.setAttribute( 'rel' , 'StyleSheet' );
    newStyle.setAttribute( 'href' , 'URL' );
    document.getElementsByTagName( 'head' )[0].appendChild( newStyle );
    ```
  - A böngészők többféle alapértelmezett megjelenítési stílussal rendelkezhetnek (lásd pl. Firefox/Nézet/Oldalstílus). Másképp is elérhetjük ugyanezt, ti. számos CSS-t belinkelhetünk, majd a felesleges <link> elemekre egy ciklussal beállítjuk a disabled attribútumot:

```
switcher={
        menuID:'styleswitcher',
        chooseLabel:'Choose Style:',
        init:function(){
                var tempLI,tempA,styleTitle;
                var stylemenu=document.createElement('ul');
                tempLI=document.createElement('li');
                tempLI.appendChild(document.createTextNode(switcher.chooseLabel));
                stylemenu.appendChild(tempLI);
                stylemenu.id=switcher.menuID;
                var links=document.getElementsByTagName('link');
                for(var i=0;i<links.length;i++){
                        if(links[i].getAttribute('rel')!='stylesheet' &&
                        links[i].getAttribute('rel')!='alternate stylesheet'){
                                continue;
                        }
                        tempLI=document.createElement('li');
                        tempA=document.createElement('a');
                        styleTitle=links[i].getAttribute('title');
                        tempA.appendChild(document.createTextNode(styleTitle));
                        tempA.setAttribute('href','#');
                        tempA.onclick=function(){
                                switcher.setSwitch(this);
                        }
                        tempLI.appendChild(tempA);
                        stylemenu.appendChild(tempLI);
                }
```

```
            document.body.appendChild(stylemenu);
        },
        setSwitch:function(o){
            var links=document.getElementsByTagName('link');
            for(var i=0;i<links.length;i++){
                var title=o.firstChild.nodeValue;
                if(links[i].getAttribute('title')!=title){
                    links[i].disabled=true;
                } else {
                    links[i].setAttribute('rel','stylesheet');
                    links[i].disabled=false;
                }
            }
            return false;
        }
    }
    DOMhelp.addEvent(window,'load',switcher.init,false);
```

- Objektum definiálása:

```
objektum-név =
{
   jellemző1:'érték', //A jellemző egyes változók elkülönített tárolására való.
   jellemző2:'érték',
   módszer1:function(paraméter){kód;},
   módszer2:function(paraméter){kód;},
}
```

- Paraméter átadása az előbbi objektum 1-es és 2-es módszere között (a this kulcsszó használata):

```
objectName =
{
   method1:function()
   {
      nodeName = document.createElement( 'a' );
      nodeName.onclick = function(){objectName.method2(this);}
   },
   method2:function(parameter2)
   {
      var newVariable = parameter2;
   }
}
```

Itt newVariable = nodeName.

- Link elem disabled attribútumának beállítása:
  linkElements = document.getElementsByTagName('link');
  linkElements[0].disabled = true; vagy linkElements[0].setAttribute('disabled' , 'true');
  Ebben az esetben (disabled=true) az adott linket a böngésző nem veszi figyelembe.
- A Style Switcher eljárások hasznosak, ha az olvashatóságot (kontraszt, betűméret) javítják, azonban ne használjuk őket az oldal megjelenésének felesleges, ízléstelen variálására (pl. átszínezgetés). Hasonló célra sokféle script alkotható.
- A CSS és JS integritásának megőrzését gátolja, ha a projekt kidolgozása során a class-nevek változnak (pl. újabb funkcionalitás hozzáadásakor vagy content management – okokból). Ennek kiküszöbölésére a class-nevek tárolását külön véltozókban végezzük; íhy a JS folyamán végig a változó neve szerepel, melynek értékét bármikor egyszerűen megváltoztathatjuk.
- A class-nevek és egyéb változók elkülönítésére az objektumok jellemzői alkalmasak:
  objektum

{
jellemző:'érték', //Annak leírása, hogy a jellemző mit szabályoz.
}
ezek bárhol másutt meghívhatók az objektum.jellemző kifejezéssel.
Így az objektumok elején, magyarázat mellett felsorolt (pl. class-neveket tároló) változókat a CSS-fejlesztő is könnyen módosíthatja, a JS egyéb részleteinek veszélyeztetése nélkül.

- Ha nagyon sok ilyen CSS-oldalról érdekelt jellemző szükséges egy webhely programozásakor, akkor azokat egy külön .js fájlban (pl. cssClassNames.js), pl. egyetlen CSS nevű objektum jellemzőiként határozhatjuk meg; melyre a projekt dokumentációjában is ki kell térnünk.
- A cssClassNames.js (CSS objektum) kialakítása klasszikus JS-jelölésekkel, és a rájuk való hivatkozás:
  CSS =
  {
  hide:'hide', //A string bármikor megváltoztatható, a script-be a változó-nevet iktatjuk!
  supported:'dynamic'
  }

  var no1 = CSS.hide;
  var no2 = CSS.supported;
- A cssClassNames.js kialakítása JSON-jelölésekkel, valamint hivatkozásuk:
  CSS =
  {
  'hide elements' : 'hide',
  'dynamic scripting enabled' : 'dynamic'
  }

  var no1= CSS['hide elements'];
  var no2= CSS['dynamic scripting enabled'];

- **CSS-támogatási problémák kiküszöbölése**
  - A CSS-sel a bonyolult, statikusan egymásba ágyazott táblázat-designo helyett egyszerű (pl. floating) CSS-designek kerültek előtérbe, valamint az egérérintési hatások révén némi interaktivitást lehetett kölcsönözni a weboldalnak, JS-ismeretek nélkül is. Ez azonban helyenként piszkos trükkökhöz vezetett a CSS működési elégtelenségeinek kipótlására, ainek végeredménye szabálytalan, működőképessége pedig erősen böngésző-függő.
  - A táblázatos pozícionálással ellentétben a CSS floating-nál az egyes oszlopok magassága eltérő.
  - A
    var highest = elem.offsetHeight;
    kifejezés segítségével felvetethetjük a legmagasabb oszlop magasságát, majd az összes többi oszlopét ezzel egyenlővé tehetjük:
    elemek.stye.height = parseInt(highest) + 'px';
    vagy
    elemek.setAttribute('style' , parseInt(highest) + 'px');
    Fontos, hogy az elemek méretezésekor a gyermekektől a szülők felé haladjunk, mert utóbbiak végső mérete az előbbiekétől függ.
  - A :hover támogatatlanságának kiküszöbölése:
    over class és a vonatkozó formázás hozzáadásával/elvételével. Mint látható, az onmouseover és onmouseout eseményeknél a this kulcsszóval adjuk át az éppen felvett elemet:
    newsh =
    {
    overClass : 'over',
    init : function()

```
    {
      var newsList = document,getElementById( 'news' );
      var newsItems = newsList.getElementsByTagName( 'li' );
      for ( var i=0 ; i<newsItems.length ; i++ )
      {
        newsItems[i].onmouseover = function()
        {DOMhelp.cssjs( 'add' , this , newshl.overClass );}
        newsItems[i].onmouseout = function()
        {DOMhelp.cssjs( 'remove' , this , newshl.overClass );}
      }
    }
  }
```

- A CSS dinamikus kijelölései (pl. a :hover, :link, :active, :visited, :focus) csakis a bennük lévő (gyermek) elemekre vonatkoznak, visszaható jellegű nincs köztük. A JS-tel a teljes DOM-ágazat elérhető (pl. .parentNode, .firstChild, .nextSibling).
- Hogyha pl. az előző példában a linkek érintésére az őket tartalmazó bekezdések színét akarjuk megváltoztatni, azt a következő kiegészítéssel érhetjük el:

```
var newsItemLinks = newsList.getElementsByTagName( 'a' );
for ( var i=0 ; i<newsItemLinks.length ; i++ )
{
  newsItemLinks[i].onmouseover = function()
  {DOMhelp.cssjs( 'add' , this.parentNode.parentNode , newshl.overClass );}
  newsItemLinks[i].onmouseout = function()
  {DOMhelp.cssjs( 'remove' , this.parentNode.parentNode , newshl.overClass );}
```

(Klasszikus) eseménykezelés

- Példák eseményekre:
  - A dokumentum kezdeti betöltése és kezelése.
  - Egy kép betöltése.
  - Egy gomb megnyomása.
  - Egy billentyű leütése.
  - Egérérintés.
- Az eseménykezelés legegyszerűbb módja az inline event call, pl.:
  `<a href="moreinfo.html" onclick="return infoWindow(this.href)">more information</a>`
- Elegánsabb, hogyha az elemet egy class vagy id révén magából a script-ből jelöljük ki kezelésre. Ennek legegyszerűbb formája, hogyha az eseménykezelést az objektum jellemzőjeként definiáljuk:
  HTML: `<a href="moreinfo.html" id="info">more information</a>`
  JavaScript:
  ```
  var triggerLink=document.getElementById('info');
  triggerlink.onclick=infoWindow;
  ```
  Mivel a triggerlnk.onclick módszer csak akkor kerülhet meghívásra, ha a triggerlink objektum valóban létezik, így e tényt külön ellenőrizni felesleges!
- Az előbbi eljárás hibái:
  - A node nem átadásra, hanem közvetlen meghívásra kerül.
  - A node-ot az esemény idején csak ez az egy függvény kezelheti.
- Mint látjuk, a node adott eseményeihez közvetlenül rendelt függvények ugyanúgy felülírják egymást, mint a változók értékei az újradefiniáláskor; emiatt közvetlenül nem lehet egyszerre több függvényt rendelni egy objektum-eseményhez. Ezt küszöböli ki az alábbi kód:

```
var triggerLink=document.getElementById('info');
triggerlink.onclick=function()
{
  showInfoWindow(this.href);
  highLight(this);
  setCurrent(this);
```

```
}
function showInfoWindow(url){// Other code}
```
Mint látjuk, a triggerLink onclick eseményéhez (mely egy a#info objektumnak felel meg) egyszerre három függvényt rendeltünk. A this itt a triggerlink objektumot jelképezi, ami az első (showInfoWindow) függvény számára az objektummal (<a>) együtt annak href attribútumát is átadja.

- Mint látjuk, bizonyos eseményekhezfüggvények végrehajtását trásíthatjuk, mely függvényekkel augyanazon vagy más objektumot is kezelhetünk, melyre az eseményt vonatkoztattuk. Tehát pl. a window.onload eseményt, mely a window objektumra vonatkozik, nemcsak ennek a megváltoztatására, hanem pl. linkek elrejtésére is használhatjuk.
  Mint az imént láttuk, több (tetszőleges objektumra vonatkozó) függvényt is hozzá lehet rendelni egy (tetszőleges objektumra vonatkozó) eseményhez; ha azokat egy közös, névtelen függvénybe zárjuk. Továbbra is fennáll azonban a probléma, hogy az eseménykor azután kizárólag ezek a függvények mehetnek végbe. Nemcsak felülírni lehet azonban az eseményhez rendelt módszereket, hanem kiegészíteni is:

```
function addLoadEvent(func)
{
  var oldonload = window.onload;
  if (typeof window.onload != 'function')
  {
    window.onload = func;
  }
  else
  {
    window.onload = function()
    {
      oldonload();
      func();
    }
  }
}
```

Az előbbi függvény paraméterként megkapja az esemény már meglévő funkcionalitását kiegészítendő func függvényt. A régi funkcionalitást eltárolja az oldonload változóba. Ha az nem létezik, a window.onload eseménynek egyszerűen megadja a func-ban leírt funkcionalitást. Ha már létezett, akkor egy névtelen függvényben visszaadja neki az eredeti funkcionalitást [oldonload();], kiegészítve az újjal [func();].

W3C szabványos eseménykezelés

- A W3C DOM-1, DOM-2 és DOM-3 standardjai az előzőeknél határozottabb eseménykezelési struktúrával bírnak:
  - DOM-1:
    - Az event (=esemény) az, ami történik (pl. click).
    - Az event handler (=eseménykezelő) (pl. onclick) az a DOM-1 tároló-objektum, ahová az eseményről létrejöttét regisztráljuk.
  - DOM-2:
    - Az event target (=céltárgy) az az objektum, melyen ez esemény történik (többnyire HTML-elem).
    - Az event listener (=eseményfigyelő) az a függvény, mely az eseményt kezeli.
  - DOM-3:
    - Megjelenik az event capturing fogalma.
- Az eseményeket az addEventListener() módszerrel definiálhatjuk. E függvény első paramétereként megadjuk az esemény nevét (az „on” előtag nélkül), másodikként az eseményre meghívandó függvény nevét (zárójelek nélkül), harmadikként pedig egy logikai (true/false) értéket, ti. hogy be akarjuk-e kapcsolni az event capturing-öt. Ennek egyelőre false értéket

adhatunk.
Pl. az a#info elem click eseményéhez a következőképpen rendelhetjük az infoWindow() függvényt:
var triggerLink = document.getElementById('info');
triggerLink.addEventListener('click' , infoWindow , false);
Két további sorral, a highlight() és unhighlight() függvények beidézésével egy egér-érintési (ill. elhagyási) hatást is definiálhatunk:
triggerLink.addEventListener('mouseover' , highligt , false);
triggerLink.addEventListener('mouseout' , unhighligt , false);

- Az addEventListener() függvény létrehozza az e-vel jelölt event objektumot, ami feleslegessé teszi az onevent típusú jellemzők használatát.
- Az event objektum jellemzői (melyeket az eventListener() függvényben is felhasználhatunk):
  - target az az elem, mely az eseményt kiváltja.
  - type az esemény neve (pl. click)
  - button a lenyomott egérgomb száma (0 = bal ; 1 = középső ; 2 = jobb).
  - keyCode/data/charCode a lenyomott billentyű karakter-kódja. A W3C szabványban mint data, a legtöbb böngészőben charCode, az IE-ben keyCode szerepel. A keyCode jellemző-nevet azoban minden böngésző megérti.
  - shiftKey/ctrlKey/altKey logikai érték, mely true, ha az eseménykor a SHIFT, CTRL ill. ALT gomb nyomva volt.
- Az event objektum (ill. annak jellemzői) révén egyetlen függvénybe foglalhatjuk az adott objektum összes eseményére vonatkozó kód-részleteket:

```
var triggerLink=document.getElementById('info');
triggerLink.addEventListener( 'click', infoWindow, false);
triggerLink.addEventListener( 'mouseout', infoWindow, false);
triggerLink.addEventListener( 'mouseover', infoWindow, false);

function infoWindow(e)
{
   switch(e.type)
   {
      case 'click':
      // Code to deal with the user clicking the link
      break;
      case 'mouseover':
      // Code to deal with the user hovering over the link
      break;
      case 'mouseout':
      // Code to deal with the user leaving the link
      break;
   }
}
```

- A target elem típusának felismerésére is jó az e objektum (mely előbbit a böngésző-közi eltérések kompenzálására pl. a toLowerCase() módszerrel át kell alakítanunk):

```
function infoWindow(e)
{
   targetElement=e.target.nodeName.toLowerCase();
   switch(targetElement)
   {
      case 'input':
      // Code to deal with input elements
      break;
      case 'a':
      // Code to deal with links
```

```
            break;
            case 'h1':
            // Code to deal with the main heading
            break;
        }
    }
```

- Az események definiálásakor és az addEventListener() függvénnyel való megfigyelésükkor két, ún. eseményterjedési problémával szembesülünk:
  - Sok eseményhez tartozik valamilyen alapértelmezett művelet, pl. a formok és linkek click eseménye továbbítást, átirányítást eredményez.
  - Felléphet az event bubbling jelensége, ami abból adódik, hogy az adott elem eseménye az őt magukba záró szülő-elemekre is vonatkoztatható.
- Példa az event bubblingra:

```
<ul id="news">
    <li>
        <h3><a href="news.php?item=1">News Title 1</a></h3>
        <p>Description 1</p>
        <p class="more"><a href="news.php?item=1">more link 1</a></p>
    </li>
    <!-- and so on -->
</ul>
```

Hogyha az itt szereplő linkekhez pl. egy mouseover eseményt rendelünk, akkor az egérérintés az összes bennfoglaló elem mouseover eseményfigyelőjét is elindítja!

- Rendeljünk az előbbi HTML-elemtípusok mindegyikéhez egy-egy click eseményfigyelőt:

```
bubbleTest =
{
    init:function()
    {
        if(!document.getElementById || !document.createTextNode){return;}
        bubbleTest.n=document.getElementById('news');
        if(!bubbleTest.n){return;}
        bubbleTest.addMyListeners('click',bubbleTest.liTest,'li');
        bubbleTest.addMyListeners('click',bubbleTest.aTest,'a');
        bubbleTest.addMyListeners('click',bubbleTest.pTest,'p');
    },
    addMyListeners:function(eventName,functionName,elements)
    {
        var temp=bubbleTest.n.getElementsByTagName(elements);
        for(var i=0;i<temp.length;i++)
        {
            temp[i].addEventListener(eventName,functionName,false);
        }
    },
    liTest:function(e)
    {
        alert('li was clicked');
    },
    pTest:function(e)
    {
        alert('p was clicked');
    },
    aTest:function (e)
    {
        alert('a was clicked');
```

```
    }
}
window.addEventListener('load',bubbleTest.init,false);
```
Mint látjuk, a click eseményre itt alert ablakok nyílnak meg; csakhogy az eseményterjedég miatt pl. a legbelső <a> elemekre kattintva, mind az <a>-ra, mind pedig a <p> és -re vonatkozó alert-ek megjelennek, ebben a sorrendben.

- Ezt kiküszöbölhetjük, hogyha az egyes esemény-függvényekbe az e.stopPropagation(); parancsot beírjuk, pl.:
```
pTest:function(e)
{
   alert(’p was clicked’);
   e.stopPropagation();
},
```
Ekkor csak a DOM-ba legmélyebben beágyazott elemre vonatkozó függvény fut le, a böngésző a bennfoglaló elemeket.már nem tekinti „klikkeltnek”.
- Említettük, hogy egyes elemek eseményei alapértelmezett akciókat eredményeznek. Pl. a linkekre kattintva új dokumentumok nyílnak meg. Adott esetben ezeket ki kell iktatnunk az event listener függvény végrehajtása után.
- A DOM-1 modellben ezt úgy érhető el, hogy az eseményre meghívott függvényt false-ként térítjük vissza:
```
element.onclick = function()
{
   //do other code
   terurn false;
}
```
- A DOM-2 modellben a preventDefault() módszert kell az akciót leíró függvény végén alkalmazni:
```
aTest:function(e)
{
   alert(’a was clicked’);
   e.stopPropagation();
   e.preventDefault();
}
```
Ekkor az egyetlen, a-ra vonatkozó alert megjelenítése után az oldal a korábbiakkal ellentétben már nem próbál meg továbblépni a news.php?item=1 oldalra.
- Példa: HTML DOM-2 event handling CSS-sel:

HTML:
```
<ul id="news">
   <li>
      <h3><a href="news.php?item=1">News Title 1</a></h3>
      <p>Description 1</p>
      <p class="more"><a href="news.php?item=1">more link 1</a></p>
   </li>
   <!-- and so on -->
</ul>
```

CSS:
```
.hide {display:none;}
li.current {background:#ccf;}
li.current h3 {background:#69c;}
```

JS:
```
newshl =
{
```

```
    // CSS classes
    overClass:'over', // Rollover effect
    hideClass:'hide', // Hide things
    currentClass:'current', // Open item

    init:function()
    {
      var ps,i,hl;
      if(!document.getElementById || !document.createTextNode){return;}
      var newsList = document.getElementById('news');
      if(!newsList){return;}
      var newsItems = newsList.getElementsByTagName('li');
      for(i=0;i<newsItems.length;i++)
      {
        hl = newsItems[i].getElementsByTagName('a')[0];
        hl.addEventListener('click',newshl.toggleNews,false);
        hl.addEventListener('mouseover',newshl.hover,false);
        hl.addEventListener('mouseout',newshl.hover,false);
      }
      var ps = newsList.getElementsByTagName('p');
      for(i=0;i<ps.length;i++)
      {
        DOMhelp.cssjs('add',ps[i],newshl.hideClass);
      }
    },

    toggleNews:function(e)
    {
      var section = e.target.parentNode.parentNode;
      var first = section.getElementsByTagName('p')[0];
      var action = DOMhelp.cssjs('check',first,newshl.hideClass)?'remove':'add';
      var sectionAction = action = = 'remove'?'add':'remove';
      var ps = section.getElementsByTagName('p');
      for(var i=0;i<ps.length;i++)
      {
        DOMhelp.cssjs(action,ps[i],newshl.hideClass);
      }
      DOMhelp.cssjs(sectionAction,section,newshl.currentClass);
      e.preventDefault();
      e.stopPropagation();
    },

    hover:function(e)
    {
      var hl = e.target.parentNode.parentNode;
      var action = e.type = = 'mouseout'?'remove':'add';
      DOMhelp.cssjs(action,hl,newshl.overClass);
    }
  }
  window.addEventListener ('load',newshl.init,false);
```

- Az előbbi script működésének vázlata:
  - Az oldal betöltésére definiáljuk a newshl.init akciót (ennek régies, azaz DOM-1 kifejezése: window.onload = newshl.init;

újabb, DOM-2 írásmódja pedig:
window.addEventListener('load' , newshl.init , false);).

- A newshl objektumban először jellemzőkként defifináljuk a felhasználandó CSS-osztályok neveit.
- A meghívásra kerülő newshl.init módszer (függvény) felveszi az <ul> elemet, majd az abban található -ket, majd a bennük lévő első <a> elemeket, és click, mouseover és mouseout eseményfigyelőket definiál rájuk, melyek közül az első a newshl objektum toggleNews, a második kettő pedig a hover módszerét hívja meg. Az <ul> <p> elemeihez pedig hozzáadja a hide class-t.
- Az első linkek click eseményekor meghívott toggleNews módszer először az e.target (azaz <a>) elemektől felemelkedik a nagyszülőkig (<li>), majd az ez alatti első <p>-n ellenőrzi, hogy annak osztálya hide-e? Ha igen, eltávolítja azt, ha nem, hozzáadja azt a <li> alatti összes bekezdéshez..
  Eztután a <li> elemekhez ezzel ellentétes ütemben hozzáadja ill. elveszi a current osztályt.
  Végül az e (=<a>) objektum alapértelmezett módszereit és az eseményterjedést letiltja.
- Végül a hover módszer attól függően, hogy a mouseout vagy mouseon esemény hívta-e meg, a -ekről rendre eltávolítja ill. hozzáadja az over osztályt.

Eseménykezelés nem szabványos böngészőkben

- A nem szabványos (mára már feltehetően kiveszett) böngészők tárgyalásra kerülő eseménykezelési segéd-kódjait a DOMhelp objektum tartalmazza.
- Microsoft IE 6:
  - Az addEventListener() függvény helyett attachEvent()-et kell írni.
  - Eseményfigyelők helyett az összes függvényt közvetlenül az eseményhez kell rendelni.
  - A window a legátfogóbb, alapértelmezett objektum.
  - Példa: általános eseményfigyelő kód a böngésző-közi inkompatibilitások kivédésére (megtalálható a DOMhelp-ben):

```
function addEvent(elm, evType, fn, useCapture)
{
   if (elm.addEventListener)
   {
      elm.addEventListener(evType, fn, useCapture);
      return true;
   }
   else if (elm.attachEvent)
   {
      var r = elm.attachEvent('on' + evType, fn);
      return r;
   }
   else
   {
      elm['on' + evType] = fn;
   }
}
```

  - A példa magyarázata:
    - Megadjuk az elm = target-elem , evType = esemény-név (on nélkül) , fn = függvény-név és useCapture = event capture true/false paramétereket.
    - Hogyha az adott böngészőben az elm.addEventListener módszer létezik, akkor végrehajtásra kerül.
    - Ha nem, akkor az IE 6 írásmódja szerint, az elm.attachEvent módszerrel csatolja a függvényt az eseményhez.
    - Ha ez sem elérhető, akkor a függvényt közvetlenül rendeli hozzá az onevent eseményhez. Ezzel azonban felülírjuk az adott elemre értelmezett összes eseményt!

- Mivel az IE 6 nem ismeri az event objektumot, egy külön függvénnyel kell felismertetnünk az esemény target-elemét. Ezt nehezíti, hogy az általános target, vagyis a window.event objektum jellemzői a W3C event objektumétól eltérők:
  - A W3C target jellemző helyett itt srcElement szerepel.
  - Míg a W3C-ben a button = 0 a bal, = 1 a középső, = 2 pedig a jobb egérgombnak felel meg, addig IE-nél 1 = bal, 2 = jobb és 4 = középső. A bal és jobb gomb egyszeri lenyomásához a 3, mindhárom egyszeri lenyomásához pedig a 7-es érték tartozik.
- A Safari böngésző a linkek helyett azok #text node-jait veszi fel target-ként!
- Az előbbi két probléma-pontot küszöböli ki a következő függvény (megtalálható a DOMhelp-ben):

```
function getTarget(e)
{
   var target;
   if(window.event)
   {
      target = window.event.srcElement;
   }
   else if (e)
   {
      target = e.target;
   }
   else
   {
      target = null ;
   }
   if(!target){return flase;}
   if(target.nodeName.toLowerCase() != 'a'){target = target.parentNode;}
   else{return target;}
}
```

vagy egyszerűbben:

```
getTarget:function(e)
{
   var target = window.event ? window.event.srcElement : e ? e.target : null;
   if (!target){return false;}
   if(target.nodeName.toLowerCase() != 'a'){target = target.parentNode;}
   return target;
}
```

- Az IE hibája továbbá, hogy a stopPropagation() módszer helyett a cancelBubble jellemzőt támogatja (a window.event objektumon). Az ezt érvényesítő DOMhelp függvény:

```
stopBubble:function(e)
{
   if(window.event && window.event.cancelBubble)
   {
      window.event.cancelBubble = true;
   }
   if(e && e.stopPropagation)
   {
      e.stopPropagation();
   }
}
```

- Az IE a stopDefault() függvényt sem támogatja, hanem a returnValue módszert. E hiba kiküszöbölése (lásd DOMhelp; a könyv 168. oldalán helytelen!):

```
stopDefault:function(e)
```

```
{
   if(window.event && window.event.returnValue)
   {
      window.event.returnValue = false;
   }
   if(e && e.preventDefault)
   {
      e.preventDefault();
   }
}
```

- Mivel általában mind az eseményterjedést, mind az alapértelmezést tiltani kell, az előbbi kettőt a DOMhelp cancelClick() módszerében egyesítettük:

```
cancelClick:function(e)
{
   if (window.event && window.event.cancelBubble && window.event.returnValue)
   {
      window.event.cancelBubble = true;
      window.event.returnValue = false;
      return;
   }
   if (e && e.stopPropagation && e.preventDefault)
   {
      e.stopPropagation();
      e.preventDefault();
   }
}
```

- A Safari böngésző, bár „névre” ismeri a preventDefault() módszert, nem tudja végrehajtani. Ezért a DOM-1 szerinti (onevent) írásmódban kell hozzárendelnünk egy függvényt az adott elemhez. Pl. egy <a> elem click eseményéhez tartozó alapértelmezett akció tiltására a megszokott kód:

```
anc = document.getElementById('anchor1');
anc.addEventListener('click',func,false);
func(e)
{
   //vegrehajtando akcio
   e.preventDefault();
}
```

helyett a következőt írjuk:

```
DOMhelp =
{
   safariClickFix:function(){return false;}
}
anc = document.getElementById('anchor1');
anc.addEventListener('click',func,false);
anc.onclick = DOMhelp.safariClickFix;
func(e)
{
   //vegrehajtando akcio
}
```

Mint látjuk, egy return false; rendelkezést tartalmazó függvényt rendelünk az onclick eseményhez. Ezt azért csomagoltuk be egy safariClickFix nevű módszerbe, hogy ha a Safarit később megszabadítják ettől a hibától, akkor könnyen megtaláljuk és kitörölhessük. A második kódot tehát egyszerűsíthetnénk:

```
anc = document.getElementById('anchor1');
```

```
anc.addEventListener('click',func,false);
anc.onclick = function(){return false;}
func(e)
{
   //vegrehajtando akcio
}
```

A kód-karbantartási tanácsok

- Mindig tekintsük át korábbi kódjainkat, frissítsük azt az új szabványok szerint, különítsük el minél jobban a behavior layert a többitől, és szabaduljunk meg a funkcionálisan felesleges vagy ideiglenes részletektől!
- Kerüljük a ciklusok egymásba ágyazását!
- A fő-objektum jellemzői igen alkalmasak a fontos globális változók tárolására.
- Hogyha egy kód-részletet sokszor ismltelgetünk, készítsünk annak elvégzésére egy (ciklikus) módszert (függvényt), melyet a későbbiekben így csak egy helyen kell majd karbantartanunk.
- Hogyha egy adott elemet sokszor kel elérnünk, akkor tároljuk el egy jellemzőben (változóban). A felesleges DOM.elérési kifejezések túlbonyolítják a kódot.
- Sok egymás utáni if és else feltételes rendelkezés helyett terner műveleteket vagy méginkább switch rendelkezéseket használjunk!
- A jövőben kiküszöbölésre kerülhető böngésző-hibák ellen alkotott függvényeket célszerű külön jellemző nevű módszerekként a kódba írni, hogy később is feltűnőek és könnyen eltávolíthatók legyenek.
- Igyekezzünk minél inkább függetleníteni kódunkat a HTML-szerkezettől, mivel ez gyakorta változhat!

Oldal-betöltési hatás

- Hogyha a JS elrejtéseket irányító CSS-eket az @import módszerrel, <style> tagekkel olvastatjuk be, fellép az ún. „flash of unstyled content" (FOUC) jelensége; azaz az elrejtendő részek a CSS beolvasásáig, azaz a betöltődés kezdetén láthatóak lesznek.
- Ezt kiküszöbölhetjük, ha egy script-tel, document.write() parancsokkal belső CSS-oldalt íratunk a weboldalba. Ebben az esetben azonban vigyáznunk kell, hogy a HTML-tagek kacsacsőreit külön idézőjelekbe tegyük, mert a régi böngészők a JS-ben leévő komplett HTML-tageket érvénytelen HTML-nek tekinthetik!
- Példa:

```
<head>
   <script type="text/javascript">
      document.write('<' + 'style type="text/css">');
      document.write('#news li p {display:none;}');
      document.write('<' + '/style>');
   </script>
</head>
```

Karakter-entitások olvasása és szűrése

- A click a leggyakoribb esemény; mely minden elemre értelmezhető és elvileg mind egérrel, mind pedig billentyűzettel kiváltható.
- A szabványos kexdown és keyup (valamint a nem szabványos, és az előbbi kettő után értelmezett keypress) eseménykezelőkkel a felhasználó által bevitt szöveget már a billentyű-leütésekkor ellenőrizhetjük; nem kell a továbbításig várni.
- Példa: A következő kód ellenőrzi, hogy a felhasználó csak számokat ad-e meg a beviteli mezőben. Ellenkező esetben a Submit gomb ot letiltja és hibaüzenetet ír ki:
  HTML:

```
<input type="text" name="Voucher" id="Voucher" />
```

  JS:

```
voucherCheck =
{
   errorMessage:'A voucher can contain only numbers.',
   error:false,
   errorClass:'error',
   init:function()
   {
      if (!document.getElementById || !document.createTextNode) { return; }
      var voucher = document.getElementById('Voucher');
      if(!voucher){return;}
      voucherCheck.v = voucher;
      DOMhelp.addEvent(voucher, 'keyup', voucherCheck.checkKey, false);
   },
   checkKey:function(e)
   {
      if(window.event)
      {
         var key = window.event.keyCode;
      }
      else if(e)
      {
         var key = e.keyCode;
      }
      var v = document.getElementById('Voucher');
      if(voucherCheck.error)
      {
         v.parentNode.removeChild(v.parentNode.lastChild);
         voucherCheck.error = false;
         DOMhelp.closestSibling(v,1).disabled = "";
      }
      if(key<48 || key>57)
      {
         v.value = v.value.substring(0,v.value.length-1);
         voucherCheck.error = document.createElement('span');
         DOMhelp.cssjs('add', voucherCheck.error, voucherCheck.errorClass);
         var message = document.createTextNode(voucherCheck.errorMessage)
         voucherCheck.error.appendChild(msg);
         v.parentNode.appendChild(voucherCheck.error);
         DOMhelp.closestSibling(v,1).disabled = 'disabled';
      }
   }
}
DOMhelp.addEvent(window, 'load', voucherCheck.init, false);
```

- A példa magyarázata:
  - Először jellemzőket definiálunk, majd az oldal betöltésére meghívódik a voucherCheck.init függvény, mely – egyébként abszolút feleslegesen – voucherCheck.v jellemzőként felveszi a 'Voucher' beviteli mezőt. A voucher-en definiáljuk a voucherCheck.checkKey függvényre mutató keyup eseményfigyelőt.
  - A checkKey(e) először kiolvassa a keyup eseményhez tartozó keyCode-ot (var key = e.keyCode;). Ezután ellenőrzi, hogy a voucherCheck.error változó létezik-e. Ha igen, akkor a hibaüzenet látszik, és a submit gomb attribútuma disabled="disabled". A script pedig ezt ellenkezőjére fordítva, eltünteti a hibaüzenetet és disabled="" attribútumot állít be a gombon. Ezután, hogyha a felhasználó nem a megfelelő (szám-)billentyűt nyomta meg, akkor a Voucher értékét visszaállítjuk a billentyű leütése utáni állapotnál eggyel rövidebb

állapotára, majd megjelenítjük a hibaüzenetet és a submit gomb attribútumát disabled=”disabled”-re állítjuk.

- Objektum-jellemző definiálása:
  Hagyományos módon:
  objektumNév = {
  jellemzőNév:”érték”,
  }
  Függvény végrehajtása közben, változóként:
  objektumNév.jellemzőNév = ”érték”;
- Billentyű ASCII-kódjának kiolvasása a keyup eseményből:
  e.keyCode (W3C) ill. window.event.keyCode (IE).
- SHIFT, CTRL ÉS ALT gomb értelmezése a keyup event során:
  e.shiftKey, e.ctrlKey és e.altKey; pl.:
  if(e.altKey && key = = 48){alert(”You pressed ALT + 0 !”);}
- Oldalunkon testreszabott billentyűparancsokat definiálhatunk.

Az eseménykezelés veszélyei

- Az eseménykezelés sok egyedi megoldásra késztethet, de nem szabad túlzásba esni.
- Tekintettel kell lennünk az egérrel nem rendelkező felhasználókra. A nem aktív elemeket linkekkel tabulálhatóvá kell tenni.
- A felhasználók egyenlő hozzáférését biztostíani kell, a beviteli eszközöktől függetlenül.
- A darg-and-drop elemeknek a JS támogatásának hiányában is használhatónak kell lenniük click eseménnyel vagy tabulálással.
- A billentyűkombinációk a keyup W3C esemény révén minden böngészőben használható, azonban elképzelhető, hogy az általunk választott kombináció a felhasználói környezetben már foglalt! Így, különösen, mert a felhasználói környezetek mindegyikében fontos szerepük van a billentyűkombinációknak, célszerű a mieinket opcionálissá ill. egyedileg beállíthatóvá tenni.
- Az accesskey HTML-attribútum értrékéül adott betű (és pl. IE-ben ill. Firefoxban az ALT) lenyomásakor az adott elem aktiválódik. Ez gyakorlatilag megegyezik azzal, mintha az elemen egy eseményt és -figyelőt definiáltunk volna, ami az adott billentyűkombinációra az adott elemre fókuszál, vagy elindítja annak alapértelmezett akcóját. Ez, amennyiben a billentyű szám, biztonságosnak tűnhet; azonban lehetnek különleges karaktereket tartalmazó nevű felhasználók, akik ilyenkor nem tudnák zavartalanul beírni a nevüket.

6. fejezet:

A JS alkalmazásai (képek és ablakok)

- Korábban csak JS-tel lehetett kép-effekteket elérni. Most már az egyszerűbben karbantartható CSS-sel is.
- Dokumentum összes képének elérése:
  Új módszer: var images = document.getElementsByTagName('img');
  Régi módszer var images = document.images;
  Mindkét módszer kép-objektumokból álló array-t ad vissza.
- Dokumentum harmadik képe alt attribútumának kiolvasása:
  Új módszer: var alternative = images[2].getAttribute('alt');
  Régi módszer: var alternative = images[2].alt;
- Az <img> elemre értelmezett attribútumok:
  - border: az elem szegély-jellemzői; ÉRVÉNYTELEN!
    Helyette: style/border-width.
  - name: az img tag neve;
  - complete: értéke a kép betöltődése után true. Csak olvasható (felülírhatatlan),
  - height: a kép magassága pixeben,
  - width: a kép szélessége pixeben,
  - hspace: horizontal space (jobb és bal margó pixelben), ÉRVÉNYTELEN!
    Helyette: style/margin-left és style/margin-right.
  - vspace: vertical space (felső és alsó margó pixelben), ÉRVÉNYTELEN!
    Helyette: style/margin-top és style/margin-bottom.
  - lowsrc: a kép-előnézet URL-je, ÉRVÉNYTELEN!
  - src: a kép URL-je.
- **Képek előzetes betöltése (preload)**
  - Diavetítésekhez, bemutatókhoz előnyös, ha az összes képet előre betöltjük. Mivel ez sok időt vehet igénybe, célszerű opcionálissá tenni.
  - Példa egyetlen kép Image() objektumként való betöltésére:
    ```
    kitten = new ImageI();
    kitten.src = 'pictures/kitten1.jpg';
    ```
  - Példa több kép egyszeri betöltésére:
    ```
    function preload()
    {
       var args = preload.arguments;
       document.imageArray = new Array( args.length );
       for( var i = 0 ; i < args.length ; i++ )
       {
          document.imageArray[i] = new Image();
          document.imageArray[i].src = args[i];
       }
    }
    preload( 'pictures/cat.jpg' , 'pictures/dog.jpg' );
    ```
  - Képek JS-független betöltéséhez 1 x 1 px-es <img>-eket hozunk létre egy CSS-sel elrejtett container-ben. Ez azonban a structure és behavior layer keveredését jelenti!
- Egérérintési hatások:
  - Egy primitív példa; kép elérése a name attribútummal és cseréje:
    HTML:
    ```
    <a href="contact.html" onmouseover="rollover('contact','but_contact_on.gif')"
    onmouseout="rollover('contact','but_contact.gif')"><img src="but_contact.gif"
    name="contact" width="103" height="28" alt="Contact Us" border="0" /></a>
    ```
    JavaScript:
    ```
    function rollover( img, url )
    {
       document.images[img].src=url;
    ```

}
Az új kép betöltése késéssel történik!

- Elem kijelölése objektum-array-ből a name attribútum alapján:
  Új módszer: document.getElementsByTagName('img')[name];
  Régi módszer: document.images[name];
- A késést a képek előzetes betöltésével küszöbölhetjük ki:
  HTML:

```
<a href="contact.html"
onmouseover="rollover('contact',1)"
onmouseout="rollover('contact',0)">
<img src="but_contact.gif" name="contact"
width="103" height="28" alt="Contact Us" border="0" />
</a>
```

  JavaScript:

```
contactoff = new Image();
contactoff.src = 'but_contact.gif';
contacton = new Image();
contacton.src = 'but_contact_on.gif';
function rollover( img, state )
{
   var imgState = state = = 1 ? eval(img + 'on.src') :
   eval(img + 'off.src');
   document.images[img].src = imgState;
}
```

  Az eval() függvény helyett a következő kifejezést is használhatjuk:

```
imgState = state = = 1 ? contacton.src : contactoff.src;
```

  vagy bővebben:

```
if( state = = 1)
{
   var imgState = contacton.src;
}
else
{
   var imgState = contactoff.src;
}
```

  E megoldás hibája, hogy minden egyes képhez sok kódot kell megírni.

Innentől (2011.IV.28.) belehúzok, hogy május közepére kész legyek. Csak olvasom a könyvet, igyekszek készségszinten megérteni minden kódot, és a példákat szerkesztgetem. Csak a legfontosabb tanulságokat jegyzem fel:

- Image objektum definiálása és src attribútumának megadása:
  contactoff = new Image();
  contactoff.src = 'but_contact.gif';
- Event (e) objektum tarrget-elemének felvétele a DOMhelp segítségével:
  var tagretElement = DOMhelp.getTarget(e);
- Elem háttér-pozíciójának beállítása:
  elem.style.backgroundPosition = '0 100px';
- Ha egy függvényen belül a változókat objektumként (a var kulcsszóval) definiáljuk, akkor azok nem fogják felülírni az azonos nevű globális változókat (csak a függvényen belül).
- A CSS kifejlettsége folytán ma már célszerűbb csupán a desing-class-okat és ID-ket hozzáadni a scripttel, és a vizuális hatások létrehozását teljességgel a CSS-designerre hagyni. Tehát

felesleges és nehezen áttekinthető kódot eredményezne, ha a megjelenítést is a scriptben határoznánk meg.

- A window objektum módszerei:
  - open('URL' , 'név', 'jellemzők')
    megnyitja az URL webhelyet egy name nevű ablakban, melyre beállítja a jellemzőket.
  - close()
    becsukja az aktuális ablakot.
  - blur()
    a böngésző-ablak vagy -lap háttérbe küldése.
  - focus()
    a böngésző-ablak vagy -lap előtérbe hozása.
- Példa: új ablak megnyitása
  myWindow = window.open('demo.html', 'my',
  'jellemző1=érték1,jellemző2=érték2,jellemző3=érték3');
  A jellemzőket tároló stringet egy változóba is elkülöníthetjük.
- A window objetum jellemzői: height, width, left, top, location, menubar, resizable, scrollbars, status, toolbar (241. old.)
- A window.open('URL','név','jellemzők') módszer név értékével elnevezhetjük az ablakot (pl. 'elso_ablak'. Így az oldal linkjei a target="elso_ablak" attribútummal elérhetik azt, és a linkelt tartalom a megnevezett ablakban fog megjelenni.
- Példa:

  Open grid (opens in a new window)

  Ez a link, ha !popup(), akkor false értékkel tér vissza! Feltétlenül közöljük a felhasználóval, hogy egy link új ablakra vagy lapra lép, az esetleges félreértések elkerülésére!
- A megnyitott ablakhoz tartozó módszerek mondattana:
  - Az ablak megnyitása:
    változóNév = window.open('URL','név','jellemzők');
  - Ablak elrejtése a többi mögé:
    változóNév.blur();
  - Ablak kiemelése a többi elé:
    változóNév.focus();
  - Ablak bezárása:
    változóNév.close();
  - Ablak elmozdítása jobbra x és lefelé y pixellel:
    változóNév.moveBy(x,y);
  - Ablak bal felső csücskének beállítása az (x,y) pixelnek megfelelő helyzetbe:
    változóNév.moveTo(x,y);
  - Ablak vízszintes és függőleges méretének megnövelése x ill. y pixellel:
    változóNév.resizeBy(x,y);
  - Ablak vízszintes és függőleges átméretezése x ill. y pixelesre:
    változóNév.resizeTo(x,y);
  - Ablak tartalmának elmozdítása balra x ill. felfelé y pixellel:
    változóNév.scrollBy(x,y);
  - Az ablak-tartalom (x,y) pixel-koordinátáknak megfelelő pontjának illesztése az ablak bal felső sarkához:
    változóNév.scrollBy(x,y);
  - Visszalépés az előző oldalra:
    window.back();
  - Továbblépés a következő oldalra:
    window.forward();
  - Átlépés a főoldalra:
    window.home();

- A dokumentum betöltésének leállítása:
  window.stop();
- Nyomtatás indítása (a nyomtató-párbeszédpanel meghívása):
  window.print();
- A jelenlegi ablakot megnyitó ablak elérése:
  parentWindow = window.opener;
- A jelenlegit megnyitóból megnyitott másik ablak elérése:
  parentWindow = window.opener;
  otherWindow = parentWindow.változóNév2;
- A jelenlegit megnyitóból megnyitott másik ablak bezárása:
  parentWindow = window.opener;
  otherWindow = parentWindow.változóNév2.close();
- A jelenlegit megnyitó ablakhoz tartozó függvény hivatkozása:
  parentWindow = window.opener;
  parentWindow.függvényNév();

- Ablak-időtartam:
  változó-név = setInterval(’kód’ , x);
  Az idézőjelek közé írt JS-kódot (vagy első paraméterként beírt JS-módszert/függvényt) x milliszekundumonként végrehajtja.
  clearInterval(változó-név);
  A változó-névvel jelölt kód-végrehajtogató ritmust leállítja.
- Ablak-időzítés:
  változó-név = setTimeout(’kód’ , x);
  Az ablak megnyitásától számított x. milliszekundumig késlelteti a JS ’kód’ végrehajtását.
  clearTimeout(változó-név);
  Ha a ’kód’ még nem futott le, leállítja az időzítést (tehát a ’kód’ nem is fog lefutni).
- Mivel az <a> tagnek a href kötelező attribútuma, azt feltétlen meg kell adnunk. Hogyha az <a> tag csak a tabulálhatóságot szolgálja, azaz egy event handlerrel meggátoljuk, hogy a böngésző kövesse a linket, és az eseményt mi magunk határozzuk meg, akkor „placebo” href-et kell megadnunk. E célra leggyakrabban a következő kód szolgál:

- A push módszerrel egy újabb részletet illeszthetünk egy array végéhez. A módszer az új array hosszát adja vissza:
  array1 = new Array();
  array2 = [”elem1”,”elem2”,”elem3”];
  array1.push(array2); vagy egyes elemekre: array1.push(array2[0] , array2[1] , array2[2]);
- Valamely elemben lévő HTML-tartalmat az
  elem.innerHTML;
  kifejezéssel vétethetünk fel (stringként).
- Olyan elemekhez is hozzá lehet rendelni pl. class attribútumot, melyeknek még nincs értékük, de formálisan már definiálva vannak. Pl.:
  popupElement = null;
  DOMhelp.cssjs(’add’ , popupElement , popupClass);
  popupElement = document.createElement(’div’);
- Pozícionálási jellemzők (236/261. oldal):
  - Magasság (px):
    elem.offestHeight;
  - Az elem és szülő-eleme bal szélei közti távolság (px):
    elem.offsetLeft;
  - Az elem és szülő-eleme teteje közti távolság (px):
    elem.offsetTop;
  - Az elem szülőjére vonatkozó értékek felvétele (itt magasság):
    elem.offsetParent.offsetHeight;
    vagy

parent = elem.offsetParent;
parent.offsetHeight;
Ha nincs szülő-elem, az elem.offsetParent; értéke null!

- Elem abszolút pozícionálása a bal-felső sarokhoz képest:
  elem.style.left = 10px;
  elem.style.top = 10px;
- Az <inframe></iframe> elem egy inline frame-et jelent, melyben bármilyen, az src attribútummal hivatkozott dokumentumot (pl. HTML) megjeleníthetünk. Az <iframe></iframe> elembe írt szöveges tartalmat csak az <iframe>-et nem támogató böngészők jelenítik meg.

7. fejezet:

Navigáció és formok

- A temp elemen belüli, ’id’ ID-jű elem felvétele (változóként):
  változóNév = temp.elements[’id’];
- Az elemNév nevű option list kiválasztott tagja (pl.: <option value=”no_1”>no_1</option>) értékének felvétele (változóként):
  változóNév = elemNév.options[elemNév.selectedIndex].value;
- A window.location objektum jellemzői (melyeket az aktuális oldal URL-éből olvas ki, pl. http://www.example.com:8080/index.php?s=JavaScript#searchresults):
  - window.location.hash;
    Az anchor neve (#searchresults)
  - window.location.host;
    Domain név (www.example.com)
  - window.location.hostname;
    Domain, subdomain név és portszám (www.example.com:8080)
  - window.location.href;
    A teljes URL (http://www.example.com:8080/index.php?s=JavaScript#searchresults)
  - window.location.pathname;
    Elérési út (/index.php)
  - window.location.port;
    Portszám (8080)
  - window.location.protocol;
    Protokol (http:)
  - window.location.search;
    Keresési paraméterek (?s=JavaScript)
- A window.location objektum search jellemzőjének felülírása:
  winow.location.search = ’?DOM scripting’;
- A window.location objektum módszerei:
  - window.location.reload();
    A dokumentum újbóli letöltése.
  - window.location.replace(URL);
    A böngészőt az URL-re küldi; a korábbi URL pedig nem jelenik meg a böngészési előzményekben. Így a korábbi oldalra való visszalépés sem lehetséges!
- A window.history objektum az ablakban megjelent összes oldal URL-jét tartalmazza. Egyetlen jellemzője a
  window.history.length;
  ami ezen URL-ek számát jelenti.
- A window.history objektum módszerei:
  - window.history.back;
    visszalépés az előző oldalra.
  - window.history.forward;
    Előrelépés a következő oldalra.
  - window.history.go(n);
    Az n egy egész szám. Ha n<0, akkor az ablak visszalép az n-edik oldalra, ha 0<n, akkor előrelép az n-edik oldalra, ha pedig n=0, akkor újratölti az oldalt.
- Az (oldalon belüli) könyvjelzőzéshez használt <a> elemek szabványos hivatkozó attribútuma a href (pl. href=”#elem1”), hivatkozottja pedig az id (pl. id=”elem1”); de a régebbi böngészők még a name attribútumot támogatták. Ezért célszerű mindkettőt alkalmazni; pl. egy oldalon belüli link (könyvjelző) esetében:
  Hivatkozó
  Hivatkozott
- Hogyha IE-ben tabulálással próbálunk elérni egy oldalon belüli könyvjelző-listából egy könyvjelzőt, akkor az IE csak akkor helyezi át a fókuszt a hivatkozott köbnyvjelzőre, ha az egy

meghatározott (pl. 100%-os) szélességű elembe van ágyazva (pl. egy elembe)!!!

- Az alábbi kifejezés egy URL-ből csupán a # után álló részt, azaz a könyvjelző-nevet szolgáltatja egy változó értékéül:
  változóNév = elem.getAttribute(’href’).replace(/.*#/ , ’’);
- Az oldalon belüli navigációs listáknál („füles” megoldások) nem érdemes a link-követést meggátolni, mert ez a kód futását nem befolyásolja, viszont lehetővé teszi az oldal könyvjelzőzését és a JS-et nem támogató felhasználók számára a navigáláshoz is szükséges.
- A cloneNode() módszer az objetumon értelmezett eseményeket nem klónozza, azokat a másodpélányhoz ismét hozzá kell(het) adni!
- Formok elérési módjai:
  - A DOM-on keresztül:
    - getElementsByTagName();
    - getElementById();
  - A forms objektumon keresztül:
    - Helyszámmal:
      document.forms[2];
    - Objektumként értelmezhető name attribútummal:
      document.forms.formNév;
    - Csak stringként értelmezhető (pl. különleges karaktereket tartalmazó) name attribútum alapján:
      document.forms.[’formNév’];
- A form-típusú objektumok jellemzői:
  - A dokumentumban lévő formok száma:
    .length;
  - A form továbbításakor meghívott script:
    .action;
  - A form kódolása, azaz a <form> elem enctype attribútumának értéke:
    .encoding;
  - A form-továbbító művelet típusa (POST vagy GET):
    .method;
  - A form neve (a name attribútum értéke):
    .name;
  - A cím, ahová a form adatait továbbítjuk:
    .target;
- A form-típusú objektumok módszerei:
  - A form eredeti állapotba való visszatérítése, azaz az üres mezőkbe beírtak törlése és a value, selected ill. checked attribútumok által meghatározott preset visszaállítása.
    .reset();
  - A form továbbítása:
    .submit;
- A forms objektumot formon belüli elem felvétele a form objektumból:
  - Elem felvétele helyszám alapján:
    var formNév; (A formok forms objektumbóli felvételét lásd feljebb!)
    var elem = formNév.elements[0];
  - Elem felvétele objektum-névvel:
    var elem = formNév.elements.elemNév;
  - Elem felvétele string-névvel:
    var elem = formNév.elements.[’elemNév’];
- Formon belüli elemek száma (a forms.elements objektum csak olvasható jellemzője):
  formNév.elements.length;
- A formon belüli elemek lehetséges jellemzői:
  - Kipipáltság (logikai érték):
    checked;

    - Eredeti kipipáltság (logikai értéke IGAZ, ha az elem eleve ki volt pipálva):
      checked;
    - Érték (a value attribútum felhasználói értéke!):
      .value;
    - Eredeti érték (a value attribútum eredetileg megadott értéke!]:
      .value;
    - Az elemet tartalmazó form visszaadása:
      .form;
    - Név (a name attribútum értéke):
      .name;
    - Típus (az elem típusa):
      .type;
- A formon belüli elemek lehetséges módszerei:
    - Az elem kijelölésének megszüntetése:
      .blur();
    - Az elem kijelölése:
      .focus;
    - Az elemre-klikkelés szimulációja:
      .click;
    - Szövegmezők tartalmának kijelölése:
      .select;
- A többi attribútumot a korábban megismert, általános DOM-módszerekkel módosíthatjuk, ha az objektumot a forms.elements objektumból kijelöltük. Az attribútum felvételekor ill. módosításakor ellenőrizzük az elem típusát, mert bizonyos elemek egyes attribútumokkal eleve nem rendelkezhetnek!
- A defaultValue érték(ek) beálítása után az adott formot illik visszaállítani a kezdőállapotba (hiszen ekkor elvileg egy attribútumot módosítottunk, amit a böngésző már nem olvasna be). Ez az IE-ben szükséges, a normális (értsd: szabványos) böngészőkben (pl. Firefox) azonban gyakorlatilag felesleges, mivel utóbbiak maguktól is észreveszik a változást.
- A name=”name” attribútummal rendelkező rádió-gombok felvétele egy dokumentum első <form> eleméből:
  var változóNév = document.forms[0].elements.name;
- Gombok típusai:
    - submit: formot továbbít.
    - reset: a form összes mezőjét az eredeti értékre állítja vissza
    - push: nincs alapértelmezett viselkedésük, azt kliens-oldali JS-tel kell hozzájuk rendelni.
- Annak ellenőrzése, hogy egy elem (típusa) kép vagy submit-e:
```
for(…)
{
   if( !/submit|image/.test(elem) ){continue;}
}
```
- A continue; rendelkezéssel csak az adott ciklus-lépést, a break;-kel az egész ciklus futását megszakítjuk.
- Az <input type=”image” name=”név”> elem kattintásakor a név.xxx.yyy üzenet érkezik a szerverhez. A szerver-oldali szcript így nemcsak az elem aktiválásának tényét, hanem annak helyét is figyelembe veheti a reagáláshoz. (Ilyen beviteli elemet ritkán alkalmaznak.)
- Egy adott select box felvételéhez a helyszám helyett az elem name attribútumát használjuk, mivel a select boxot jelentő HTML-elem <select> ill. <select-multiple> egyaránt lehet. Egy selectName nevű <select> elem (azaz egyszerű select box) éppen kijelölt opcióját a következőképpen olvashatjuk be:
  var változóNév = document.forms[0].elements[’selectName’].selectedIndex;
- Array-objektum gyors definiálása:
  var arrayNév = [];

- Új opció definiálása [a new Option(); konstruktorral]:
  var változóNév = new Option(’name’ , ’text’ , defaultSelected(=1/0) , selected (=1/0));
- Új elem hozzáadása egy array végéhez (érdekes megoldás):
  arrayNév[arrayNév.length] = új Elem;
  minthogy az array hossza mindig eggyel nagyobb, mint a legutolsó (meglévő) elem helyszáma.
- Formok dinamizálása: a ki nem töltendő elemek eltüntetése (hide) ill. kikapcsolása (disabled=1).
- Habár a forms ill. forms.elements objektumok csak HTML-re értelmezettek, tehát a DOM-mal ellentétben pl. XML stringekre nem érvényesek, még minig egyszerűbb megoldást kínálnak a form-beviteli elemek kezelésére, mint a form elemek alelemein értelmezett (DOM szerinti) ciklusok.

8. fejezet:

Szerver-oldai scriptek és Ajax-interakció.

- Ajax = Asynchronous JavaScript and XML; webalkalmazás-fejlesztési módszer.
- A frame-eket a szikron webes lekérdezések korában alkalmazták a letöltendő tartalom csökkentésére.
- Az aszinkron hívásokkal a felhasználói- és szerver-oldal közti folyamatos kommunikáció révén csak az oldal változásai kerülnek letöltésre.
- Az Ajax segítségével a kliens JS-lekérdezéseket intézhet a szerverhez, az oldal újratöltése nélkül. A kapott tartalom XML, amit az Ajax HTML-lé alakít.
- Így egy formot anélkül ellenőrizhetünk/továbbíthatunk a kliensoldalon és ellenőrizhetünk/regisztrálhatunk a szerveroldalon, és jeleníthetjük meg a válasz-üzenetet, hogy az oldalt frissíteni kellett volna.
- XML HTTP Request = XHR; http lekérdezés helyett az Ajax ilyennel fordul a szerverhez. A Safari/Fireox/Opera alkalmazza; az IE az ActiveX-et használja. Az IE tehát csak akkor tud Ajax hívásokat indítani, ha mind a JS, mind az ActiveX engedélyezve van rajta. (Egyébként hibát jelez.)
- XHR- ill. ActiveX-lekérdezés (objektum) létrehozása try…catch rendelkezéssel. Ha a böngésző egyiket sem támogatja, akkor a triggerelő link eredeti href-jét követi a böngésző:

```
var request;
try
{
   request = new XMLHttpRequest();
}
catch( error )
{
   try
   {
      request = new ActiveXObject( "Microsoft.XMLHTTP" );
   }
   catch( error )
   {
      return true;
   }
}
```

- 330