



Juhász Tibor \* Kiss Zsolt

# Tanuljunk programozni

Visual Basic Script  
Objektumok  
Web





# **T a n u l j u n k p r o g r a m o z n i**



**Juhász Tibor  
Kiss Zsolt**

# **T a n u l j u n k p r o g r a m o z n i**

**Visual Basic Script  
objektumok  
web**

LEKTOR

**Kuzmina  
Jekatyerina**



**COMPUTERBOOKS  
BUDAPEST, 2004**



Jelen dokumentumra a Creative Commons Nevezd meg! – Ne add el! – Ne változtasd meg! 3.0 Unported licenz feltételei érvényesek: a művet a felhasználó másolhatja, többszörözheti, továbbadhatja, amennyiben feltünteti a szerzők nevét és a mű címét, de nem módosíthatja, és kereskedelmi forgalomba se hozhatja.

**A könyv nyomtatott változatát vagy annak részleteit a Kiadó engedélye nélkül bármilyen formában vagy eszközzel reprodukálni, tárolni és közölni tilos.**

A könyv készítése során a szerzők és a Kiadó a legnagyobb gondossággal jártak el.

Ennek ellenére hibák előfordulása nem kizárható.

Az ismeretanyag felhasználásának következményeiért sem a Kiadó, sem a szerzők felelősséget nem vállalnak.

Minden jog fenntartva.

© Juhász Tibor, Kiss Zsolt, 2004

ISBN 963 618 316 3

© Kiadó: ComputerBooks Kiadói Kft  
1126 Budapest, Tartsay Vilmos u. 12.  
☎: 3751-564, 📠: 3753-591  
E-mail: [info@computerbooks.hu](mailto:info@computerbooks.hu)  
<http://www.computerbooks.hu>

Borítóterv: Székely Edith

Felelős kiadó: ComputerBooks Kft ügyvezetője

Nyomdai munkák: Pressman Nyomdaipari Bt., Dabas

# TARTALOMJEGYZÉK

ELŐSZÓ.....	1
BEVEZETÉS.....	3
<b>I. A PROGRAMOZÁS ALAPJAI.....</b>	<b>7</b>
1. PROGRAMOK ÉS OBJEKTUMOK .....	9
1.1. Informatikai alapok .....	9
1.2. Objektumok és események.....	14
2. BEVEZETÉS A HTML-BE .....	19
2.1. A weblapok felépítése és kódolása.....	19
2.2. A weblap szövege .....	24
2.3. Események a weblapokon .....	30
2.4. Az objektumok metódusai.....	34
2.5. Parancsgombok használata.....	37
2.6. Szövegek beírása és megjelenítése .....	39
2.7. Konténerobjektumok.....	43
3. A VBSCRIPT ALAPJAI .....	47
3.1. Programok a HTML-kódban .....	47
3.2. Változók a programban .....	53
3.3. Aritmetikai műveletek.....	63
3.4. Függvények .....	73
3.5. Eljárások.....	83
3.6. HTML-alkalmazások (HTA).....	91
4. VEZÉRLŐSZERKEZETEK .....	93
4.1. Szelekciók .....	93
4.2. Többágú szelekciók.....	103
4.3. Összetett feltételek .....	110
4.4. Iterációk .....	120
5. TÖMBÖK ÉS TÁBLÁZATOK .....	137
5.1. Egydimenziós tömbök.....	137
5.2. Kollekciónak.....	145
5.3. Többdimenziós tömbök.....	150
5.4. Dinamikus tömbök .....	160
5.5. Tömbök alkalmazása.....	170
<b>II. PROGRAMOZÁS HALADÓKNAK.....</b>	<b>183</b>
6. VÁLTOZÓK ÉS KIFEJEZÉSEK.....	185
6.1. A változók altípusai.....	185
6.2. Numerikus változók .....	194
6.3. Karakterláncok kezelése.....	201
6.4. Dátumok és időpontok .....	212
6.5. Változók az alprogramokban.....	215

7. ESEMÉNYKEZELÉS .....	223
7.1. Az event-objektum .....	223
7.2. Egér-események .....	229
7.3. Billentyűzet-események .....	235
7.4. Hibakezelés a szkriptekben .....	239
8. DHTML HALADÓKNAK.....	245
8.1. Képek a weblapon .....	245
8.2. Időzítők használata .....	253
8.3. Űrlapok.....	258
8.4. Kommunikáció a weblapok között.....	264
8.5. Objektumok beillesztése .....	271
8.6. Párbeszédablakok és menük.....	278
9. ÖSSZETETT ADATTÍPUSOK .....	287
9.1. Tömbök és listák .....	287
9.2. Szótárak és halmazok .....	293
9.3. Objektumok.....	299
9.4. Rekordok.....	312
10. FÁJLKEZELÉS.....	317
10.1. A fájlrendszer objektumai .....	317
10.2. Szövegfájlok.....	321
10.3. Nyomtatás .....	328
11. ALGORITMUSOK .....	333
11.1. Érték hozzárendelése sorozathoz.....	334
11.2. Rendezési algoritmusok .....	339
11.3. Sorozatok hozzárendelése sorozatokhoz .....	343
11.4. Rekurzív algoritmusok .....	346
11.5. A visszalépéses keresés (back track).....	351
11.6. A programfejlesztés folyamata.....	355
<b>FÜGGELÉK .....</b>	<b>361</b>
F1. A TEXTPAD ALKALMAZÁS .....	363
F2. A MICROSOFT SCRIPT DEBUGGER .....	367
F3. A MICROSOFT SCRIPT EDITOR .....	373
F3.1. A Script Editor telepítése és felépítése.....	373
F3.2. A Script Editor használata.....	377
F3.3. Hibakeresés a Script Editorral.....	380
F4. FELADATGYŰJTEMÉNY .....	385
IRODALOMJEGYZÉK .....	409
TÁRGYMUTATÓ .....	411



# ELŐSZÓ

Kirándulásra hívjuk az Olvasót a számítógépes programozás területére. A számítógépek manapság nélkülözhetetlen részét képezik életünknek. Pénztárgépekkel összekötte végzik az áruházak raktári nyilvántartását csakúgy, mint a repülőgépek, repülőterek forgalomirányítását. Számítógépeket használunk a játékprogramokhoz, és a Naprendszer távoli vidékeit felderítő űrszondák vezérléséhez.

Hogyan tud egyetlen eszköz ennyiféle feladatot ellátni? A titok nyitja a számítógépes program. Program nélkül a számítógép nem működik, programokkal elvileg bármilyen feladatot el tud végezni. Korlátot legfeljebb a gép sebessége, a memória mérete vagy hasonló fizikai paraméterek jelentenek.

Könyvünk megismerteti az Olvasót a számítógépek programozásával. Barátságos vidékről indulunk, nem tételezünk fel semmilyen programozási ismeretet. Némi felszerelésre szükségünk lesz, de hátizsákunkat csak a Windows kezelésének alapismerteivel kell megtöltenünk. Ha be tudjuk kapcsolni a gépet, el tudjuk indítani az Intézőt, meg tudjuk nézni egy mappa tartalmát, akkor máris kezdhethetjük.

Utunk változatos tájakon fog haladni. Az árnyasabb erdők után újra kisüt majd a nap. Néha meredekebb lesz az emelkedő, ekkor célszerű lassabban haladni, pihenőket tartani. Mire sziklafalakhoz érünk, már elég erősek leszünk. Felszerelésünk segítséget nyújt a megmászásukhoz. A hegytetőn elénk táruló látvány kárpótol a fáradozásért. A csodálatos panoráma felfedi a programozás titkait, túránk során megerősödve képesek leszünk az általunk kitűzött feladatok megoldására készíteni a számítógépet.

A programozási ismeretek elsajátításához a Visual Basic Script nyelvet választottuk. Használatához egy számítógépen és a Windows operációs rendszeren kívül nincsen szükség semmilyen más eszközre. A Windows bármelyik változata (Windows 95, 98, ME, NT, 2000, XP) megfelel. Programjainkat az operációs rendszer részét képező Jegyzetomb alkalmazás segítségével írjuk meg. A könyv CD-mellékletén található kényelmesebb fejlesztőeszköz használatát a függelékben ismertetjük.

Az egyes fejezetek konkrét példákon keresztül tanítják meg az Olvasót a programozási ismeretekre. A könyvünkben bemutatott közel 500 példa teljes, működő programja megtalálható a CD-n, így nem kell a gépeléssel fáradnia. A függelékben kitűzött feladatok részletes megoldása szintén szerepel a CD-n.

Programjainkat egy web-böngésző segítségével futtatjuk, de nincs szükségünk Internet-kapcsolatra. Az Internet Explorer része a Windowsnak, nem kell külön telepíteni. Néhány példától eltekintve a 4-es vagy 5-ös változata is megfelel. A könyv írásánál a 6-os verziót használtuk, melyet a legtöbb számítástechnikai újság CD-mellékletén megtalálhatunk, vagy az Internetről letölthetünk. Az Explorer helyett bármilyen böngészővel útnak indulhatunk, amely kezelni tudja a Visual Basic Script programokat.

A Visual Basic Script a weblapok egyik programozási nyelve. Az Olvasó tehát nem csak a programozás alapjaival ismerkedik meg, hanem képes lesz ügyes és okos weblapok készítésére, melyeket lehetőség szerint az Internetre is feltehet. Ha főleg a

web programozása érdekli, akkor a 2., 3., 4., 5.2., és 8. fejezetet nézze át. Reméljük, közben megjön a kedve a kihagyott részek megismeréséhez.

A Visual Basic Script további előnye, hogy alapját képezi a Visual Basic for Applications programnyelvnek, mellyel a Word-öt, az Excelt és más Windows-alkalmazásokat lehet programozni. Ha sikerült felkelteni az érdeklődést a programozás iránt, akkor könnyű áttérni a Visual Basicre, amely a Visual Basic Script nagy testvére, és sokkal bővebb eszköztárral rendelkezik. Használatához azonban meg kell vásárolni a fejlesztőrendszert.

A Visual Basic Script objektumalapú programozási nyelv. Segítségével megismerkedünk a korszerű programozási nyelvek alapvető fogalmával, az objektummal. Látni fogjuk, hogy az objektumok a számítógépek programozásának természetes eszközei, nélkülük semmire se megyünk. Könyvünk az objektumok fogalmára *alapozva* tekinti át a programozási ismereteket.

A Visual Basic Script segítségével úgynevezett HTML-alkalmazásokat is készíthetünk, melyeket már semmi sem különböztet meg egy szokásos Windows-programtól. Tudomásunk szerint ez az első olyan magyar nyelven megjelenő könyv, amely ismerteti a hta programok írásának módját.

Könyvünk tartalmazza a 2005-ben bevezetésre kerülő közép- és emelt szintű informatika érettségi weblap készítés, algoritmizálás, adatmodellezés és programozási ismeretek fejezeteinek teljes anyagát. Szeretnénk megmutatni, hogy a Visual Basic Script erre a célra is alkalmazható. Segítségével egyszerűen és könnyen írhatunk látványos programokat, kezelhetjük a grafikus felhasználói felületet, használhatjuk a Windows eszközkészletét.

## **Köszönetnyilvánítás**

Szeretnénk köszönetet mondani kollégáknak, Szűcs Gergelynek és volt tanítványunknak, Böjtös Beának, amiért figyelemmel kísérték, megjegyzéseikkel segítették a könyv megírását.

Külön köszönetet mondunk Devecz Ferenc kollégáknak, akinek észrevételeiből, tanácsaiból sokat tanultunk. Nélküle nem született volna meg ez a könyv.

Hálásak vagyunk Sandra Rawlinsonnak és a Helios Software Solutionsnek, amiért engedélyezték, hogy a TextPad szövegfájl szerkesztő programjuk szerepelhessen a könyvhöz mellékelt CD-n.

Köszönet illeti Kuzmina Jekatyerinát és Tóth Bertalant. Lektori feladataikon túlmenően is hozzájárultak a DHTML-objektumok újszerű szemléletmódjának a kialakításához. Gondos és precíz munkával gyomlálták ki a kézirat hibáit.

Az egyik szerző, Juhász Tibor hálával tartozik feleségének, Zsuzsának, amiért biztosította a könyv megírásához szükséges nyugodt feltételeket, és elviselte a fáradságos munkával járó feszültségeket.

# BEVEZETÉS

## A könyv szerkezete

Könyvünk két részből áll. Az első részben röviden összefoglaljuk a megértéshez szükséges informatikai alapokat, majd megismerkedünk legfontosabb eszközeinkkel, az objektumokkal. A Visual Basic Script programokat egy böngésző segítségével hajtjuk végre, ezért áttekintjük a weblapok készítésének alapelemeit is. Ezekkel a tudnivalókkal felvértezve megismerkedünk a Visual Basic Script nyelv utasításaival, a programozás során felhasznált változókkal, kifejezésekkel, vezérlési szerkezetekkel, tömbökkel. Nem tárgyaljuk meg részletesen ezeket az eszközöket, mindig csak annyi ismertetet közlünk, amennyire a továbbiakhoz szükségünk van. Az első részt egy nagyobb feladat kidolgozásával zárjuk, amelyben bemutatjuk egy összetett program készítésének a menetét.

A könyv második részében már részletesen ismertetjük a Visual Basic Script által nyújtott lehetőségeket. Kitérünk a hatékony programok írásához szükséges tudnivalókra. Megismerkedünk olyan objektumokkal, melyekkel látványos és felhasználóbarát weblapokat készíthetünk. Az összetett adattípusok és a fájlkezelés segítségével már bármilyen programozási feladatot megoldhatunk. Végül ismertetjük a programokban előforduló leggyakoribb elemi algoritmusokat.

A könyvben csak azokra az eszközökre hivatkoztunk, melyek elérhetőek minden olyan számítógépen, amelyen Windows operációs rendszer fut (Jegyzettömb, Internet Explorer). A függelékben kitérünk a programok írását, weblapok szerkesztését elősegítő TexPad és a hibakeresést megkönnyítő Microsoft Script Debugger kezelésére. Ismertetjük a Microsoft Office 2000, illetve XP programcsomag részét képező Microsoft Script Editor grafikus fejlesztői rendszert, amellyel gyorsan és hatékonyan készíthetjük el a programjainkat.

## A CD-melléklet tartalma

A programozási ismereteket közel ötszáz, részletesen kidolgozott példán keresztül mutatjuk be. A könyvhöz mellékelte CD tartalmazza mindegyik példa teljes, működő forráskódját. Így az Olvasónak nem kell a gépeléssel fáradnia, és egyből kipróbálhatja a tanultakat. A példák a fejezeteknek megfelelő Fejezet02 – Fejezet11 nevű mappákban találhatók. Egy mappán belül a fájlokat sorszámoztuk.

A gyakorlást a függelékben kitűzött feladatok segítik. A feladatok a könyv szerves részét képezik. A részletes megoldások a CD-melléklet Feladatok mappájában található. Javasoljuk az Olvasónak, hogy feltétlenül nézze át ezeket a megoldásokat!

A korlátozott terjedelem miatt nem kerültek be a könyvbe a nagyobb táblázatok, a weblapok formázását végző stílusok részletes leírása és a Visual Basic Script hibaüzenetei. Ezeket a Dokumentumok mappa weblapjai ismertetik. Könyvünkben hivatkozunk a megfelelő fájlokra.

A függelékben tárgyalt TextPad telepítő készlete a CD-melléklet TextPad mappájában található. Telepítését a program bemutatásánál ismertetjük.

A CD-n lévő fájlokat a Windows Intéző segítségével nyithatjuk meg. A főkönyvtárban elhelyeztünk egy Tartalom.htm nevű állományt, melynek segítségével az Olvasó könnyebben eligazodhat a lemezen.

### **A könyvben alkalmazott jelölések**

A könyvben folyamatosan hivatkozunk a CD-n szereplő példákra. A hivatkozásokat félkövér betűvel írtuk, hogy jól kiemelkedjenek a szövegből. Az egyes példák programrészleteit Courier betűvel szedtük. A dőlt betűs részeket az alkalmazás során helyettesíteni kell a megfelelő kifejezésekkel (változónevekkel, utasításokkal stb.).

A szövegben szereplő kulcsszavakat, változóneveket, objektum-azonosítókat is dőlt betű jelzi. A legfontosabb fogalmak definícióit keretezéssel emeltük ki. A szakki-fejezések magyarázata mellett az angol nyelvű szavak magyar fordítását szintén megadtuk. A szó szerinti fordítás helyett a jelentéshez illeszkedő kifejezést kerestünk.

A könyvben szereplő lábjegyzetek a tájékozottabb felhasználók számára készültek. Így a könyv olvasásakor figyelmen kívül hagyhatók.

### **Források az Interneten**

Könyvünk programozni tanítja az Olvasót. Nem nyújthatja a weblapok objektumainak és a Visual Basic Script eszközeinek teljes körű leírását. Ezek részletes ismertetése megtalálható a Microsoft Development Network weblapján:

<http://msdn.microsoft.com/library>

Sok kérdésre kaphatunk választ a Visual Basic Script hírcsoportjában:

<news://microsoft.public.scripting.vbscript>

Ha az Internet kiszolgálónk nem tárolja ezt a hírcsoportot, akkor használhatjuk a Microsoft szerverét: [news.microsoft.com](http://news.microsoft.com). A hírcsoportokat a weben keresztül is elérhetjük, például a

<http://www.google.co.hu>

Csoportok kategóriájában.

Könyvünk legtöbb példáját az Internet Explorer 4.0-val is kipróbálhajtuk, amely a Visual Basic Script 3-as változatát ismeri. Főleg a saját objektumok kezelésével kapcsolatban több utasítást a Visual Basic Script 5-ös változatában vezettek be. Ezek végrehajtásához az Internet Explorer 5.0 telepítése szükséges, vagy a programnyelv bővítése letölthető a következő weblapról:

<http://msdn.microsoft.com/downloads/list/webdev.asp>

Válasszuk a használt operációs rendszernek megfelelő „Microsoft Windows Script” feliratot. Erről a weblapról tölthető le a programnyelv dokumentációja is („Microsoft Windows Script Documentation”).

## A példafájlok használata

A Windowsban egy állomány azonosítója két részből áll. Az első rész a fájl neve, a második rész az úgynevezett kiterjesztés. A kiterjesztés az állomány tartalmára utal. Példafájljaink kiterjesztése .htm, ami a weblapok egyik gyakori jelölése.

A Windows Intéző alapértelmezés szerint nem mutatja meg a fájlok kiterjesztését. A listában egy kis ábra jelzi a típusukat. Az Internet Explorer használatakor a .htm fájlokat kék e-betű jelöli. Ha duplán kattintunk egy ilyen állományra, akkor elindul az Internet Explorer, és megjeleníti a weblapot.

A könyvben hivatkozni fogunk a CD-n lévő példafájlok tartalmára, az úgynevezett forráskódra. Ezt a legegyszerűbben úgy nézhetjük meg, ha az Intézőből dupla kattintással megnyitjuk az állományt a böngészőben, majd kiválasztjuk a Nézet menüpont Forrás parancsát. A Windows elindítja a Jegyzetömböt, és megjeleníti a fájl tartalmát.

A programozás során ilyen fájlokat írunk és módosítunk. A módosításokat a Jegyzetömb ablakában végezhetjük el. Ne feledkezzünk meg a mentésről! Ha a CD-ről nyitottuk meg az állományt, akkor a Jegyzetömb Fájl menüjéből válasszuk ki a „Mentés másként” parancsot, és jelöljük ki a merevlemezen egy mappát. Célszerű a gyakorláshoz külön mappát létrehozni. A mentésnél a fájlnev után feltétlenül írjunk be egy pontot és a .htm kiterjesztést. Ha jól csináltuk, a kijelölt mappában létrejön az e-betűs ikonnal rendelkező állomány.

Zárjuk be a böngészőt, és az Intéző segítségével nyissuk meg az elmentett fájlt. A további változtatások után elegendő a Jegyzetömb Mentés menüpontját választani. Ha a példafájlokat az Intéző segítségével átmásoljuk a merevlemezre, akkor legelőször sincs szükség a „Mentés másként” parancsra.

A módosítások csak akkor jelennek meg a weblapon, ha a mentés után visszaváltunk az Internet Explorerre, és a Frissítés gombra kattintunk.

A módosítás folyamata tehát a következő lépésekből áll:

1. A weblap megnyitása.
2. Nézet/Forrás menüpont kiválasztása a böngészőben.
3. A módosítások elvégzése a megnyíló Jegyzetömb ablakban.
4. Mentés a Jegyzetömbben.
5. Átváltás a böngésző ablakára.
6. A weblap frissítése a böngészőben.

Új állomány létrehozásához a legegyszerűbben a CD-melléklet főkönyvtárában lévő Üres.htm fájlt használhatjuk fel. Nyissuk meg ezt az Intézőből dupla kattintással. (Az Internet Explorer egy üres dokumentumot mutat.) Kattintsunk a Nézet/Forrás menüpontra. Írjunk a megjelenő Jegyzetömbbe bármilyen szöveget, és mentsük el a merevlemezre Próba.htm néven, de ne zárjuk be a Jegyzetömböt! Ezzel el is készítettük első weblapunkat, melyet az Intézőből nyithatunk meg. A további módosításokhoz a nyitva lévő Jegyzetömb ablakot használhatjuk. A változtatások után elegendő elmenteni (Fájl/Mentés), visszaváltani a böngésző ablakába, és frissíteni a dokumentumot.

A fájlokat a Jegyzettömb mellett bármilyen szövegszerkesztő programmal elkészíthetjük. A mentést azonban egyszerű szöveggként végezzük el. Ne alkalmazzunk semmilyen formázási parancsot. A Microsoft Word a weblapként elmentett dokumentumokat számos kiegészítéssel látja el, így ne használjuk ezt a lehetőséget!

# **I. A PROGRAMOZÁS ALAPJAI**





# 1. PROGRAMOK ÉS OBJEKTUMOK

Ebben a fejezetben összefoglaljuk a programozáshoz szükséges informatikai alapismereteket. Csak a könyvben felhasznált fogalmak magyarázatára térünk ki. Részletebben foglalkozunk a programozás alapelemeivel, a továbbiakban központi szerepet játszó objektumokkal és eseményekkel.

## 1.1. Informatikai alapok

### Számrendszerek

A számítógép működését a kettes számrendszer segítségével lehet modellezni. A mikroprocesszor, a memória, a háttértár nullák és egyesek sorozatát dolgozza fel, illetve tárolja. A kettes számrendszer latin eredetű elnevezése alapján ezt bináris kódnak nevezzük. A 0 és az 1 számjegyet pedig bitnek hívjuk.

A bináris kódot nehéz áttekinteni. A nullák és egyesek írása, olvasása sok hibára ad alkalmat. Ezért az informatikában gyakran a 16-os számrendszert alkalmazzák. Ez az úgynevezett hexadecimális kód 0-tól 9-ig a tízes számrendszer számjegyeit használja, további számjegyeit pedig az ábécé betűivel jelöljük:

10: A    11: B    12: C    13: D    14: E    15: F

Az átváltás a 2-es és a 16-os számrendszer között sokkal könnyebb, mint a 2-es és a 10-es között. Az átszámításra nem lesz szükségünk, ezért nem tárgyaljuk. Az érdeklődő Olvasó az alapfokú informatika könyvekben megtalálja a módszert.

A bináris és hexadecimális elnevezések mintájára a 10-es számrendszert gyakran decimális számrendszernek hívják.

### A karakterek kódolása

A tárolás és adatáramlás során a biteket általában nyolcasával csoportosítjuk. 8 bit alkot egy bájtot. A bájt egy 0 és 255 közé eső egész szám, azaz 256-féle jel tárolására alkalmas. Ez a tartomány az átszámítás módjából következik.

Ha a bájtot a billentyűzeten vagy a képernyőn megjelenő karakterek (betűk, számjegyek, írásjelek) kódolására használjuk, akkor rögzítenünk kell, hogy melyik szám melyik karaktert jelöli. Egy nagyon elterjedt rendszert az angol nyelvű rövidítés alapján ASCII-kódnak hívunk. Az ASCII-kód 0-tól 32-ig speciális, úgynevezett vezérlő karaktereket jelöl (köztük a tabulátort vagy a soremelést). Utána következnek a számjegyek, írásjelek és az angol ábécé betűi. 128-tól 255-ig eredetileg egy-két ékezetes karaktert, továbbá matematikai és grafikus jeleket tartalmazott, melyek segítségével például táblázatokat lehetett egyszerűen bekeretezni.

Néhány betű ASCII-kódja:

A: 65    B: 66    Z: 90    a: 97    b: 98    z: 122

Figyeljük meg, hogy a nagybetűk kódja kisebb, mint a kisbetűké!

A Windows által használt ANSI-kód 0-tól 127-ig megegyezik az ASCII-kóddal. 128-tól a különböző nyelvekben – köztük a magyarban – előforduló ékezetes magánhangzók és más speciális karakterek kódja következik. Ez a tartomány nem egységes, függ az operációs rendszer területi beállításaitól. A karakterek ANSI-kódját a számítógépen érvényes beállítás mellett a könyvhöz mellékelt CD Dokumentumok mappájában elhelyezkedő ANSI-kódok.htm fájl megnyitásával tanulmányozhatjuk.

256-féle lehetőség nem elég a különböző nyelvek karaktereinek tárolására. A Unicode (uniform code, egységes kód) rendszerben két bájtot, azaz 16 bitet használunk egy karakter kódolásához. Így 65536-féle jelet tudunk ábrázolni, ami bőven elegendő a föld jelenlegi és múltbeli nyelveinek ábécéjéhez.

A Windowsban a Jegyzetkönyvvel készült fájlok mentésénél meghatározhatjuk, hogy melyik kódolást alkalmazzuk.

Ha egy állomány csak a karakterek kódját tartalmazza, akkor gyakran ASCII- vagy szövegfájlnak hívják. A Jegyzetkönyvvel például ilyen fájlokat kezelünk. A Jegyzetkönyvvel készített állományokban nem tudjuk tárolni a formátumot (félkövér kiemelés, betűtípus, szín stb.).

A Word dokumentumok nem szövegfájlok, a formátumra vonatkozó speciális kódokat is tartalmaznak. A weblapok fájljai viszont szövegfájlok, mert a formátumra vonatkozó előírásokat is szöveggént, nem pedig speciális kódokkal tárolják. Ha egy weblapot a Jegyzetkönyvvel nyitunk meg, akkor olvasható, bár számunkra még nem értelmezhető kódot látunk.

### Input és output

Akár szövegeket, akár más információt tárolunk, először be kell írunk azt a számítógépbe. A bevitt idegen szóval inputnak hívjuk. Az input végrehajtható a billentyűzeten keresztül, az egér segítségével, vagy előzetes tárolás után egy háttértárról történő olvasással.

Az adatok, eredmények kivitelét outputnak nevezzük. Az output történhet a képernyőre, nyomtatóra vagy valamilyen háttértárra. Számítógépes hálózat használatánál az input vagy az output irányulhat a hálózat valamelyik eszközére is.

Input:	az adatok beolvasása a billentyűzetről, a háttértárról vagy valamilyen más hardvereszköztől, illetve a hálózatról.
Output:	az eredmények megjelenítése a képernyőn, kiküldése valamilyen hardvereszköztől, hálózatra vagy mentése egy háttértárra.

A felhasználó számára közvetlenül hozzáférhető input- és outputeszközöket konzolnak hívjuk. Egy személyi számítógép esetén a billentyűzet, az egér, a képernyő alkotja a konzolt. A programok vezérlése (beleértve az indítást és a leállítást) is a konzolon keresztül történhet.

A személyi számítógépek operációs rendszerei a képernyőn már nem csak szöveget, hanem ikonokat és egyéb grafikus elemeket, úgynevezett grafikus felhasználói felületet szolgáltatnak. A grafikus felhasználói felület (angol rövidítése: GUI) sokkal

kényelmesebbé teszi a számítógép kezelését, mint a régebbi, karakteres felületek, melyeknél általában angol nyelvű parancsok begépelésével vezérelhettük a működést.

## Programok és programozási nyelvek

A számítógéppel különböző feladatokat oldunk meg. A megoldás módszerét algoritmusnak nevezzük.

Algoritmus: egy feladat megoldásának lépésenkénti leírása.

Az algoritmustól elvárjuk, hogy egyértelmű legyen, és véges sok lépésben vezessen el a célhoz. Az algoritmus független a végrehajtáshoz felhasznált számítógéptől.

Az algoritmusok konkrét megvalósítása a számítógépes program.

Számítógépes program: a számítógép által értelmezhető utasítások sorozata.

A program utasításait programozási nyelven fogalmazzuk meg. A programozási nyelvek szolgáltatják számunkra a program megírásához szükséges utasításokat. A programozáshoz sokféle programnyelv közül választhatunk. Gyakran találkozunk Pascal, Java, Visual Basic, C++ vagy más nyelvű programokkal.

Egy programnyelvben az utasításokat szigorú formai szabályok betartásával írjuk le. Ezek a formai szabályok alkotják a programnyelv helyesírását, az úgynevezett szintaxist.

Szintaxis: a programozási nyelv formai szabályainak összessége.

A Java programozási nyelvben például minden utasítás után pontosvesszőt kell tennünk. Ez a szintaxis egy szabálya. Ha elfeledkezünk róla, akkor szintaktikus hibát vétünk.

Ahhoz, hogy a programot a mikroprocesszor végre tudja hajtani, bináris kóddá kell átalakítanunk. A programok bináris kódját gépi kódnak hívjuk. A gépi kód kötődik a mikroprocesszorhoz. Ugyanaz a bitsorozat a különböző mikroprocesszorok esetén más és más utasítást jelenthet.

Egy Intel mikroprocesszornál például a

```
10110000 00000011
```

gépi kódú utasítás hatására az AL-lel jelölt bájt tartalma 3 lesz.

A gépi kódú programot be lehet tölteni a számítógép memóriájába, ahonnan a mikroprocesszor egyesével kiolvassa és végrehajtja az utasításokat. Ezt a folyamatot a program futtatásának nevezzük.

Gépi kódban nehéz programozni, ezért olyan programozási nyelveket használunk, melyekben az utasítások könnyebben olvashatók és megjegyezhetők.

A gépi kódhoz legközelebb az assembly nyelvek állnak. Egy assembly program utasításai általában megfelelnek a gépi kód utasításainak, de bitek helyett angol szavakat vagy rövidítéseket használunk. A fenti gépi kód assembly megfelelője:

MOV AL, 3      (írj be az AL-be 3-at)

Ezt azonban nem érti meg a mikroprocesszor, ezért az assembly nyelven megírt programot le kell fordítani gépi kódra. A fordítást rábízhatjuk egy programra, a fordítóprogramra. Az assembly programok fordítóprogramját assemblernek nevezzük.

Assemblyben nehéz programozni, mert logikája és jelölésrendszere a mikroprocesszorhoz illeszkedik, nem pedig a mi gondolkodásmódunkhoz. A magas szintű programnyelvek már közel állnak az emberi beszédhez. Az utasítások szinte értelmes mondatokat alkotnak:

AL = 3      (AL legyen 3) vagy

If Speed > 90 Then Fine = True

(Ha *Sebesség* nagyobb mint 90, Akkor *Büntetés* legyen igaz)

Ez utóbbi sor szó szerint megfelel egy angol mondatnak.

### A programnyelvek generációi

Az informatika története során számos programozási nyelv született. Egyre fejlettebb változataik alakultak ki, gyakran egy feladattípushoz készült egy újabb programnyelv. A különböző típusú feladatok megoldásához más és más programnyelven lehet hatékony programot írni.

A programozási nyelveket generációkba sorolhatjuk. Az első generációhoz a fent említett gépi kódú nyelvek, a második generációhoz pedig az assembly nyelvek tartoztak.

A harmadik generációtól kezdve magas szintű programnyelvekről beszélünk. Első képviselőik közé sorolhatjuk a COBOL-t, a Fortrant, a Basicet. A további fejlődés során alakultak ki azok a strukturált nyelvek, melyek a programfejlesztés technikájára is hatással voltak. Az Algol, a PL1, majd a Pascal nem csak egy újabb programnyelvet jelentett, hanem lehetővé tette az egyszerű, világos és áttekinthető programszerkezet kialakítását.

Az objektumok bevezetése kibővítette a programozási eszközöket, és megoldotta azt a krízist, amit az egyre bonyolultabbá váló rendszerek programozása, kezelése okozott. A SmallTalk, majd később a C++, az Object Pascal, a Java részben vagy egészben az objektumok használatára épült.

Az objektum-orientált programozási technikán alapuló negyedik generációs nyelvek (4GL, 4. generation languages) a programfejlesztésben is új módszereket hoztak. A program vázát egy vizuális fejlesztői környezetben készítjük el. Az utasítások begépelése helyett eszköztárakból tesszük a helyükre a program elemeit, menükből állítjuk be tulajdonságaikat. Ezek az elemek akár hosszú utasítássorozatok helyettesíthetnek. A Delphi, a Visual C++, a Visual Basic és más vizuális fejlesztőeszközök nagymértékben leegyszerűsítik a programírást.

Az utasításokat és vezérlési szerkezeteket alkalmazó, úgynevezett eljárásalapú nyelvek mellett más logikai eszközökre épülő programozási nyelvek is léteznek. A lekérdő nyelveken például viszonylag kevés programozói ismerettel, egyszerűen véghezjuthat az adatbázisok kezelése. A LISP vagy a PROLOG a mesterséges intelligen-

cia kutatások során alakult ki. A legtöbb szakértői rendszer elkészítéséhez a LISP-et használták fel. A PROLOG-ban utasítások helyett szabályokat adunk meg, és egy állítás igaz vagy hamis voltát szeretnénk eldönteni. Gyakran a nem eljárásalapú nyelveket tekintik negyedik, a mesterséges intelligencia kutatásával kapcsolatos nyelveket pedig ötödik generációnak.

### **Fordítóprogramok**

Az algoritmus megvalósításához ki kell választanunk egy programnyelvet, és át kell írunk a lépéseket utasítások sorozatává. Ez alkotja a program forráskódját.

Forráskód (kód): egy programozási nyelven megírt utasítássorozat.

A forráskód a programozási nyelvhez kötődik. Általában független attól a géptől, amelyen majd a program fut, és többé-kevésbé szabványos elemeket használ fel.

A forráskódot szövegszerkesztővel készítjük, és szövegfájl formájában tároljuk. A programok javítása, módosítása is a forráskódban történik. A forráskóddal szemben a gépi kódot szokás tárgykódnak nevezni.

A forráskódot fordítóprogram (compiler) alakítja át gépi kóddá. Szintaktikus hiba esetén a fordítóprogram hibaüzenetet ad, és nem készül gépi kód.

A magas szintű programnyelvek fordítóprogramjai egy-egy utasításból már eléggé összetett gépi kódot állítanak elő. A fordítás folyamata általában több lépésben zajlik. Végrehajtásra a lefordított gépi kód kerül, amit fájlokban tárolunk. Az újabb futtatáshoz nincs szükség ismételt fordításra.

Egy programnyelvhez többféle fordítóprogram is létezik. Ezek hatékonyságban, árban, a programok fejlesztéséhez nyújtott szolgáltatásokban különböznek egymástól.

A fordítóprogram gyakran egy nagyobb programcsomag, a fejlesztői rendszer része. Az integrált fejlesztői rendszer (IDE) a fordítóprogramon kívül a programnyelv sajátosságaihoz illeszkedő szövegszerkesztőt, hibakeresési eszközöket, a forráskód megírását megkönnyítő komponenseket is tartalmaz.

Megjegyezzük, hogy a Java programozási nyelvénél a fordítóprogram a forráskódból egy közbenső, úgynevezett bájtkódot hoz létre. A bájtkód már gépközel kód, de még nem függ attól, hogy milyen számítógépes környezetben fut majd a program. A bájtkódot a futtatás előtt (vagy közben) egy másik program alakítja át gépi kóddá. A gépi kód előállítására esetenként más megoldásokat is alkalmaznak, melyekre itt nem térünk ki.

### **A szkript nyelvek**

Több programozási nyelv esetén a program nem kerül lefordításra a futtatás előtt. Helyette egy értelmező program (interpreter) veszi sorra, és hajtja végre az utasításokat. Ezeket a nyelveket szkript-nyelveknek nevezzük.

Szkript: olyan utasítássorozat, amelyet egy másik program hajt végre.

Interpreter: a szkriptek utasításait értelmező és végrehajtó program.

A szkript nyelvek első képviselője a BASIC volt, melyet 1964-ben fejlesztett ki Kemény János és Thomas Kurtz.

Az interpreterek futás közben értelmezik a forráskódot. Elővesznek egy utasítást, majd végrehajtják. Az ismétlődő utasításokat minden egyes alkalommal értelmezni kell. Az interpreterek használata esetén jóval lassúbb a program végrehajtása, az értelmezés jelentős részét teszi ki a futási időnek. Interpreterekkel azonban egyszerűbb az utasítások kipróbálása. Nem szükséges a teljes program elkészítése, akár néhány utasítás is végrehajtható.

Több programnyelv, például a Visual Basic fordítóprogrammal és értelmezővel is rendelkezik. A könyvünkben szereplő Visual Basic Scriptnek csak interpretere van. Ez az Interneten történő felhasználás sajátosságaival függ össze.

Szkripteket a számítógép működését felügyelő operációs rendszer vezérléséhez is használnak. A többször végrehajtásra kerülő parancsokat szövegfájlba írhatjuk, majd utasíthatjuk az operációs rendszert a végrehajtásra. A parancsainkat tartalmazó állományt parancsfájlnak nevezzük. A Unix operációs rendszerben, illetve a Windows kiszolgálókon fontos szerepet töltenek be ezek a parancsfájlok.

A szkript nyelveket gyakran parancsnyelveknek hívják. Mára azonban meglehetősen eltávolodtak az operációs rendszereket vezérlő parancsfájloktól. Teljes értékű programnyelvnek tekinthetők, melyek segítségével sokféle programozási feladatot megoldhatunk.

A szkriptek egyik fontos alkalmazási területe a weblapok programozása. A weblapok kódjába szkripteket lehet beilleszteni. Ezeket a böngészőbe épített interpreter hajtja végre. Az Interneten használt szkript nyelvek közül a Java Script és a Visual Basic Script a legismertebb.

Könyvünkben a Visual Basic Script<sup>1</sup> segítségével vezetjük be az Olvasót a programozás rejtelmeibe.

## 1.2. Objektumok és események

A modern programozási nyelvek objektumokkal dolgoznak, és eseményeket kezelnek. Ezek a fogalmak a Visual Basic Scriptben is alapvető szerepet játszanak, így meg kell velük ismerkednünk.

### Objektumok és objektumosztályok

A körülöttünk lévő világ különböző dolgokból áll. Az ablakunk előtt egy fa leveleit mozgatja a szél, az úton éppen egy kutya szalad át, holnap reggel pedig iskolába vagy a munkahelyünkre igyekszünk, ahol munkatársainkkal találkozunk. A szomszédék kutyáját Bodrinak hívják, a Petőfi Sándor Általános Iskola elvégzése után a Dobó Gimnáziumban érettségiztem, Kovács Pista íróasztala a munkahelyemen az enyém mellett helyezkedik el.

---

<sup>1</sup> Microsoft Visual Basic Scripting Edition.

Az informatikában ezeket a konkrét dolgokat objektumoknak nevezzük. Az objektumalapú szemléletmód nagyon hasznos segédeszköz lesz a számunkra. A továbbiakban Bodri kutya, a Petőfi Sándor Általános Iskola, Kovács Pista mind egy-egy objektumot fog jelenteni.

Környezetünk sokféle alkotóeleméből általánosítással fogalmakat, osztályokat alkotunk. Az osztályokba sorolás a gondolkodás és a nyelv szerves része. Az egymáshoz hasonló dolgok ugyanahhoz az osztályhoz tartoznak. Bodri és társai kutyák, a Dobó Gimnázium iskola, Kovács Pista pedig munkatárs. A kutya, az iskola, a munkatárs egy-egy fogalom. A fogalmak sok konkrét formában jelenhetnek meg. A fogalmak, mint osztályok megjelenési formái az objektumok.

Osztály: az egymáshoz hasonló dolgok összessége.
Objektum: az osztály egy konkrét egyede, példánya.

A hétköznapi életben ugyanaz az objektum sokféle osztályhoz tartozhat. Bodri kutya egyben emlős és élőlény is. Ugyanaz a személy lehet munkatársunk, szomszédunk, barátunk. A programozási nyelvekben már egyértelműen meghatározható, hogy az objektumok pontosan mely osztályokhoz tartoznak. Több programnyelv, így a Visual Basic Script esetén pedig egy objektum csak egyetlen osztálynak az egyede.

Ha bekapcsoljuk a számítógépet, akkor a monitoron objektumokat látunk. Az Asztalon megjelenik a Sajtógép ikonja, ha elindítjuk az Intézőt, láthatjuk a rendelkezésünkre álló meghajtókat, mappákat, fájlokat. Az asztal, a sajtógép, a mappa és a fájl objektumosztályok. Konkrét egyedeik az Asztal, a Sajtógép, a Windows mappa vagy a benne lévő fájlok pedig objektumok (a fájlok nem a mappa osztály egyedei!). A mappa- vagy a fájlosztálynak sok objektuma lehet, az asztal vagy a sajtógép osztályt egyetlen objektum képviseli.

Az Internetről letöltött weblapok szintén objektumokból állnak. A szöveg egyes bekezdései a bekezdésoosztályhoz (P, paragraph-objektumok), a lapon látható képek pedig a képosztályhoz (IMG, image-objektumok) tartoznak. Maga az egész weblap is egy objektum, a dokumentumosztály egyetlen objektuma (document), a megjelenítést végző böngésző ablaka pedig az ablakosztály egyetlen objektuma (window).

Az eddigi példákat a következő oldalon lévő táblázatban foglaltuk össze.

### **Tulajdonságok, metódusok, események**

Az objektumok sokféle tulajdonsággal rendelkezhetnek. Egy kutyának van neve, fajtája, színe. Munkatársainknak szintén van neve, magassága, lakcíme és még számos további tulajdonsága. A mappáknál lekérdezhető a létrehozási ideje, mérete. Egy weblapon lévő képnek megadható a mérete, a keret színe, vastagsága, a képet tartalmazó fájl neve, elérési útja. A programokban felhasználjuk az objektumok tulajdonságait, és módosítani tudjuk az értéküket.

Az osztályok objektumai számos tevékenységet el tudnak végezni. Egy kutya eszik, iszik, alszik, ugat. A számítógép egy meghajtója formázni tudja a háttértárat, adatokat ír, olvas. Az objektumok által végrehajtható tennivalókat metódusoknak nevezzük. A későbbiekben meg fogjuk látni, hogy függvények és eljárások egyaránt lehetnek.

Világ		Windows		Weblap	
<i>Osztály</i>	<i>Objektum</i>	<i>Osztály</i>	<i>Objektum</i>	<i>Osztály</i>	<i>Objektum</i>
kutya	Bodri, Kormos, ...	mappa	Dokumentumok, Windows mappa, ...	bekezdés	1. bekezdés 2. bekezdés, ...
iskola	Dobó Gimnázium, ...	meghajtó	c:, d:, ...	kép	1. kép, 2. kép, ...
munkatárs	Kovács Pista, ...	asztal	Asztal	dokumentum	document
szomszéd	Vass Lajos, ...	sajátgép	Sajátgép	ablak	window

1–1. táblázat Osztályok és objektumok

<i>Osztály</i>	<i>Objektum</i>	<i>Tulajdonság</i>	<i>Esemény</i>	<i>Metódus</i>
kutya	Bodri	név, szín, fajta	füttyszó: vendég jön:	a gazdájához fut, ugat
meghajtó	c:	betűjel, típus, kapacitás	Mentés parancs: kikapcsolás:	ír, parkol
Windows- ablak	Intéző-ablak	méret, hely, az állapotsor szövege	kattintás a Bezárás gombra: a keret mozgatása:	bezárul, méretváltoztatás

1–2. táblázat. Tulajdonságok, események, eseménykezelő metódusok



Az objektumok bizonyos tennivalókat a felhasználók kérésére végeznek el (lenyomunk egy billentyűt, kattintunk az egérrel), vagy más esemény bekövetkezése váltja ki a tevékenység végrehajtását. Az objektumok ezekre az eseményekre reagálnak, eseményeket kezelnek. Bodri füttyszóra a gazdájához fut, visszahozza az elhajított labdát. A Windowsban egy ablak a Bezárás gombra kattintva bezárul, a keret mozgatásakor megváltoztatja a méretét. Az eseménykezelés a leggyakrabban valamilyen metódus végrehajtását jelenti a felhasználó (vagy egy másik objektum) kérésére.

## A programok vezérlése

A programok adatokkal dolgoznak, a beolvasás után számításokat végeznek, majd közlik az eredményeket. A bemenő adatok általában valamilyen háttértáron helyezkednek el, vagy a felhasználó a program futása során adja meg őket. Az is előfordulhat, hogy a kiindulási értékek a program kódjában szerepelnek.

Köteget (batch) feldolgozás: az adatok valamilyen háttértáron vagy a program kódjában találhatók. A felhasználó az indítás után általában nem tud beleavatkozni a feldolgozás folyamatába.

Interaktív feldolgozás: a program működése közben kommunikál a felhasználóval, folyamatosan lehetőség van a vezérlésre, az adatok beírására, módosítására.

A továbbiakban először interaktív programokat fogunk írni. Az interaktív programokat további két nagy csoportra bonthatjuk.

Algoritmus-vezérelt program: a kommunikációt a program irányítja. Beolvassa az adatokat, elvégzi a számításokat, majd közli az eredményeket és befejezi a működést. Csak akkor van lehetőségünk a vezérlésre, ha a program erre külön rákérdez, vagy újra elindítjuk.

Eseményvezérelt program: a kommunikációt a felhasználó irányítja. Közli a kívánságait a programmal, tetszőlegesen módosíthatja az adatokat. A program futása is a felhasználó döntésére fejeződik be.

Az *algoritmus-vezérelt program* futtatása egy folyamat: a beolvasás, feldolgozás, eredményközlés folyamata. Az *eseményvezérelt program* működése inkább egy állapot, általában eseményekre (például a felhasználó kéréseire) vár, majd a feldolgozás és eredményközlés után megint várakozó állapotba kerül.



## 2. BEVEZETÉS A HTML-BE

A Visual Basic Script programokat a weblapok kódjába illesztjük, és egy böngésző segítségével futtatjuk. Ezért először meg kell ismerkednünk a weblapok felépítésével, a leírásukhoz használt HTML-kóddal.

### 2.1. A weblapok felépítése és kódolása

#### A HTML-kód

A weblapok szerkezetének, megjelenítésének leírásához a HTML-kódot alkalmazzák. A HTML a Hypertext Markup Language (hipertext leírónyelv) rövidítése. A HTML-kódot szövegfájlban tároljuk, amely tetszőleges szövegszerkesztővel, például a Jegyzettömbbel is olvasható, szerkeszthető, módosítható.

Egy weblap a HTML-kód által leírt objektumokból áll. Az objektumok tartalmazhatják egymást, egy bekezdésben például elhelyezhetünk egy ábrát, a táblázat objektumnak vannak cellái, a cellákban lehet szöveg vagy kép.

Megjegyezzük, hogy ha csak a weblapok szerkesztése, összeállítása a cél, akkor az objektumok helyett gyakran elemekről beszélnek. A programok írásánál azonban mi az objektumalapú szemléletmódot tartjuk fontosnak, így elemek helyett mindvégig objektumokat említünk.

#### Objektumok leírása a HTML-kódban

A HTML-kódban az objektumok definícióját egy nyitó és egy záró tag<sup>2</sup> határolja, melyek az objektumok tartalmát fogják közre. A nyitó tag csúcsos zárójelek (kisebb, nagyobb jelek) között megadja az objektum osztályának megnevezését, azonosítóját és egyéb tulajdonságait. A záró tag a csúcsos zárójeleken belül egy /-jel után megismétli az objektum osztályának a nevét.

Egy objektum további objektumokat tartalmazhat. Ezek kódja a nyitó és a záró tag közé kerül. Gyakran ide írjuk az objektum által megjelenített szöveget is.

Az egymást tartalmazó objektumok esetén nem hozhatunk létre átlapolást, azaz a HTML-kódban legelőször a legutoljára megnyitott objektumot kell lezárni:

```
<1. objektum nyitó tag>
  <2. objektum nyitó tag>
    <3. objektum nyitó tag>
      <3. objektum záró tag>
    <2. objektum záró tag>
  <4. objektum nyitó tag>
    <4. objektum záró tag>
<1. objektum záró tag>
```

---

<sup>2</sup> A tag szó az angol szakirodalomból származik, és eredetileg címkét, jelzőt jelentett. Mivel magyarul is jól kifejezi a határoló elemek szerepét, ezért teljesen elterjedt a magyaros használata.

A fenti példában az első objektum tartalmazza a 2. és 4. objektumot, míg a 2. objektum tartalmazza a 3. objektumot. Az áttekintést megkönnyíti a forráskód strukturált elrendezése.

Strukturált elrendezés: az azonos szinthez tartozó elemeket egy oszlopban, egymás alá írjuk, az általuk tartalmazott elemeket pedig beljebb kezdjük a forráskódban.

A továbbiakban mind a HTML-kód, mind a programjaink esetén törekedni fogunk a strukturált elrendezésre.

Néhány speciális objektum nem tartalmaz további elemeket (objektumokat vagy szöveget). Ezeket üres objektumoknak nevezzük. Az üres objektumoknak csak nyitó tagjuk van.

### Az objektumok azonosítása

A weblap objektumait névvel, úgynevezett azonosítóval láthatjuk el. Az azonosító az objektum egy tulajdonsága. A tulajdonság neve: *id* (identifier, azonosító). Azonosító segítségével könnyen hivatkozhatunk az objektumra. Nem kötelező azonosítót alkalmazni, mert a böngésző 0-val kezdve sorszámozza a weblap objektumait. Ezekkel a sorszámokkal azonban nehezebb a hivatkozás.

Az azonosítót a nyitó tagban az *id* tulajdonságnevet követő egyenlőségjel után, idézőjelek között adjuk meg:

```
id = "Azonosító"
```

Az objektumok azonosítói az angol ábécé betűiből, számokból és aláhúzásjelekből állhatnak, de betűvel kell kezdődniük. Speciális esetektől eltekintve nem ajánlatos különböző objektumoknak ugyanazt az azonosítót adni, mert így megnehezítjük a rájuk való hivatkozást. Célszerű az objektumra jellemző azonosítót választani, hogy könnyebben áttekinthető legyen a kód.

Szabályos azonosítók:

```
abra, gomb1, gomb2, Kutya, Bodri_kutya, BodriKutya
```

Hibás azonosítók:

ábra	nem szerepelhet benne ékezetes magánhangzó,
2gomb	nem kezdődhet számmal,
gomb.2	nem szerepelhet benne pont (és az aláhúzásjelet kivéve más írásjel).

A HTML nyelv nem különbözteti meg a kis- és nagybetűket, így a következő azonosítók mind ugyanazt az objektumot jelentik:

```
Bodri, bodri, BODRI, boDRi
```

A Visual Basic Script sem tesz különbséget a fenti karaktersorozatok között. A kód áttekintését, a hibakeresést azonban megkönnyíti, ha mindig ugyanabban a formában írjuk az azonosítókat.

Az aláhúzásjelet, a nagybetűket felhasználhatjuk a hosszabb nevek olvashatóvá tételére (szóközt nem írhatunk az azonosítókba):

```
JobbGomb, jobb_gomb, BalGomb, bal_gomb
```

## A weblapok alapobjektumai

A weblapokat a böngésző ablakában jelenítjük meg. Az ablakobjektum az ablak objektumosztály egyetlen példánya.<sup>3</sup> Azonosítója kötelezően: *window*. Tulajdonságai közé tartozik a mérete, a helye, az állapotsor szövege. Metódusai lehetővé teszik az átméretezését, a mozgatását, a bezárását. Használatára később látunk példákat.

A weblap elemeit a dokumentum osztály egyetlen objektuma tartalmazza. Azonosítója kötelezően: *document*. A *document*-objektum tulajdonságai közé tartozik például a háttér és a betűk színe, az ablak címsorának a felirata. *Write* metódusának segítségével jeleníthetünk meg programjainkból szöveget a weblapon. Többféle esemény kezeléséhez is a *document*-objektumot használjuk.

A *window* és a *document*-objektum definíciója közvetlenül nem szerepel a HTML-kódban, de a böngésző a Weblap megnyitásakor automatikusan létrehozza őket.

## A HTML-objektum

A HTML-kód a HTML-objektum leírásával kezdődik. Ha nem adunk meg azonosítót és semmilyen más tulajdonságot, akkor definíciója a **2–1. példában** látható:

```
<HTML>  
</HTML>
```

Ha ezt a kódot begépeljük a Jegyzetömbbe, és .htm kiterjesztéssel elmentjük, akkor a fájlt megnyithatjuk egy böngészővel. Megnyitásakor a böngésző létrehoz egy *window*, egy *document* és egy HTML-objektumot, illetve megjeleníti a teljesen üres weblapot. A HTML-objektumra nem szoktunk hivatkozni, így általában nem kap azonosítót, és nem használjuk egyéb tulajdonságait sem.

## A fej- és a törzsobjektumok

A HTML-objektum tartalmazza a fej- és törzsobjektumot. A fejobjektum a HEAD osztály, a törzsobjektum pedig a BODY osztály egyetlen példánya a dokumentumban.

A HEAD által tartalmazott objektumok általában a weblappal kapcsolatos információkat tárolják (karakterkészlet, kulcsszavak stb.). A törzsobjektum tartalmazza magát a képernyőn megjelenített weblapot.

HTML-kódunk a fenti 3 objektum definíciójával a **2–2. példában** látható:

```
<HTML>  
  <HEAD>  
  </HEAD>  
  <BODY>  
  </BODY>  
</HTML>
```

---

<sup>3</sup> Keretek alkalmazása esetén több ablakobjektum is létrejön, de ezzel könyvünkben nem foglalkozunk.

Figyeljük meg a tagolt (strukturált) elrendezést! Az egymáshoz tartozó nyitó és záró tagok egymás alatt, egy oszlopban helyezkednek el, míg az egyik objektum (itt a HTML) által tartalmazott objektumok tagjait beljebb kezdjük. Ez az elrendezés nem kötelező a HTML-kódban, de nagymértékben megkönnyíti az áttekintést, ezért a továbbiakban következetesen alkalmazzuk.

Ha megnyitjuk egy böngészővel a fájlt, akkor létrejön a *window*, a *document*, a HTML-, a fej- és a törzsobjektum. A böngésző megjeleníti az egyelőre üres weblapot.

### A címobjektum

Bár a weblap ablakának a címe a *document*-objektum egy tulajdonsága, a többi tulajdonságtól eltérően a címobjektum segítségével is megadhatjuk. A címobjektum a TITLE (cím) osztály egyetlen példánya. A címobjektumot a HEAD-ben helyezzük el:

```
<HEAD>
  <TITLE>
    Az ablak címsorának megadása
  </TITLE>
</HEAD>
```

A nyitó és záró tag közé eső tartalma (szövege) megjelenik az ablak címsorában. A **2–3. példa** betöltésekor figyeljük meg a hatását!

A címobjektum csak a felhasználó tájékoztatását szolgálja, ezért általában nem kap azonosítót. Példáinkban mindig alkalmazzuk a címobjektumot az ablak címsorának a megadásához.

Megjegyezzük hogy, a böngésző akkor is létrehozza a HTML-, a fej- és a címobjektumot, ha nem szerepelnek a kódban.

### A META-objektumok

Mint említettük, a fejobjektumban különböző információkat tárolhatunk. Erre a META objektumosztály egyedeinek tulajdonságait használjuk. A továbbiakban a tömörebb fogalmazás érdekében a META objektumosztály egyede helyett egyszerűen META-objektumot írunk. Ezt a szóhasználatot alkalmazzuk a weblap többi objektumosztályával kapcsolatban is.

A META-objektumok csak nyitó taggal rendelkező, üres objektumok. A tulajdonságok megadása eltér a többi objektumnál alkalmazott szintaxistól. A tulajdonság nevét a *name* kulcsszó után, értékét pedig a *content* kulcsszó után egyenlőségjellel, idézőjelekben adjuk meg:

```
<META name = "tulajdonságnév" content = "a tulajdonság értéke">
```

Egy META-objektum definíciójában csak egy név-érték pár szerepelhet. A **2–4. példában** beillesztjük a HTML-kódba a szerzők nevét és a könyv címét:

```
<HEAD>
  <META name = "szerző" content = "Juhász Tibor - Kiss Zsolt">
  <META name = "cím" content = "Tanuljunk programozni">
</HEAD>
```

A fejobjektum definíciójában a TITLE- és META-objektumok sorrendje tetszőleges lehet. A META-objektumok tartalma nem jelenik meg a weblapon, csak a kódban olvasható.

A META-objektum egy fontos alkalmazása a weblap karakterkészletének a megadása. Ha a magyar ékezetes karakterek nem megfelelően jelennek meg a képernyőn, akkor a következő META-objektumot kell beilleszteni a HEAD-be:

```
<META http-equiv = "Content-Type"
      content = "text/html; charset = Windows-1250">
```

A begépelésnél figyeljünk a pontos szintaxisra! A kód részleteivel nem foglalkozunk, de megjegyezzük, hogy itt a *name* kulcsszó helyett *http-equiv* áll. Ez a böngésző számára azt jelenti, hogy nem egy a felhasználó által definiált tulajdonságot adtunk meg, hanem a META-objektum a weblap megjelenítéséhez tartalmaz információt.

## A törzsobjektum tulajdonságai

A fentiekben ismertetett objektumokat a dokumentum betöltésekor létrehozza a böngésző, de nem szoktunk rájuk hivatkozni. Ezért általában nem kapnak azonosítót. A törzsobjektumra azonban néha szükségünk lesz. Az azonosítót a többi tulajdonsággal együtt az objektum nyitó tagjában helyezzük el. Megadjuk a tulajdonság nevét (*id*), majd egy egyenlőségjel után az értékét. Az érték valamilyen szöveg (karakteresorozat), ezért idézőjelbe tesszük. Megjegyezzük, hogy az Internet Explorer az *id* megadásánál nem igényli az idézőjel használatát, de célszerű a többi tulajdonsághoz hasonlóan itt is alkalmazni.

A **2–5. példában** a BODY-objektumot a *Törzs* azonosítóval látjuk el:

```
<BODY id = "Törzs">
```

Emlékeztetünk arra a szabályra, hogy az objektumok azonosítói nem tartalmazhatnak magyar ékezetes magánhangzókat.

A törzsobjektum segítségével már változatosabbá tehetjük weblapjainkat. A háttérszint például a *bgColor* (background color, háttérszín) tulajdonsággal adjuk meg. Értékét a böngésző által ismert színek angol megnevezései közül választhatjuk, vagy egy speciális, hexadecimális kóddal írjuk le. A **2–6. példa** pirosra állítja a háttérszint:

```
<BODY id = "Törzs" bgColor = "red">
```

A színek angol elnevezéseit és a megfelelő kódot a mellékelt CD Dokumentumok mappájában találjuk meg.

A weblap több tulajdonsága a BODY és a *document*-objektum segítségével egyaránt elérhető.

## Hibák a HTML-kódban

Ha megpróbálkoztunk az eddigi példák begépelésével, akkor könnyen előfordulhatott, hogy valamilyen hibát vétettünk. Kimaradt egy < jel, rosszul írtuk be a *bgColor* tulajdonságnevet, lemaradt valamelyik idézőjel vége. A böngészők elég rugalmasan kezelik a HTML-kód hibáit, lehetőség szerint megjelenítik a dokumentumot. A legrosszabb esetet az jelenti, ha le hagyjuk a tagok végéről a > jelet vagy a záró idézőjelet,

mert ekkor a kód folytatását is beleértik a tagba vagy az idézőjelek közé. Ha rosszul gépelünk be egy tulajdonságnevet, akkor az a tulajdonság nem kap értéket, de más hatása nem lesz a weblap megjelenítésére.

Amikor nem a várakozásunknak megfelelően jelenik meg a dokumentum, akkor mindig figyelmesen olvassuk át a kódot. A hibakeresést segítik azok a szerkesztőprogramok, melyek különböző színekkel jelölik a tagokat, a tulajdonságokat és a weblap egyéb elemeit. Ezek közé tartozik a Microsoft FrontPage Express, a Macromedia DreamWeaver, az UltraEdit vagy sok más, az Internetről letölthető program.

Egy HTML-szerkesztőprogram használatát mindenképpen javasoljuk az Olvasónak. A CD-melléklet TextPad mappájában található TextPad program telepítését és használatát a függelékben ismertetjük.

## 2.2. A weblap szövege

Eddigi példáinkban színes, de üres weblapokat láttunk. Az alábbiakban áttekintjük, hogyan lehet szöveget megjeleníteni a képernyőn.

### Szövegek megjelenítése

Bár a weblap szövege a BODY-objektum egy tulajdonságának értéke, terjedelmére való tekintettel nem a nyitó tagban, hanem a nyitó és a záró tag között adjuk meg, csakúgy, mint a törzs többi objektumát. Programjainkban azonban majd szabályosan, az objektumok tulajdonságaként fogjuk elérni és módosítani.

A **2–7. példa** bemutatja, hogy a `<BODY>` és `</BODY>` tagok közé írt bármilyen szöveg megjelenik a weblapon:

```
<BODY>
  Ez az első szöveg.
</BODY>
```

Néhány speciális karakternek nincs hatása a szöveg elrendezésére, a böngésző egyetlen szóközzel helyettesíti őket. Ezeket tagoló karaktereknek<sup>4</sup> nevezzük. Hozzájuk tartozik az Enter és a tabulátor. Ha több szóközt írunk egymás után a HTML-kódba, akkor is csak egy kerül rá a weblapra.

A **2–8. példa** a több szóköz és az új sor ellenére a szöveg az előző példával teljesen megegyező formában jelenik meg a képernyőn:

```
<BODY>
  Ez      az első
  szöveg.
</BODY>
```

Tagoló karaktereket a tagokon belül is használhatunk, a `< BODY >` vagy a

```
<
  BODY
>
```

---

<sup>4</sup> Angolul: white space.



kód a megjelenítés szempontjából egyenértékű a `<BODY>`-val, de minden szóköz vagy Enter egy-egy bájtal megnöveli a fájl méretét.

A tagoló karaktereket felhasználhatjuk a HTML-kód strukturált elrendezésére.

## Karakterentitások

Ha több szóközt szeretnénk egymás mellett elhelyezni, akkor a HTML-kódba a nem törhető szóköz rövidítését (`&nbsp;`, non breaking space) vagy Unicode-értékét (`#160`) kell beírni egy `&` jel és egy pontosvessző közé. Ezt a **2–9. példa** mutatja be:

```
<BODY>
  Ez &nbsp; &nbsp; &#160; &#160; az első
  szöveg.
</BODY>
```

Figyeljük meg, hogy a szóközők már megjelennek a weblapon, de a teljes szöveg most is egy sorba került! A tördelést további objektumok segítségével végezzük.

A weblapok kialakításánál gyakran használunk a nem törhető szóközhöz hasonló rövidítéseket vagy kódokat, úgynevezett karakterentitásokat.

Karakterentitás:  
a karaktereknek megfelelő HTML-kód vagy hexadecimális érték.

A `<` és `>` jelek például speciális jelentésük miatt nem szerepelhetnek a weblapon, a böngésző hibásan jelenítené meg a szöveget. Helyettük a nekik megfelelő karakterentitást kell használni:

<code>&lt;</code>	<code>&amp;lt;</code>	(less than)	vagy <code>&amp;#60;</code>
<code>&gt;</code>	<code>&amp;gt;</code>	(greater than)	vagy <code>&amp;#62;</code>

Töltsük be a böngészőbe a **2–10. példát**, és figyeljük meg az eredményt!

```
<BODY>
  Különleges karakterek használata.
  Helyes: A&lt;B, hibás: A<B
</BODY>
```

Karakterentitásokat speciális karakterek vagy a magyar ékezetes magánhangzók helyett használunk, hogy a különböző platformokon is helyesen jelenjen meg a szöveg. A HTML-kódba írható karakterentitások felsorolását megtaláljuk a mellékelt CD Dokumentumok mappájában.

## A szöveg színének megadása

A szöveg színét a teljes weblapra vonatkozóan a `BODY`-objektum *text* (szöveg) tulajdonságával állíthatjuk be. A színeket a háttérszínhez hasonló módon lehet megadni. A **2–11. példában** kék betűket írunk a világoskék háttérre:

```
<BODY bgColor = "lightblue" text = "blue">
  Színes háttér és színes szöveg.
</BODY>
```

Mint már említettük, a weblap több tulajdonsága a BODY- és a *document*-objektumnál is elérhető. A szöveg színét azonban a *document*-objektumnál az *fgColor* (foreground color, előtérszín) tulajdonság értéke határozza meg. Ez pontosan megegyezik a BODY-objektum *text* tulajdonságával, csak a neve más. Alkalmazására a későbbiekben látunk példát.

### Bekezdések a weblapon

A szövegeket bekezdésekkel tagoljuk. Létrehozásukhoz a P (paragraph, bekezdés) objektumosztály bekezdésobjektumát használjuk. Azonosítót csak akkor kapnak, ha a későbbiekben meg akarjuk változtatni a formájukat vagy a tartalmukat. A böngészők a bekezdések között kihagynak egy sort.

Bekezdést úgy készítünk, hogy a szöveget az objektum `<P>` nyitó és `</P>` záró tagja közé írjuk (**2–12. példa**):

```
<BODY>
  <P>
    Ez az első bekezdés.
  </P>
  <P id = "MasodikBekezdes">
    Ez a második bekezdés.
    A bekezdésobjektumnak van azonosítója is.
  </P>
</BODY>
```

Ha csak sortörést akarunk létrehozni, akkor a sor végére a BR (line break, sortörés) objektumot illesztjük. A **2–13. példában** megfigyelhetjük a sortörés és a bekezdés hatását:

```
<P>
  Hull a szilva a fáról,<BR>
  Most jövök a tanyáról.<BR>
  Ej, haj, ruca-ruca<BR>
  Kukorica derce.<BR>
</P>
<P>
  Egyik ága lehajlott,<BR>
  Az én rózsám elhagyott.<BR>
  Ej, haj, ruca-ruca<BR>
  Kukorica derce.<BR>
</P>
```

A sortörés üres objektum, tehát nincsen záró tagja. Mivel a programokban szinte soha nem hivatkozunk rá, általában nem kap azonosítót.

## A bekezdések igazítása

A bekezdésobjektum egyik fontos tulajdonsága az *align* (igazítás). Az *align* értéke lehet:

*left* (balra)

*center* (középre)

*right* (jobbra)

Alkalmazását a **2–14. példa** mutatja be:

```
<P align = "left">
  Balra igazított bekezdés.
</P>
<P align = "center">
  Középre igazított bekezdés.
</P>
<P align = "right">
  Jobbra igazított bekezdés.
</P>
```

A fájl betöltése után próbáljuk megváltoztatni az ablak méretét, és figyeljük meg a szöveg elrendezését!

A többsoros bekezdéseket sorkizárással tudjuk esztétikusan megjeleníteni. Ehhez az *align* tulajdonság értékét *justify*-ra kell állítani. Ha nem adjuk meg az igazítást, akkor a böngésző balra zárja a szöveget.

## Címsorobjektumok

A szövegek tagolására, kiemelésére a HTML-kód hatféle, előre megformázott címsort tartalmaz. A címsorok a H (heading, címsor) osztály objektumai. Az osztálynevek egy H betűből és egy 1-től 6-ig terjedő sorszámból állnak. Az alábbi kóddal 3. szintű címsorobjektumot adunk meg:

```
<H3>
  Ez egy 3. szintű címsor.
</H3>
```

A címsorok formátumát a **2–15. példa** mutatja be. A címsor tartalma bekezdésnek számít, tehát kimarad utána egy sor.

Mint a példából látható, a böngészők a címsorokat balra igazítják. Ennek megváltoztatására a címsorobjektum *align* tulajdonságának megadásával van lehetőségünk. Az *align* tulajdonságot a bekezdésobjektumnál leírt módon használhatjuk.

A **2–16. példa** középre igazítja a címsort:

```
<H1 align = "center">
  Ez egy középre igazított címsor
</H1>
```

A címsorobjektumokkal a címeket emeljük ki, ezért általában nem kapnak azonosítót.

### Betűformázó objektumok

A szövegrészek kiemelésére a B (bold: félkövér), az I (italics: dőlt) és az U (underlined: aláhúzott) osztály objektumait használjuk. A **2–17. példa** bemutatja a betűformázó objektumok hatását:

```
Ez a szöveg
<B>
    félkövér,
</B>
ez pedig
<I>
    dőlt betűkkel
</I>
van írva.
<U>
    Alá is tudjuk húzni a szöveget.
</U>
```

Az egy-egy szóra vonatkozó formázó objektumoknál használt strukturált elrendezés nagyon széttördeli a szöveget, ezért általában nem alkalmazzuk:

```
Ez a szöveg <B>félkövér,</B> ez pedig <I>dőlt betűkkel</I>
van írva. <U>Alá is tudjuk húzni a szöveget.</U>
```

A strukturált elrendezést főleg a programok logikai szerkezetének kiemelésére használjuk.

A **2–18. példában** az objektumokat egymásba ágyazva egyszerre több formátumot is beállítunk:

```
<I>Dőlt, <B>dőlt és félkövér, <U>dőlt, félkövér és aláhúzott
betűk.</U> Ez már nincs aláhúzva,</B> ez már csak dőlt.</I>
```

Ne felejtjük el, hogy először az utoljára megnyitott objektumot kell lezárni!

A HTML-kódban egyéb betűformázó objektumok is szerepelhetnek. Ezeket a **2–19. példa** mutatja be. Felhívjuk a figyelmet a PRE-objektumra, amely a könyvben látható forráskódok betűtípusával, a tagoló karaktereket is figyelembe véve jeleníti meg a szöveget a weblapon. A PRE-objektumon belül tehát minden szóköz és soremelés kikerül a képernyőre pontosan úgy, ahogyan a forráskódban szerepel.

A betűformázó objektumokra a programokból csak ritkán hivatkozunk, ezért általában nem kapnak azonosítót, és egyéb tulajdonságaikat sem használjuk.

### Hivatkozások beillesztése

A World Wide Web a népszerűségét és hatékonyságát a látványos megjelenés mellett főleg azzal nyerte el, hogy a dokumentumba hivatkozásokat (linkeket) lehet beilleszteni. Ezek mutathatnak a lapon belül egy megjelölt részre vagy egy tetszőleges másik fájlra.<sup>5</sup> Az állomány lehet ugyanazon a számítógépen, mint a weblapunk, de lehet a világ bármely más számítógépén, ha elérhetővé tették az Interneten keresztül. A

---

<sup>5</sup> A hivatkozások egyéb lehetőségeit (például mailto stb.) könyvünkben nem tárgyaljuk.

böngészők a hivatkozásokat valamilyen módon kiemelik, a szöveget például automatikusan aláhúzzák.

A hivatkozásokat az A (anchor: horgony, illetve kapocs) osztály objektumainak tartalmához rendeljük. A nyitó és záró tag között lévő objektumokra, szövegekre kattintva a böngésző letölti a célként megadott állományt. A fájl helyét a nyitó tagban szereplő *href* tulajdonság adja meg. Ha a letöltendő állomány ugyanabban a mappában van, mint a hivatkozást tartalmazó dokumentum, akkor a *href* értéke egyszerűen a fájl neve és kiterjesztése. A **2–20. példa** első hivatkozása az előző példa fájljára mutat:

```
<A href = "Pelda2-19.htm">Az előző példa</A>
```

Ha az állomány ugyanazon a gépen, de egy másik mappában található, akkor relatív elérési utat adunk meg. A Fejezet02 mappából a benne lévő Minta mappa fájljára például a következőképpen hivatkozhatunk:

```
<A href = "Minta/MásikFájl.htm">Letöltés az almappából</A>
```

Az elérési útban az eggyel feljebb lévő mappát (a szülő mappát) két pont jelzi. Ha a Minta mappában lévő másik fájlból akarunk visszalépni a Példa2-20.htm-re, akkor a következő hivatkozást használjuk:

```
<A href = "../Pelda2-20.htm">Vissza a szülő mappába</A>
```

Egy távoli gépen lévő állomány esetén a teljes címet kell megadni:

```
<A href = "http://www.origo.hu/index.html">
Letöltés az Internetről</A>
```

Ne feledkezzünk meg a `http://` előtagról, amivel a böngészők kiegészítik a címet a weblapok letöltésénél, de itt nem hagyhatjuk el. Ha nem adunk meg fájlnevet, akkor a webszerver egy előre megadott állományt küld el a számunkra. Ez általában az `index` vagy a `default` nevet viseli.

A hivatkozásokban nem csak HTML-állományok, hanem videók, hangfájlok, képek is szerepelhetnek:

```
<A href = "http://liftoff.msfc.nasa.gov/temp/StnLoc.gif">
  A Nemzetközi Űrállomás helyzete a Föld felett
</A>
```

A fenti hivatkozásokat a 2–20. példa segítségével próbálhatjuk ki.

## Megjegyzések a HTML-kódban

A HTML-kód magyarázatához megjegyzéseket alkalmazhatunk. Szövegük nem jelenik meg a weblapon. A megjegyzéseket a `<!--` és `-->` karaktersorozatok közé írjuk, formai szempontból tehát egy üres elem nyitó tagjának számítanak (**2–21. példa**):

```
<BODY>
  Ez a szöveg megjelenik a weblapon.
  <!--
    Ez a szöveg megjegyzés, tehát nem jelenik meg.
  -->
</BODY>
```

## 2.3. Események a weblapokon

A weblapokat a továbbiakban a programokkal történő kommunikációra, parancsok kiadására, adatok bevitelére, az eredmények megjelenítésére használjuk. A bevitelhez begépelünk valamit a billentyűzeten, kattintunk valahova az egérrel. Ezek a műveletek az operációs rendszer és a program számára eseményeket jelentenek. Egy esemény bekövetkezésekor a böngésző azonosítja a hozzá tartozó objektumot, és megnézi, hogy az adott objektum rendelkezik-e az esemény kezelésére vonatkozó előírásokkal, tennivalókkal.

### Események kezelése

Az esemény bekövetkezésekor végrehajtandó tennivalókat, az eseménykezelést általában programrészletek, úgynevezett eseménykezelő eljárások tartalmazzák. A végrehajtandó utasítások, az eljáráshívások gyakran a megfelelő objektum nyitó tagjában találhatók:

```
<Osztálynév ... eseménynév = "utasítások">
```

A szintaxis szempontjából tehát az esemény neve az objektum egy „tulajdonsága”, az utasítások pedig a tulajdonság értékét jelentik. Ha a nyitó tag rendelkezik egy az eseménynek megfelelő tulajdonsággal, akkor az esemény bekövetkezésekor a böngésző végrehajtja a tulajdonság értékeként megadott utasításokat.

Az eseménykezelés tehát egy programnyelv utasításainak a végrehajtását jelenti. Az utasítások értelmezéséhez a böngészővel közölnünk kell, hogy milyen nyelvet használunk. A programozási nyelvet az objektumok *language* tulajdonsága jelzi. A Visual Basic Script esetén a "VBScript" vagy a "VBS" rövidítéseket alkalmazzuk.

A *language* tulajdonságot sajnos minden eseménykezelőnél meg kellene adnunk. Egyszerűbben járhatunk el, ha a HEAD-be egy SCRIPT-objektumot helyezünk, melynek nyitó tagjába írjuk a *language* értékét. Ez a továbbiakban az egész weblapra érvényes lesz:

```
<SCRIPT language = "VBS">  
</SCRIPT>
```

A SCRIPT-objektumunk egyelőre üres, csak a nyelv megadásához használjuk. A későbbiekben programjainkat ilyen objektumok segítségével illesztjük a dokumentumokba. Megjegyezzük, hogy ha sehol sem szerepel a *language* tulajdonság, akkor a böngésző JavaScript nyelven várja az utasításokat.

Eseményeket kezelni programozási ismeretek nélkül is tudunk. A következő példákban az objektumok tulajdonságait változtatjuk meg egy-egy esemény bekövetkezésekor.

## Hivatkozás az objektumokra és a tulajdonságokra

Egy objektum tulajdonságára az objektum azonosítóját követő pont után írt tulajdonságnévvel hivatkozunk:

```
document.title, Torzs.bgColor, MasodikBekezddes.align
```

Ezt minősített hivatkozásnak nevezzük (az objektum azonosítójával minősítjük a tulajdonságot).

A tulajdonság új értékét a hivatkozást követő egyenlőségjel után adjuk meg. Ha az érték egy karaktersorozat, akkor idézőjelek közé írjuk:

```
Torzs.bgColor = "yellow", MasodikBekezddes.align = "right"
```

Mivel maga az értékadás is egy tulajdonság (az esemény) értéke lesz, így szintén idézőjelbe tesszük. Nem nyithatunk meg azonban egy másik idézőjelet anélkül, hogy le ne zárnánk az elsőt, ezért külső idézőjelként aposztrófot fogunk alkalmazni:

```
'Torzs.bgColor = "yellow"'
```

Ha az objektum tulajdonságának értéke egy szám, akkor nincs szükség a belső idézőjelre. Ilyennel a későbbiekben találkozunk.

## Kattintás az egérgombokkal

Az egyik leggyakoribb esemény akkor jön létre, ha valamelyik objektumra rákattintunk az egérgombbal. Kezeléséhez az objektum nyitó tagját ki kell egészíteni egy *onclick* „tulajdonsággal”, melynek értéke általában egy eljáráshívás. Mivel eljárásokat (más néven szubrutinokat) még nem tudunk készíteni, ezért az alábbiakban eseménykezelésként az objektumok tulajdonságainak adunk új értéket.

A **2–22. példában** a weblap háttérszínét változtatjuk meg a *document*-objektum *bgColor* tulajdonságának az átírásával. Az eseménykezelést most magához a *BODY*-objektumhoz rendeljük, ezért bárhová kattinthatunk a weblapon. Mivel az *onclick* „tulajdonság” értékét (az eseménykezelő utasítást) és benne a szint is idézőjelbe kell tenni, kétféle karaktert (aposztrófot és idézőjelet) használunk:

```
onclick = 'document.bgColor = "yellow"'
```

A *BODY*-objektum nyitó tagja tehát a következőképpen alakul:

```
<BODY bgColor = "red" onclick = 'document.bgColor = "yellow"'>
```

Töltsük be a weblapot a böngészőbe, és kattintsunk valahová az ablak belsejében. Az eredetileg piros háttérszín sárgára fog változni. A weblap frissítésével visszaállítjuk az eredeti színt, és újra kipróbálhatjuk az eseménykezelő működését.

Figyeljük meg, hogy a weblap szövege közli a felhasználóval a tennivalóját. A továbbiakban törekedjünk az ilyen „barátságos” felhasználói felület kialakítására!

A weblap háttérszíne a *BODY* *bgColor* tulajdonságával is elérhető. Maga a *BODY* azonban nem az objektum, hanem az objektumosztály neve. A tulajdonságot viszont az objektum azonosítójával kell minősíteni. Nem használhatjuk az osztály nevét akkor

sem, ha csak egyetlen objektuma létezik. A törzsobjektumra tehát úgy tudunk hivatkozni, ha azonosítóval látjuk el.

A **2–23. példa** az előzővel egyenértékű megoldást mutat be:

```
<BODY id = "Torzs" bgColor = "red"
      onclick = 'Torzs.bgColor = "yellow"'>
```

A **2–24. példában** egy bekezdésobjektum igazítását változtatjuk meg, ha rákattintunk a bekezdésre (az első sorra):

```
<P id = "Szoveg" align = "right"
  onclick = 'Szoveg.align = "left"'>
  Kattintson ide, és megváltozik az igazítás.
</P>
```

A weblap frissítésével állítsuk vissza az eredeti igazítást, majd próbáljunk meg az ablak más területére kattintani. Mivel az eseménykezelőt a bekezdésobjektumhoz rendeltük hozzá, ezért nem fog reagálni az egérműveletre. Bonyolultabb eseménykezelő eljárásokat majd a programozási ismeretek birtokában tudunk írni (például a következő kattintásra visszaállítani az igazítás eredeti értékét).

Megjegyezzük, hogy ha egy kijelölt parancsgomb esetén lenyomjuk az Enter billentyűt, akkor is az *onclick* esemény jön létre.

### Több utasítás az eseménykezelésnél

Egy esemény bekövetkeztekor egyszerre több tulajdonságot is megváltoztathatunk. A felsorolásban az egyes értékadások közé kettőspontot kell írni (ezzel tulajdonképpen két Visual Basic Script utasítást választunk el egymástól). A **2–25. példa** a *Torzs*-objektum háttér- és betűszínét is megváltoztatja:

```
Torzs.bgColor = "yellow" : Torzs.text = "green"
```

Ugyanez a BODY-objektum nyitó tagjában:

```
<BODY id = "Torzs" bgColor = "green" text = "yellow"
      onclick = 'Torzs.bgColor = "yellow" : Torzs.text = "green"'>
```

Az eseménykezelést csak az áttekinthetőség miatt írtuk új sorba. Emlékezzünk vissza arra, hogy a HTML-kódban a tagoló karakterek begépelésének nincsen jelentősége. Figyeljünk a szintaxisra, mindkét értékadás (Visual Basic Script utasítás) az aposztrófjeleken belül áll!

A háttér és a szöveg színe a *document*-objektum tulajdonságaival is megadható. Ezt mutatja be a **2–26. példa**, amely egyenértékű az előző megoldással. Mivel a *document*-objektumra hivatkozunk, a BODY-nak nem kell azonosítót adni. A háttér színe a *document*-nél is a *bgColor* tulajdonság, de a szöveg színét a *text* helyett az *fgColor* határozza meg:

```
<BODY bgColor = "green" text = "yellow"
      onclick = 'document.bgColor = "yellow" :
                document.fgColor = "green"'>
```



## Az ondblclick esemény

Egyes műveletek végrehajtásához duplán kell kattintani az egérrel. A dupla kattintás hozza létre az *ondblclick* (double click) eseményt. Kezeléséhez az objektum nyitó tagját ki kell egészíteni az *ondblclick* tulajdonsággal, melynek értéke tartalmazza a végrehajtandó műveleteket. A **2–27. példa** a dupla kattintás hatására megváltoztatja a címsor igazítását:

```
<P ondblclick = 'Cimsor.align = "right"'>
  Valóban balra igazítottuk a címsort?
  Kattintson duplán erre a sorra!
</P>
```

Töltsük be a példafájlt a böngészőbe, és próbáljunk meg duplán kattintani az ablak különböző részeire. Mivel az eseménykezelést a bekezdésobjektumhoz rendeltük hozzá, csak a szövegére történő dupla kattintásnál változik meg az igazítás. A böngésző a dupla kattintásra kijelöli a szót. A kijelölés megszüntetéséhez kattintsunk bárhová egyszer az egérrel, frissítsük a weblapot, majd kattintsunk egyszer a szövegre. Az *onclick* eseményhez nem rendeltünk utasításokat, ezért nem történik semmi.

Figyeljük meg, hogy a példában a bekezdésobjektum eseménykezelője a címsor igazítását módosította! Az egyik objektum egy másik objektum tulajdonságát változtatta meg.

## Több esemény kezelése egy objektumnál

Egy objektum több eseménykezelővel is rendelkezhet. Mindegyiket fel kell venni a nyitó tagba a tulajdonságok közé, és meg kell adni, hogy mi történjen az esemény bekövetkezésekor.

Két eseménykezelő segítségével például már visszaválthatjuk a színeket. A **2–28. példában** egy kattintásra megváltozik a háttér és a szöveg színe, dupla kattintásra pedig megint az eredeti értékeket veszik fel:

```
<BODY id = "Torzs" bgColor = "green" text = "yellow"
  onclick = 'Torzs.bgColor = "yellow" :
              Torzs.text = "green"'
  ondblclick = 'Torzs.bgColor = "green" :
                Torzs.text = "yellow"'>
```

## 2.4. Az objektumok metódusai

Az eddigiek során megismerkedtünk a weblapot alkotó néhány objektummal, tulajdonságaikkal, és bemutattuk, hogyan lehet ezeket a tulajdonságokat eseményekkel megváltoztatni. Az objektumok a tulajdonságok mellett metódusokkal is rendelkeznek. A metódus olyan tevékenység, melyet az objektum el tud végezni. Ezeket a tevékenységeket programjaink utasításai indítják el, de eseményekkel is kiválthatjuk a végrehajtásukat.

### A metódusok és paraméterek megadása

A metódusokra a tulajdonságokhoz hasonlóan hivatkozhatunk. Az objektum azonosítója után pontot teszünk, majd szóköz nélkül megadjuk a metódus nevét. A metódusoknak adatokat adhatunk át, amiket paramétereknek nevezünk.<sup>6</sup> A továbbiakban eljárásokkal és függvényekkel is megismerkedünk, melyek szintén kaphatnak adatokat.

Paraméter: a metódusoknak, függvényeknek, eljárásoknak átadott adat.

A paramétereket a metódusnév után kell felsorolni, egymástól vesszővel elválasztva:

```
objektum_azonosító.metódusnév Paraméter1, Paraméter2, ...
```

Megjegyezzük, hogy a programokban a paraméterek listáját zárójelbe tesszük.

### A window-objektum metódusai

Bár az objektumok metódusait majd a programozás során fogjuk igazán felhasználni, szemléltetésként bemutatjuk a *window*-objektum néhány metódusát.

A *resizeTo* metódus átméretezi az ablakot. Paraméterként a vízszintes és függőleges méret új értékét kell megadni pixelben:<sup>7</sup>

```
window.resizeTo új_szélesség, új_magasság
```

A **2–29. példa** egérekattintásra az ablak szélességét 400 pixelre, magasságát pedig 300 pixelre állítja:

```
<BODY onclick = "window.resizeTo 400, 300">
```

A *close* metódus bezárja az ablakot. Nem igényel további adatokat, ezért nem adunk meg paramétereket. Az Internet Explorer ennek a „veszélyes” tevékenységnek a végrehajtása előtt figyelmezteti a felhasználót, és lehetővé teszi, hogy leállítsuk a folyamatot. A **2–30. példában** egérekattintásra bezárul az ablak:

```
<BODY onclick = "window.close">
```

---

<sup>6</sup> A formális és aktuális paraméterek közötti különbséget később részletezzük.

<sup>7</sup> Pixelnek nevezzük a képernyő egy pontját. Az általában használt képernyő 800 pixel széles és 600 pixel magas. Nagyobb felbontás esetén gyakran 1024 pixel szélességet és 768 pixel magasságot állítanak be.

## Üzenet a felhasználónak

A *window*-objektum metódusainak áttekintését egy hasznos és látványos tevékenységgel zárjuk. Az *alert* metódussal üzenetet küldhetünk a felhasználónak. A végrehajtás során az üzenet egy párbeszédablakban jelenik meg, amely az OK gombra való kattintással vagy a szokásos Bezárás gombbal (☒) tüntethető el. Az üzenetet idézőjelben kell megadni a metódus neve után. A következő metódushívás az „Ez egy üzenet.” szöveget jeleníti meg az ablakban:

```
window.alert "Ez egy üzenet."
```

A metódus végrehajtását a **2–31. példában** az *onclick* eseménnyel váltjuk ki:

```
<BODY onclick = 'window.alert "Ez egy üzenet."'>
```

Az eseménykezelést bármelyik objektumhoz hozzárendelhetjük. A **2–32. példa** az üzenetben megjeleníti, hogy milyen típusú kiemelésre kattintottunk rá:

```
<B onclick = 'window.alert "Félkövér kiemelés."'>
  Kattintson
</B>
az egérrel
<I onclick = 'window.alert "Dőlt betűs kiemelés."'>
  valamelyik
</I>
kiemelt
<U onclick = 'window.alert "Kiemelés aláhúzással."'>
  szövegrészre!
</U>
```

Megjegyezzük, hogy a *window*-objektum metódusainál elhagyható az objektumra való hivatkozás, így például *window.alert* helyett csak *alert*-et is írhatunk. Ezt az egyszerűsítést azonban nem fogjuk alkalmazni, mert hangsúlyozni szeretnénk az objektumok és metódusaik kapcsolatát.

## Az eseménybuborék

A képernyőn megjelenő objektumok átfedhetik egymást. Az ablakban van a web-lap (dokumentum), a weblapon egy bekezdés, a bekezdésben egy dőlt betűs szó. Minden egyes objektumhoz tartozhat eseménykezelő eljárás. A Windows először a legbelső objektum eseménykezelőjét hajtja végre, majd egymás után, egyre kijebb halad, míg el nem érkezik a legkülső objektumhoz. Az esemény „száll fölfelé”, mint a vízben a buborék.

Eseménybuborék: az esemény terjedése az objektum-hierarchiában.

A **2–33. példában** az egérekattintáshoz eseménykezelést rendeltünk a törzs, a bekezdés és a dőlt betűs megjelenítést végző objektumnál is:

```
<BODY onclick = 'window.alert "A weblapra kattintott."'>
  <P onclick = 'window.alert "A bekezdésre kattintott."'>
    Kattintson a
    <I onclick = 'window.alert "A dőlt betűs felírára
      kattintott."'>
      dőlt betűs
    </I>
    felírára!
  </P>
</BODY>
```

A dőlt betűs szöveget a bekezdésobjektum tartalmazza, a bekezdésobjektumot pedig a törzsobjektum. Ha a dőlt betűs részre kattintunk, akkor működésbe lép az I-objektum eseménykezelője, megjelenik „A dőlt betűs felírára kattintott.” üzenet. Az eseménykezelés végrehajtása után az esemény továbbadódik a bekezdésobjektumnak, megjelenik „A bekezdésre kattintott.” üzenet, majd a törzsobjektumnak, ami megjeleníti „A weblapra kattintott.” üzenetet. Kattintsunk normál szövegre vagy az ablak egy üres részére, és figyeljük a megjelenő üzeneteket!

### Az eseménybuborék leállítása

Egy esemény bekövetkezése létrehozza az *event* (esemény) objektumot, melynek tulajdonságai hordozzák az esemény jellemzőit. Az *event*-objektumra a *window*-val együtt hivatkozunk: *window.event*.

Ha az eseménybuborék terjedését le akarjuk állítani, akkor az utoljára végrehajtandó eseménykezelőben az eseményobjektum *cancelBubble* (a buborék terjedésének megszakítása) tulajdonságát *True*-ra (igaz értékre) kell állítani:

```
window.event.cancelBubble = True
```

A bekezdésobjektum módosítása a **2–34. példában** már megakadályozza a törzsobjektum eseménykezelőjének működését:

```
<P onclick = 'window.alert "A bekezdésre kattintott." :
  window.event.cancelBubble = True'>
```

Így a kattintás hatására megjelenik „A dőlt betűs felírára kattintott.” és „A bekezdésre kattintott.” üzenet, de már nem jelenik meg „A weblapra kattintott.” üzenet. Ha az üres háttérre kattintunk, akkor természetesen működésbe lép a BODY eseménykezelője.

Emlékeztetünk arra, hogy több utasítás használata esetén kettősponttal kell őket elválasztani egymástól. A kód begépelésénél figyeljünk a szintaxisra! A két utasítást aposztrófjelek közé írjuk, hiszen ez a karaktersorozat formailag az *onclick* tulajdonság értéke.

## 2.5. Parancsgombok használata

Az előzőekben az eseményeket magára a weblapra vagy valamely szövegrészére való kattintással hoztuk létre. A továbbiakban azonban általában parancsgombokat használunk a programok vezérlésére, az eseménykezelést ezekhez a parancsgombokhoz rendeljük hozzá.

### Az INPUT objektumosztály

A parancsgombok egy többcélú osztály, az INPUT objektumai. Az objektum *type* (típus) tulajdonságát *button*-ra (parancsgomb) kell állítani. A *value* (érték) tulajdonság pedig megadja a parancsgomb feliratát. Itt tetszőleges karaktereket használhatunk:

```
<INPUT type = "button" value = "Kattintson ide!">
```

Az INPUT osztály egyedei üres objektumok, így nincsen záró tagjuk.

A következő példában a parancsgombot az első sor (bekezdés) közepére helyezzük. A **2–35. példában** az igazítást a bekezdés *align* tulajdonságával adjuk meg:

```
<P align = "center">  
  <INPUT type = "button" value = "Kattintson ide!">  
</P>
```

Mivel a gombhoz nem rendeltünk *onclick* tulajdonságot, a kattintásra természetesen nem történik semmi.

A későbbiekben az INPUT osztály többféle objektumával is megismerkedünk.

### Eseménykezelés parancsgombokkal

A parancsgombok áttekinthető vezérlést tesznek lehetővé. Az esemény nevét és a végrehajtásra kerülő utasítást az eddigiekhez hasonlóan az objektum nyitó tagjában kell megadni. A **2–36. példában** a háttérszín a parancsgombra való kattintással tudjuk megváltoztatni:

```
<BODY bgColor = "yellow">  
  <P align = "center">  
    <INPUT type = "button" value = "Piros háttér"  
      onclick = 'document.bgColor = "red"'>  
  </P>  
</BODY>
```

A **2–37. példa** két parancsgombjának a segítségével már oda-vissza tudjuk változtatni a háttérszín:

```
<INPUT type = "button" value = "Piros háttér"  
  onclick = 'document.bgColor = "red"'>  
<INPUT type = "button" value = "Sárga háttér"  
  onclick = 'document.bgColor = "yellow"'>
```

### Az állapotsor tartalma

A Windows operációs rendszer alatt futó programok általában egy-egy külön ablakban jelennek meg. Az ablakok szerkezete hasonlít egymáshoz, fölül találjuk a menüsört, eszköztárat, gyakran görgetősávokat is használunk. Az ablakok alsó sorában a programok, a kezelt dokumentumok állapotával kapcsolatos információkat láthatunk. Az Internet Explorer például egy dokumentum betöltésekor itt jelenti be, hogy „Kész”. Ezt a sávot állapotsornak nevezzük.

Az állapotsor szövege a *window*-objektum *status* tulajdonságával adható meg. Ezt a **2–38. példa** parancsgombjához rendelt eseménykezelő segítségével módunk van megváltoztatni:

```
<INPUT type = "button" value = "Változtat!"  
      onclick = 'window.status = "Üdvözlöm a kedves Olvasót!"'>
```

A böngésző a további műveletek végzésénél (például a weblap frissítésekor) átírhatja az állapotsor általunk beállított szövegét.

### Parancsgombok tiltása és engedélyezése

Egy program végrehajtása során előfordulhat, hogy a képernyőn megjelenített parancsgombok közül valamelyiknek a használatát meg akarjuk tiltani. Ehhez az objektum *disabled* (tiltva) tulajdonságát kell beállítani. A *disabled* tulajdonság értéke *False* (hamis), ha nincs tiltva a használata, és *True* (igaz), ha letiltottuk a használatát. A *True* és *False* kulcsszavak fontos szerepet töltenek be a programozás során. Nem tesszük őket idézőjelbe.

A **2–39. példában** hiába kattintunk a gombra, nem változik meg a háttérszín:

```
<INPUT type = "button" value = "Piros háttér"  
      onclick = 'document.backgroundColor = "red"' disabled = True>
```

A böngésző szürkén jeleníti meg a feliratot. Ezzel jelzi, hogy letiltottuk a parancsgomb működését.

A **2–40. példában** a *disabled* tulajdonság értékének változtatásával az első parancsgomb letiltja, az utolsó pedig ismét engedélyezi a középső működését. A hivatkozáshoz a középső parancsgombot azonosítóval láttuk el:

```
<INPUT type = "button" value = "Tilt"  
      onclick = "Szinez.disabled = True">  
<INPUT id = "Szinez" type = "button"  
      value = "Piros háttér"  
      onclick = 'document.backgroundColor = "red"'>  
<INPUT type = "button" value = "Enged"  
      onclick = "Szinez.disabled = False">
```

Figyeljük meg, hogy a weblap összeállításánál az első parancsgomb elkészítésekor a böngésző még nem ismeri a *Szinez* azonosítójú objektumot, amelyre pedig hivatkozunk az *onclick* értékének megadásánál! Ez általában hibaüzenetet eredményezne. De az eseménykezelő utasításokat csak az esemény bekövetkeztekor, tehát már az összes objektum megjelenítése után hajtja végre az interpreter. A HTML-kód feldolgozásakor

a böngésző csak regisztrálja, hogy van egy ilyen eseménykezelő, ezért nem kapunk hibaüzenetet.

## A parancsgomb méretének beállítása

Mint az előző példában szereplő weblapon láthattuk, a parancsgomb méretét a böngésző a felirat hosszához igazítja. Ez nem túl szép megjelenést eredményez, a *Tilt* feliratú gomb például elég kicsi lett.

A parancsgombok méretét a *style* (stílus) tulajdonsággal szabhatjuk meg. A *style* tulajdonság számos lehetőséget nyújt az objektumok formájának szabályozására, ezért a többi tulajdonságtól kissé előrében használjuk. Az idézőjelen belül meg kell adni, hogy az objektum megjelenésének mely elemét akarjuk megváltoztatni, majd egy kettőspont után következik a tulajdonság új értéke. Ha az objektum szélességét (*width*) írjuk elő, akkor a következő szintaxist alkalmazzuk:

```
style = "width: A_szélesség_új_értékeMértékegység"
```

Több mértékegység közül választhatunk, mi csak a pixelt fogjuk használni (rövidítése: px). A következő kód az objektum szélességét 100 pixelre állítja:

```
style = "width: 100px"
```

A szám és a mértékegység között nem hagyunk szóközt.

A **2–41. példa** már nem bízta a böngészőre a működést tiltó és engedélyező parancsgombok méretének meghatározását, mindkettőt 100 pixel szélességűre állítja:

```
<INPUT type = "button" value = "Tilt" style = "width: 100px"
      onclick = "Szinez.disabled = True">
<INPUT id = "Szinez" type = "button"
      value = "Piros háttér"
      onclick = 'document.bgColor = "red"'>
<INPUT type = "button" value = "Enged" style = "width: 100px"
      onclick = "Szinez.disabled = False">
```

Így már sokkal szebb megjelenést biztosítottunk a parancsgomboknak.

Megjegyezzük, hogy a pixel használatánál a mértékegység jelölése elhagyható, így a "width: 100px" helyett írhatunk csak "width: 100"-at is.

## 2.6. Szövegek beírása és megjelenítése

Programjaink a billentyűzet, az egér és a képernyő segítségével kommunikálnak a felhasználóval. Egyszerűbb esetben ezeket az eszközöket használjuk fel az adatok bevitelére, az eredmények megjelenítésére is. Ehhez meg kell ismerkednünk a szövegmező-objektummal és az objektumok szövegének a módosításával.

### Az innerText tulajdonság

A legtöbb szöveget megjelenítő objektum rendelkezik az *innerText* (belső szöveg) tulajdonsággal, melynek értéke maga a szöveg (karakter sorozat). A hivatkozáshoz azonosítóval kell ellátni az objektumot. A **2–42. példa** megváltoztatja a második bekezdés szövegét, ha a parancsgombra kattintunk:

```
<P>
  Ez a bekezdés nem változik.
</P>
<P id = "ValtozoSzoveg">
  Ez a szöveg megváltozik.
</P>
<P>
  Ez a bekezdés sem változik.
</P>
<INPUT type = "button" value = "Változzon a szöveg!"
        onclick = 'ValtozoSzoveg.innerText = "Ez már az
                    új szöveg."'>
```

A *BODY*-objektum *innerText* tulajdonsága az egész weblap szövegét tartalmazza. Felülírása az előző példához hasonlóan eltünteti a régi tartalmat, itt magát a parancsgombot is (**2–43. példa**):

```
<BODY id = "Torzs">
  <INPUT type = "button" value = "Írjon ide valamit!"
        onclick = 'Torzs.innerText = "Valami :)"'>
</BODY>
```

Megjegyezzük, hogy a weblap HTML-kódja nem változott, csak a megjelenése. Erről meggyőződhetünk, ha a változtatás után a böngészőből megnyitjuk a forráskódot. Ezt az Internet Explorerben a Nézet menü Forrás parancsának segítségével tehetjük meg. Ekkor megnyílik a Jegyzettömb, és láthatjuk, hogy a forráskódban megmaradt az *INPUT*-objektum. Frissítés után ismét megjelenik az eredeti tartalom. Ha elmentjük a weblapot, akkor az eredeti változat kerül a háttértárra.

Az *innerText* tulajdonság értéke lehet szám is, azonban a számot nem tesszük idézőjelbe (**2–44. példa**):

```
<INPUT type = "button" value = "Ide egy szám kerül."
        onclick = "Torzs.innerText = 1234567890">
```

### Az *innerHTML* tulajdonság

Ha HTML-kódot szeretnénk felhasználni az objektumban, akkor az *innerText* tulajdonság helyett az *innerHTML*-t kell megváltoztatni. A **2–45. példa** egy szót a *B*-objektum segítségével emel ki a szövegben:

```
<P id = "ValtozoSzoveg">
  Ez a szöveg megváltozik.
</P>
<INPUT type = "button" value = "Változzon a szöveg!"
        onclick = 'ValtozoSzoveg.innerHTML = "Ez már az
                    <B>új</B> szöveg"'>
```

Az *innerText* és az *innerHTML* tulajdonságok közötti egyetlen lényeges különbség, hogy az *innerText* nem tartalmazhat HTML-kódot.



## A document-objektum write metódusa

A programokban gyakran a *document*-objektum *write* metódusát használjuk a weblapok összeállításához. Ez az eljárás az eddigi kiírásokkal ellentétben nem egy objektumnak a weblapon is megjelenő szöveg-tulajdonságát szabja meg, hanem magát a forráskódot vagy annak egy részét hozza létre.

Töltsük be a **2–46. példát** a böngészőbe, és kattintsunk bárhová a lapon:

```
<BODY onclick = 'document.write "Most már van szöveg is."'>
```

Az *innerHTML*-hez hasonlóan a metódus által beírt karaktersorozat nyitó és záró tagokat is tartalmazhat (**2–47. példa**):

```
<BODY onclick = 'document.write "Első sor<BR>Második <B>sor</B>"'>
```

Mivel a *document.write* metódus magát a forráskódot hozza létre, így a weblap megjelenítése után végrehajtva törli a teljes HTML-kódot! Töltsük be újra a böngészőbe az előző példát, és a kattintás előtt figyeljük meg a címsor szövegét! A kattintás után eltűnik a felirat. A forráskódban is csak a *document.write* által beírt karaktersorozat marad, és nem tudjuk frissítéssel visszaállítani az eredeti tartalmat (mentés nélkül a fájl természetesen nem írja át)! Ezért a *document.write* metódust csak akkor használjuk, ha a HTML-kódot a programunk hozza létre a weblap megjelenítése előtt.

A metódus változata a *document.writeln* (write line, kiírás soremeléssel), amely a HTML-kódba írt szöveg után sort emel, mintha a beírásnál Entert nyomtunk volna. Ez a böngésző szempontjából tagoló karakternek számít, tehát a PRE- vagy az XMP-objektumokat kivéve nem hoz létre új sort a weblapon.

## A szövegmező-objektum

Az INPUT objektumosztály fontos típusa a szövegmező. Programjaink általában szövegmezők segítségével olvasnak be adatokat. Szövegmező létrehozásához a *type* tulajdonságot *text*-re kell állítani.

A **2–48. példa** egy szövegmezőt helyez el a weblapon:

```
<INPUT type = "text">
```

A szövegmező üres objektum, tehát nem tartalmaz további elemeket és nincsen záró tagja. Tulajdonságait a **2–49. példa** mutatja be.

A *size* (méret) tulajdonsággal adjuk meg, hogy a szövegmező hány karakter hosszúságú legyen. A következő kóddal létrehozott szövegmezőben 10 karaktert lehet látni egyszerre (de többet is beírhatunk):

```
<INPUT type = "text" size = 10>
```

A méret szám, tehát nem tesszük idézőjelbe. Ez a tulajdonság csak egy átlagos szélességű betű esetén pontos, a keskeny i betűből több, a széles m betűből kevesebb fér bele. Az Internet Explorer túl kicsi szám esetén a megadott értéknél kissé szélesebbre veszi a méretet.

A szövegmező méretét a parancsgombnál megismert módon pixelben is megadhatjuk, ekkor a *style* tulajdonságot kell használnunk. Egy 100 pixel széles szövegmező HTML-kódja például:

```
<INPUT type = "text" style = "width: 100px">
```

A méretet jelző tulajdonság nem korlátozza a beírható szöveg hosszát. Erre a *maxLength* (maximális hossz) tulajdonságot használhatjuk. A következő kóddal megadott szövegmezőbe legfeljebb 3 karaktert írhatunk be:

```
<INPUT type = "text" maxLength = 3>
```

A szövegmező tartalmát a *value* tulajdonság szabja meg. Ha a nyitó tagban szerepel az értéke, akkor a böngésző a megjelenítésnél beleírja a megadott karaktersorozatot. A következő kód megjeleníti az „Írjon ide valamit!” szöveget a szövegmezőben:

```
<INPUT type = "text" value = "Írjon ide valamit!">
```

Esetenként szükségünk lehet arra, hogy átmenetileg letiltsuk a szövegmező tartalmának a megváltoztatását. Ehhez a parancsgombnál megismert *disabled* tulajdonságot használhatjuk. Értéke *True*, ha tiltjuk a szövegmezőbe való írást, és *False*, ha írhatunk bele. A következő kód esetén a szövegmező tartalma nem módosítható:

```
<INPUT type = "text" value = "Ide nem írhat semmit!"  
      disabled = True>
```

A *disabled* beállításakor a szöveg szürkén jelenik meg, így csak nehezen olvasható. Ha tiltani akarjuk a beírást, de el akarjuk kerülni a szürkítést, akkor a *readOnly* (csak olvasható) tulajdonságot kell használni. Értéke *False* (hamis), ha a szövegmezőbe engedélyezzük, és *True* (igaz), ha tiltjuk a beírást:

```
<INPUT type = "text" value = "Ide nem írhat semmit!"  
      readOnly = True>
```

Az érdekesség kedvéért megemlítjük a *password* (jelszó) típusú INPUT-objektumot, amely teljesen hasonlóan viselkedik, mint a szövegmező, de a begépett karakterek helyett csillagok jelennek meg benne. A *value* tulajdonság természetesen a valójában beírt karaktereket tartalmazza:

```
<INPUT type = "password">
```

### A szövegmező tartalmának elérése

A szövegmezők *value* tulajdonságára eseménykezelő eljárásokban is hivatkozhatunk, így átírhatjuk más objektumokba. A következő példában a szövegmező tartalma kattintásra átkerül egy bekezdésbe. A hivatkozáshoz a szövegmezőt és a bekezdést azonosítóval kell ellátni:

```
<INPUT id = "Bevitel" type = "text">  
...  
<P id = "Tartalom"></P>
```

A *Bevitel.value* megadja a szövegmezőbe írt szöveget, amit a *Tartalom*-objektum *innerText* tulajdonságába kell átmásolnunk. Ezt végzi el a következő utasítás:

```
Tartalom.innerText = Bevitel.value
```

Ezeket az úgynevezett értékadó utasításokat mindig úgy kell értelmezni, hogy az egyenlőségjel jobb oldalán szereplő érték kerül a bal oldalon szereplő helyre. A bekezdés eredeti tartalma ilyenkor törlődik. Az értékadó utasításokkal a következő fejezetben részletesebben foglalkozunk, de az eddigi eseménykezelésnél is ezeket használtuk.

A feladatot végrehajtó kód a **2–50. példában** látható:

```
<INPUT id = "Bevitel" type = "text">
<INPUT type = "button" value = "Másolás"
      onclick = "Tartalom.innerText = Bevitel.value">
<P>
  A szövegmező tartalma:
</P>
<P id = "Tartalom">
</P>
```

## 2.7. Konténerobjektumok

### Az inline- és a blokkobjektumok

Az objektumok egy része a weblap szövegének megjelenítését szolgálja. A címsor (H1 ... H6), a bekezdés (P) objektumok szövege új sorban kezdődik. Ezeket blokkobjektumoknak nevezzük. A betűformázó (B, U, I stb.) objektumok nem hoznak létre sortörést, ezek úgynevezett inline (soron belüli) objektumok.

Blokkobjektum: önálló bekezdést alkot.

Inline-objektum: egy soron belül helyezkedik el, nem hoz létre sortörést.

Egy weblap tervezésénél, a HTML-kód írásánál figyelembe kell vennünk az objektumok ezen viselkedését.

### Konténerobjektumok

Gyakran szükségünk lesz a dokumentumot alkotó objektumok csoportosítására, összefogására. Ezt a szerepet a DIV- és a SPAN-objektum tölti be. Konténereknek hívjuk őket, mert objektumokat, illetve szövegrészeket foglalnak magukba. A weblapon csak a tartalmuk jelenik meg.

Konténer: más objektumokat tartalmazó objektum.

A DIV blokk-, a SPAN pedig inline-objektum.

### A DIV-objektum használata

A DIV-objektumot kifejezetten objektumok csoportba foglalására használjuk. A **2–51. példában** az *align* tulajdonságot a DIV-nél adjuk meg, így nem kell az összes bekezdésénél megismételni:

```
<BODY>
  <P>Balra igazít.</P>
  <DIV align = "center">
    <P>1. középre igazított bekezdés.</P>
    <P>2. középre igazított bekezdés.</P>
    <P>3. középre igazított bekezdés.</P>
  </DIV>
  <P>Megint balra igazít.</P>
</BODY>
```

Megjegyezzük, hogy ezt az elrendezést a CENTER-objektummal is elérhettük volna, de a DIV sokkal több célra használható.

### A SPAN-objektum

A SPAN-objektummal általában a szöveg egy másként ki nem emelt részét különítjük el a többitől. Így tulajdonságokat vagy eseménykezelést rendelhetünk hozzá.

A **2–52. példában** a SPAN-objektumot eseménykezelésre használjuk. Ha rákattintunk az „ide” szóra, akkor pirosra változik a háttér színe. Mivel a szót semmilyen módon nem emeltük ki, csak a SPAN-objektummal tudunk eseménykezelést hozzárendelni:

```
<BODY bgColor = "yellow">
  Kattintson
  <SPAN onclick = 'document.bgColor = "red"'>
    --ide--,
  </SPAN>
  és piros lesz a háttér.
</BODY>
```

Ha a fenti példában a SPAN helyett DIV-et írnánk, akkor az „ide” szó új bekezdésbe kerülne.

A SPAN-objektumot arra is felhasználhatjuk, hogy a programokból kiegészítsük a weblapon megjelenített szöveget. Ehhez a HTML-kódba üres SPAN-objektumot illesztünk, amely a tartalmát egy esemény bekövetkezésekor kapja meg. Ezt mutatja be a **2–53. példa**, amely név szerint üdvözli a felhasználót. A parancsgomb eseménykezelője a szövegmező *value* tulajdonságát (a begépelt nevet) kattintáskor átmásolja a SPAN-objektum *innerText* tulajdonságába, azaz megjeleníti azt a weblapon:

```
<INPUT id = "Beiras" type = "text">
<INPUT type = "button" value = "Köszön"
  onclick = "Nev.innerText = Beiras.value">
<P>
  Üdvözlöm, kedves <SPAN id = "Nev"></SPAN> barátom!
</P>
```

Figyeljük meg, hogy a HTML-kód ellenére nem maradt szóköz a SPAN záró tagja

és a „barátom” szó között! Ezt csak egy nem törhető szóköz (*nbsp*) beillesztésével tudjuk elérni (**2–54. példa**):

```
Üdvözlöm, kedves <SPAN id = "Nev"></SPAN> &nbsp;barátom!
```

### Stílusok alkalmazása<sup>8</sup>

A parancsgombok méretének megadásánál már találkoztunk a stílus tulajdonsággal. Az objektumok *style* (stílus) tulajdonsága igen sokrétű formázásra ad lehetőséget. Beállításánál az idézőjelen belül először meg kell adni a stíluselem nevét, majd kettősponttal elválasztva az értékét:

```
style = "stíluselem_neve: értéke"
```

A stíluselem határozza meg, hogy az objektum megjelenésének melyik jellemzőjét akarjuk megadni. A következő kóddal például a betűméretet (*font-size*) 50 pixele állítjuk:

```
<SPAN style = "font-size: 50px">Jó nagy betűk</SPAN>
```

Egy idézőjelen belül több stíluselem is szerepelhet. Ekkor pontosvesszővel választjuk el őket egymástól. A **2–55. példa** piros betűket (*color*: szín) jelenít meg sárga háttérrel (*background-color*: háttérszín):

```
<SPAN style = "color: red; background-color: yellow">
  Színes betűk, színes háttér
</SPAN>
```

A fenti példában csak a SPAN-objektumra vonatkoznak a beállítások. Így a szöveg egy-egy részét nagyon változatos módon emelhetjük ki. A HTML-kód számos objektumának lehet *style* tulajdonsága. A legfontosabb stíluselemeket a CD-melléklet Dokumentumok mappájában mutatjuk be.

Megjegyezzük, hogy a szkriptekben a fentiektől kissé eltérő szintaxist kell használnunk a stíluselemek megadásához. Ezzel később foglalkozunk.

### A dokumentum objektummodell

Az eddigiek során jó néhány objektummal, tulajdonsággal, metódussal megismerkedtünk. Rendszerüket az úgynevezett dokumentum objektummodell foglalja össze.

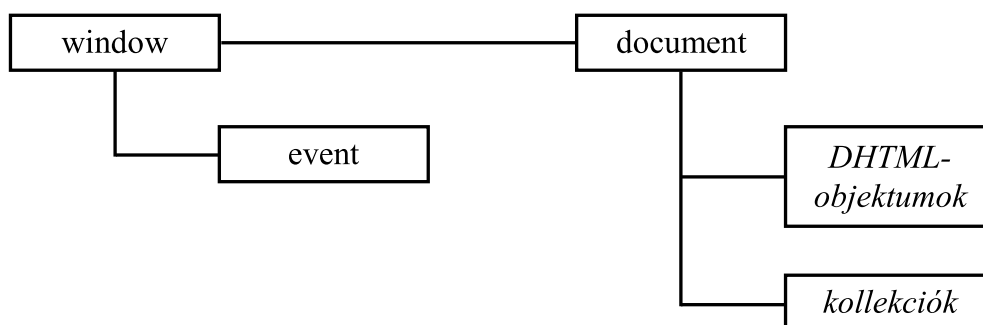
Dokumentum objektummodell (DOM): a weblapot alkotó objektumok, tulajdonságok, metódusok összefüggő és egységes rendszere.

A DOM alapelve szerint a HTML-kód minden egyes eleme objektumként viselkedik, melynek tulajdonságai és metódusai a különböző programozási nyelvekből elérhetők, módosíthatók. A DOM meghatározza az egyes objektumok hierarchiáját és azoknak az eseményeknek a halmazát, melyekre az objektumok reagálni tudnak. Ha az objektum rendelkezik a megfelelő eseménykezelővel, akkor annak utasításai az ese-

<sup>8</sup> Könyvünkben csak az inline stílusokat tárgyaljuk, mivel ezeket használjuk a programozásnál.

mény bekövetkezésekor végrehajtnak.

Hangsúlyozzuk, hogy a dokumentum objektummodell összetevői egyik programnyelvhez sem kötődnek, és minden programnyelvből elérhetők. Az egyes böngészők azonban kissé eltérő modelleket alkalmaznak. Az Internet Explorer rendszerét dinamikus HTML objektummodellnek hívják (DHTML). Ezért a továbbiakban a dokumentum objektummodell objektumait DHTML-objektumoknak nevezzük.



2–1. ábra. A dokumentum objektummodell egyszerűsített diagramja

Érdekességgként megemlíjtük, hogy az objektumokat saját tulajdonságokkal is bővíthetjük. Ehhez a nyitó tagban meg kell adni a tulajdonság nevét és értékét:

```
<Osztálynév id = "Azonosító" tulajdonságnév = "érték">
```

Az alábbi kódban a középre igazított H3-objektumot egy *CicaNev* tulajdonsággal látjuk el, melynek a *Cirmi* értéket adjuk:

```
<H3 id = "Sajat" align = "center" CicaNev = "Cirmi">
  Sajat tulajdonság használata
</H3>
```

A saját tulajdonságok természetesen nem változtatják meg a weblap megjelenését, de adatok tárolására fel lehet őket használni. Elérésük módja megegyezik a DHTML-objektumok más tulajdonságainak elérésével. A **2–56. példa** parancsgombjának eseménykezelője kiírja a *CicaNev* értékét az üzenetablakba:

```
<INPUT type = "button" value = "Kiírás"
  onclick = "Nev.innerText = Sajat.CicaNev"><BR>
A cica neve: <SPAN id = "Nev"></SPAN>
```

## 3. A VBSCRIPT ALAPJAI

Az előző részben megismerkedtünk azokkal az objektumokkal, melyek segítségével adatokat közölhetünk a programokkal, vezérelhetjük a végrehajtást, és megjeleníthetjük az eredményeket. Ebben a fejezetben áttekintjük a programozás alapelemeit, és megtanuljuk, hogyan helyezhetünk el Visual Basic Script programokat a HTML-kódban.

### 3.1. Programok a HTML-kódban

#### A HTML és a VBScript

A böngészők jelenleg kétféle szkript nyelvet ismernek, a JavaScriptet és a Visual Basic Scriptet. A Microsoft által kifejlesztett, pontos nevén Visual Basic Scripting Editiont a továbbiakban VBScriptnek fogjuk nevezni. A VBScript szintaxisa és logikája hasonlít a Visual Basichez, illetve a Word, az Excel, az Access programozására használt Visual Basic for Applications nyelvhez. A VBScript teljes értékű, objektumalapú<sup>9</sup> programozási nyelv, de – főleg a fájlkezeléssel kapcsolatban – némileg korlátozott lehetőségekkel rendelkezik. Elsősorban a weblapok és a webszerverek programozására hozták létre, azonban más célokra is alkalmazható.

A szkriptek önállóan nem futtathatók, szükségük van egy értelmező (interpreter) programra. Ilyen interpretereket a böngészők tartalmazznak. Az interpreter a HTML-kód feldolgozásakor lefordítja, és végrehajtja az utasításokat.

#### A SCRIPT-objektum

A szkriptek utasításai a HTML-kódban a SCRIPT-objektumok nyitó és záró tagjai között helyezkednek el:

```
<SCRIPT>
    utasítások
</SCRIPT>
```

A SCRIPT-objektumok rendelkezhetnek azonosítóval (*id*), amely nagyon rugalmas, dinamikusan változó utasítássorozatok alkalmazását teszi lehetővé. Megváltoztathatók a bennük szereplő utasítások, sőt, a teljes szkript törölhető, vagy beilleszthető a HTML-kódba! Egyszerű programjaink nem élnek ezzel a lehetőséggel, így nem fogjuk azonosítóval ellátni a szkripteket.

Egy dokumentumban a JavaScript és a VBScript felváltva is alkalmazható, ezért a SCRIPT-objektum *language* (nyelv) tulajdonságával adjuk meg, hogy melyiket használjuk:

```
<SCRIPT language = "JavaScript"> ... </SCRIPT> vagy
<SCRIPT language = "VBScript"> ... </SCRIPT>
```

---

<sup>9</sup> Az objektumalapú programozási nyelvekben nem használható az öröklődés.

A JavaScript helyett írhatjuk a JScript, a VBScript helyett pedig a VBS rövidítést.

A böngészők alapértelmezett nyelve a JavaScript, ennél a *language* tulajdonságot nem kell megadni. Ha egy dokumentum szkriptjeiben többféle nyelvet alkalmazunk, akkor mindig az utoljára meghatározott *language* tulajdonság érvényesül.

#### A VBScript szintaxisa

A VBScript nyelvben minden egyes utasítást külön sorba kell írni. Ha túl hosszú lenne az utasítás, ami zavarja az áttekinthetőséget, akkor egy szóköz és egy aláhúzás-jel után a következő sorban folytatható a kód. Erre később látunk példát.

A **3–1. példa** programja két VBScript utasítást tartalmaz. Az első értéket ad az ablak címsorának, a második a háttérszínt állítja sárgára:

```
<HEAD>
  <SCRIPT language = "VBS">
    document.title = "Ezt a szkript írta ide."
    document.bgColor = "yellow"
  </SCRIPT>
</HEAD>
```

Az eddigi példáktól eltérően a HTML-kódban nem szerepel a TITLE-objektum, a címsor szövege a szkriptben lévő utasítás hatására jelent meg. A programokban gyakran alkalmazunk ehhez hasonló, úgynevezett értékadó utasításokat, melyek bizonyos értékeket (itt a *document*-objektum *title* tulajdonságát) meghatároznak, vagy megváltoztatnak. A háttérszínt is egy értékadó utasítás adta meg. A későbbiekben az értékadó utasítások bonyolultabb formáival ismerkedünk meg.

Bár a VBScript nem különbözteti meg a kis- és nagybetűket, célszerű következetesen mindig ugyanabban a formában írni egy-egy kifejezést (például: *bgColor*). Így könnyebben felfedezhetjük a programban előforduló hibákat.

#### Szkriptek a HTML-kódban

Egy HTML-dokumentum tetszőleges számú SCRIPT-objektumot tartalmazhat, amelyek kódja egymástól elkülönülve helyezkedik el. A böngésző a weblap betöltésekor az egyes objektumok sorrendjében, egymás után hajtja végre a szkriptek utasításait, és építi fel a dokumentumot a HTML-kód szövegének, formázási előírásainak megfelelően. A **3–2. példában** a 3–1. példa utasításait két külön szkriptbe írjuk, mert a háttérszín beállításához a BODY-objektumot használjuk. Emlékeztetünk arra, hogy ha egy szkriptben megadjuk a *language* tulajdonságot, akkor az mindaddig érvényes, amíg meg nem változtatjuk. Ezért a második szkriptben már nem kell meghatározni a programnyelvet:

```
<HEAD>
  <SCRIPT language = "VBS">
    document.title = "Ezt a szkript írta ide."
  </SCRIPT>
</HEAD>
```



```
<BODY id = "Torzs" bgColor = "red">
  Milyen is a háttér színe?
  <SCRIPT>
    Torzs.bgColor = "yellow"
  </SCRIPT>
</BODY>
```

A böngésző a weblap felépítésénél végrehajtotta az első szkriptet, beírta a szöveget a címsorba. Utána létrehozta a BODY-objektumot piros háttérszínnel, majd a dokumentum szövegének kiírása után, a második szkript végrehajtásakor sárgára változtatta a hátteret. Mindezt azonban olyan gyorsan tette, hogy a pirosat nem is figyelhettük meg. A **3–3. példa** második szkriptjében először egy üzenet küldésével megállítjuk a végrehajtást, így már láthatjuk a változást:

```
<SCRIPT>
  window.alert("Még piros a háttér, de már nem sokáig!")
  Torzs.bgColor = "yellow"
</SCRIPT>
```

Az üzenetet most nem egy esemény váltotta ki, hanem a böngésző hozta létre a weblap megjelenítésekor, amikor a második szkript első utasításának végrehajtásához (*window.alert*) érkezett. A SCRIPT-objektumon belül az *alert* metódust már a későbbi eljáráshívásokhoz hasonlóan írjuk be, zárójelbe téve a paraméterét (az üzenetet).

## A szkriptek sorrendje

A szkriptek elhelyezésekor figyelniünk kell a végrehajtás sorrendjére. Amíg a böngésző a dokumentum összeállításánál nem hozott létre egy objektumot, addig nem hivatkozhatunk a tulajdonságaira. Ha a fenti példa szkriptjét a HEAD-be helyezzük, akkor hibaüzenetet kapunk, hiszen a végrehajtásánál még nem létezik a BODY (*Torzs*) objektum (**3–4. példa**):

```
<HEAD>
  <SCRIPT language = "VBS">
    window.alert("Még piros a háttér, de már nem sokáig!")
    Torzs.bgColor = "yellow"
  </SCRIPT>
</HEAD>
<BODY id = "Torzs" bgColor = "red">
  Milyen is a háttér színe?
</BODY>
```

A betöltés közben megjelent a *window.alert* üzenete, pedig még nem készült el a piros háttér sem, majd a *bgColor* értékadás hibás hivatkozása miatt hibaüzenetet ad a böngésző. Az Internet Explorer például az állapotsorban egy sárga táblás felkiáltójellel és a „Kész, de az oldal hibás” üzenettel jelzi a hibás kódot. Amennyire tudja, folytatja a dokumentum összeállítását, piros háttérrel (!) megjelenik a felirat.

A továbbiakban néhány kivételtől eltekintve arra fogunk törekedni, hogy csak egyetlen, a HEAD-ben elhelyezett szkriptobjektumot használjunk, mert így növelhetjük programjaink áttekinthetőségét.

## Eseménykezelés szkriptekkel

A DHTML-objektumokat bemutató fejezetben csak egy-két utasítást tartalmazó eseménykezelést tudunk megvalósítani a nyitó tagban. Több utasítás végrehajtásához eseménykezelő szkripteket alkalmazunk. A szkript nyitó tagjában meg kell adni az *event* (esemény) és a *for* (számára) tulajdonságot. Az *event* értéke jelöli ki az eseményt, a *for* pedig annak az objektumnak az azonosítóját, amelyhez az eseménykezelést hozzárendeltük:

```
<SCRIPT event = "eseménynév" for = "ObjektumAzonosító">
```

A **3–5. példában** a weblapon megjelenő parancsgombra való kattintással megváltoztatjuk a címsort, a háttér és a betűk színét, illetve a felirat igazítását. Ehhez készítünk egy parancsgombot, az objektumokat pedig azonosítóval látjuk el, hogy hivatkozni tudjunk rájuk:

```
<BODY id = "Torzs" bgColor = "red" text = "white">
  <H3 id = "Felirat">
    A címsor, a színek és az igazítás változtatása
  </H3>
  <INPUT type = "button" id = "Gomb" value = "Kattintson ide!">
</BODY>
```

Az eseménykezelő szkriptben végrehajtjuk a kívánt változtatásokat:

```
<HEAD>
...
<SCRIPT language = "VBS" event = "onclick" for = "Gomb">
  document.title = "Új cím"
  Torzs.bgColor = "yellow"
  Torzs.text = "blue"
  Felirat.align = "right"
</SCRIPT>
<HEAD>
```

Ily módon akárhány utasítást használhatunk az eseménykezelés során. Figyeljük meg, hogy a szkriptet a HEAD-be tettük, pedig a weblap összeállításánál itt még nincs definiálva a *Torzs*-, a *Gomb*- és a *Felirat*-objektum sem! A böngésző az eseménykezelő szkripteket a dokumentum megjelenítésénél nem hajtja végre, ezért nem kaptunk hibajelzést. Ilyenkor csak szintaktikus ellenőrzést végez, a végrehajtásra az esemény bekövetkeztekor kerül sor.

Ilyen szkriptet többet is elhelyezhetünk a HTML-kódban, minden objektum minden eseményéhez egyet-egyed. Egy eseménykezelő szkript azonban csak egyetlen objektum egyetlen eseményéhez tartozik. A későbbiekben megismerkedünk rugalmasabb eseménykezeléssel is.

## Megjegyzések a szkriptekben

Főleg a hosszabb programok áttekintését segíti, ha magyarázatokat helyezünk el a forráskódban. A későbbi módosítás lehetősége pedig eleve megköveteli ezeket a magyarázó megjegyzéseket. Alig néhány hét alatt teljesen elfelejtjük, hogy egy utasítás-

sorozat milyen célt szolgál, milyen logikát használtunk a program összeállítására, miért így, vagy miért úgy írtuk meg a kódot. A „visszafejtés” sok munkát igényel.

Programjainkat ezért megjegyzésekkel látjuk el. A VBScriptben a megjegyzéseket egy aposztrófjel (') után írjuk. A jeltől kezdve a sor végéig tartó szöveget az interpreter figyelmen kívül hagyja. Az aposztrófjel a sor elején is szerepelhet, ekkor az egész sor megjegyzésnek számít:

```
window.alert("Változik a háttér") ' felfüggeszti a kód végrehajtását
Torzs.backgroundColor = "yellow"
' Megváltoztatja a háttérszínt.
' Törekedjünk arra, hogy a megjegyzések ne legyenek
' semmitmondóak. A kód szöveges megismétlésének nincs értelme.
```

Mivel az aposztrófjel nem túl feltűnő, gyakran csillagokkal emelik ki:

```
'*****
'* kiemelt megjegyzés *
'*****
```

Az interpreter az aposztróf után kezdődő részt teljesen figyelmen kívül hagyja, így csillag helyett bármilyen más karaktert, karaktereket alkalmazhatunk.

A VBScript az aposztróf helyett megengedi a *Rem* (remark, megjegyzés) kulcsszó használatát is:

```
Rem Ez egy megjegyzés.
```

A *Rem* azonban utasításnak számít, tehát ha egy sor végére, egy utasítás után írjuk, akkor kettősponttal kell elválasztani tőle:

```
window.alert("Változik a háttér") : Rem Leállítja a végrehajtást.
```

A régebbi böngészők nem ismerik a SCRIPT-objektumot, ezért az utasítások végrehajtása helyett hibaüzenetet adnak, vagy végrehajtás helyett kiírják a weblapra a VBScript forráskódját. Ennek elkerülésére a szkript utasításait HTML-megjegyzésként szokták beírni a dokumentumba:

```
<SCRIPT>
<!--
    VBScript utasítások
-->
</SCRIPT>
```

Az interpreter a feldolgozásnál figyelmen kívül hagyja a HTML-megjegyzés jelölését, és végrehajtja az utasításokat. Ezt a szerkezetet mi nem alkalmazzuk, mert a legtöbb böngésző már felismeri a SCRIPT-objektumot.

Ne keverjük össze a HTML-kód és a VBScript megjegyzéseit. Az aposztróffal kezdődő sorok csak egy szkripten belül helyezkedhetnek el. A szkriptobjektumon kívül a böngésző a weblap szövegéhez hasonlóan megjelenítené őket az ablakban.

#### **A szkript hibáinak megjelenítése**

A böngészők általában nem túl feltűnő és eléggé szűkszavú módon utalnak a szkriptekben (sőt, az egész HTML-kódban) előforduló hibákra. Az Internet Explorer beállításainak segítségével bővebb hibaüzenetet kaphatunk. Ehhez az Eszközök menü Internet-beállítások parancsának ablakában a Speciális fület kell kiválasztani. A megjelenő listában keressük meg a Böngészés csoportot, és pipáljuk ki az „Üzenet megjelenítése minden parancsfájllibáról” jelölőnégyzetet (a csoport végén találjuk). Ha most újra betöltjük a hibás kódot tartalmazó példát, akkor a hibaüzenet ablakában válasszuk a részletek megjelenítését. Megkapjuk a hiba helyét (hányadik sor, hányadik karakter) és leírását. Hasonló beállítási lehetőséggel más böngészők is rendelkeznek.

Ha a Jegyzetömböt használjuk a forráskód szerkesztésére, akkor kissé nehéz viszszaérteni a hiba helyét, mert ez a program nem jelzi, hogy hányadik sorban villog a kurzor. A Microsoft webhelyéről letölthető a Microsoft Script Debugger, amely nagymértékben megkönnyíti a hibakeresést és a hibajavítást (lásd a függelék).

A Microsoft Script Debugger telepítése után egy hiba előfordulásakor a böngésző megkérdezi, hogy elindítsa-e a hibakeresőt. Ha az Igen gombot választjuk, akkor a Debugger megnyitja a fájlt, és sárgán jelzi a hibás sort. A Microsoft Script Debugger a HTML-kód szerkesztéséhez, szkriptek írásához, teszteléséhez, a hibák javításához számos lehetőséggel rendelkezik. Drága fejlesztőeszközök hiányában javasoljuk az Olvasónak a használatát.

#### **Szkriptszerkesztő programok**

Sok objektum vagy hosszú utasítássorozatok használata nehézkessé teszi a forráskód begépelését és áttekintését. Ezt a munkát számos szerkesztőprogram próbálja megkönnyíteni. A weblapkészítők, a szkriptek programozói az Interneten találnak kisebb-nagyobb tudású szerkesztőket a TextPad-től kezdve az UltraEditig, melyek különböző szolgáltatásokat nyújtanak az objektumok kezeléséhez, a szkriptek megírásához, a hibák felderítéséhez és javításához. A szerzők által is használt TextPad megtalálható a mellékelt CD TextPad mappájában.

A TextPad telepítése után, ha az Intézőben egy fájlra kattintunk a jobb egérgombbal, akkor kiválaszthatjuk a menüből a TextPad-del történő megnyitást. Nagy előnye, hogy a kód begépelésekor megtartja az előző sornak megfelelő behúzást, így könnyű biztosítani a strukturált elrendezést. A HTML Tags ablakból dupla kattintással helyezhetünk nyitó és záró tagokat a dokumentumba, de saját kódrészletekkel is bővíthetjük a listát. A shareware változat kipróbálás céljából tetszőleges ideig, és a funkciók korlátozása nélkül használható.

A Microsoft Office programcsomag tartalmazza a Microsoft Script Editort, amely már kifinomult fejlesztői eszközökkel rendelkezik. Használatát a függelékben ismertetjük.

## 3.2. Változók a programban

### Változók a VBScriptben

A programok adatokat dolgoznak fel. Adat lehet egy szám, egy karaktersorozat, de lehet jóval bonyolultabb is a megjelenési formája. Az adatokat a program futásának idejére tárolni kell a memóriában. Az adatok tárolásához változókat használunk.

Változó: névvel azonosított memóriaterület.

A szkriptek egy dokumentumban összesen 127 változót használhatnak. Ezt a korlátot azonban a későbbiekben ismertetésre kerülő tömbökkel és eljárásokkal jelentősen kibővíthetjük, így nem jelent gondot a programok írásánál. Önmagában véve sem kevés a 127 változó, eléggé összetett programot kell írni ahhoz, hogy elérjük ezt a határt.

### A változók neve

A VBScriptben a változónév az angol ábécé betűiből, számokból, illetve aláhúzásjelekből állhat. Betűvel kell kezdődnie, és legfeljebb 255 karakter hosszúságú lehet, bár ennél jóval rövidebbeket alkalmazunk. Nem használhatjuk változók elnevezésére a VBScript utasításait és egyéb fenntartott kifejezéseit, az úgynevezett kulcsszavakat. Felsorolásukat a CD-melléklet Dokumentumok mappájában lévő Kulcsszavak.htm fájlban találjuk. Különböző változóknak nem adhatjuk ugyanazt a nevet, hiszen így nem tudjuk a hivatkozásokban megkülönböztetni őket.

Aláhúzásjelekkel vagy nagybetűk alkalmazásával a neveket könnyebben olvashatóvá tehetjük. Helyes változónevek:

`Kutya_nev`, `KutyaNev`, `Varos12`

Hibás változónevek:

<code>KutyaNév</code>	nem tartalmazhat ékezetes magánhangzót,
<code>100Forint</code>	nem kezdődhet számmal,
<code>Erre.arra</code>	nem tartalmazhat pontot (vagy az aláhúzásjel kivételével más írásjelet).

A VBScript nem különbözteti meg egymástól a kis- és nagybetűket, a következő karaktersorozatok ugyanazt a változót jelölik:

`bodri`, `Bodri`, `boDRi`

Az áttekinthetőség érdekében azonban egy változó nevét mindig ugyanabban a formában írjuk le! Programjainkban a változóneveket nagybetűvel fogjuk kezdeni.

Törekedjünk olyan változónevek alkalmazására, melyek utalnak a tartalmukra. Értelmetlen karaktersorozatok, erőteljes rövidítések nehezen olvashatóvá teszik a kódot, ami megnöveli a hibalehetőségeket és megnehezíti az utólagos megértést, módosítást.

Ajánlott változónevek:

`Kutya_nev`, `KutyaNev`, `Magassag`, `X_koordinata`

Nem ajánlott, semmitmondó változónevek:

hfgm, mg5

A hosszabb változónevek ugyan megnehezítik a kód begépelését, de a program áttekinthetősége megéri a fáradságot. Ha mégis rövidítjük őket, akkor tartsuk meg a kezdetüket, és válasszunk kiejthető formákat, például:

max az mxm helyett egy érték maximumának jelölésére.

Megkönnyíti a gépelési hibák felismerését, ha a változónevek eleje különbözik:

x\_max és y\_max a maxx és maxy helyett.

Ne használjunk egymáshoz hasonló karaktereket (nagy I vagy kis l betű és 1-es számjegy, O betű és 0 számjegy, S betű és 5-ös számjegy). Vajon hogyan kell olvasni a NOS5 változónevet? „N, O, S, öt” vagy „N, nulla, öt, öt” vagy éppen „NOSS”?

#### A változók deklarálása

A programozási nyelvekben általában fel kell sorolni a használt változókat, meg kell adni a nevüket és azt, hogy milyen típusú adatot (szám, karaktersorozat stb.) foglalkoztatni. Ezen felsorolás, az úgynevezett deklaráció alapján foglalja le a helyet a fordítóprogram a memóriában. Könnyebb a gépelési hibák felismerése is, mert ha egy nem deklarált változót talál a kódban, akkor hibaüzenetet ad.

Deklaráció:

a programban használt változónevek felsorolása, esetenként a típus megadása.

A VBScript nyelvben nem kötelező deklarálni a változókat, és nem lehet megadni, hogy milyen típusú adatot tartalmaznak. Az interpreter a helyfoglalást akkor végzi el, amikor először találkozunk a változóval. A típus az első értékadásnál derül ki, és változhat a program végrehajtása során.

Bár nem szükséges, mégis ajánlatos deklarálni a változókat a VBScript programokban. A deklarálás helye általában a program eleje. A deklarálás a *Dim* (dimension, tulajdonképpen méret) kulcsszóból, majd a változónevek felsorolásából áll.<sup>10</sup> A neveket vesszővel kell egymástól elválasztani:

```
Dim Bodri_kutya, x_max, Magassag
```

A deklarációt több részre is bonthatjuk, és a programban bárhol (természetesen még a változó első használata előtt) elhelyezhetjük:

```
Dim Bodri_kutya
    utasítások
Dim x_max, Magassag
```

A továbbiakban a deklarációkat a program elejére tesszük.

---

<sup>10</sup> A dimenzió gyakran a tömbök indexeinek a számát jelenti (lásd később).

Mint említettük, a VBScriptben nem kötelező a változókat deklarálni. Ha az interpreter egy nem deklarált változóval találkozik, akkor is helyet foglal neki a memóriában. Ez a szabadság nehezen felderíthető hibákhoz vezet. Ha például egy *Szamlalo* nevű változót valahol csak *Szamlal*-nak írunk, akkor a programunk hibásan fog működni. Az *Option Explicit* utasítással kötelezővé tehetjük a programban a változók deklarálását. Alkalmazása esetén az interpreter hibaüzenetet ad, ha egy nem deklarált változót talál (például gépelési hiba esetén). Az utasítást a legelső szkript elejére kell beírni, és nem szerepelhet eseménykezelő szkriptekben. A továbbiakban mindig alkalmazzuk az *Option Explicit* utasítást.

## A változók típusai

A legtöbb programozási nyelvben az egyes változók csak egyféle típusú adatot (vagy számot, vagy karaktersorozatot stb.) tartalmazhatnak. Ez a megkötés többek között a helyfoglalást csökkenti, de némi védelmet nyújt a programozási hibák ellen is. A változó típusát a deklarációban kell megadni.

A VBScriptben egy változó értéke tetszőleges lehet, akár váltakozva vehet fel számértéket vagy karaktersorozatot. Ezért nem szerepel a típus a deklarációban. A későbbiekben azonban szükségünk lesz az alapvető típusok megkülönböztetésére. A legegyszerűbb típusú változók valamilyen számot, karaktersorozatot vagy például a *disabled* tulajdonságnál használt logikai értéket tartalmazhatnak.

Numerikus változó:	értéke valamilyen szám.
Szöveges változó:	értéke karakterek sorozata.
Logikai változó:	értéke igaz (True) vagy hamis (False).

Az 1245 vagy a -24,97 számokat például numerikus változóknak, a „Bodri” karaktersorozatot szöveges változóban tudjuk tárolni. A karaktersorozatot és magát a szöveges változót is gyakran sztringnek vagy karakterláncnak hívjuk.

Sztring, karakterlánc: karakterek sorozata, szöveges változó.

A sztringek megadásánál a karaktersorozatot idézőjelbe kell tenni. Ezért a sztringen belül nem használhatunk ugyanolyan idézőjelet, helyette aposztrófot alkalmazunk:

"Idézőjel 'használata' az idézőjelen belül."

A külső idézőjel nem tartozik a karakterlánchoz, kiírásakor nem fog megjelenni.

Ha ragaszkodunk az idézőjelhez a sztringen belül, akkor duplán kell kiírnunk. Az

"Idézőjel ""használata"" az idézőjelen belül."

karaktersorozat hatására a képernyőn a

Idézőjel "használata" az idézőjelen belül.

szöveg jelenik meg.

Egy szöveges változó bármilyen karaktert, akár számokat is tartalmazhat. Ne keverjük össze a numerikus változókat a számokból álló sztringekkel. A 23 és a 12 ösz-

szege természetesen 35, de ha „összeadjuk” (helyesebben összefűzzük) a „23” és a „12” sztringeket, akkor „2312”-t kapunk.

A VBScript interpretere a változó értéke alapján maga dönti el, hogy milyen típust használjon. A későbbiekben összetett típusokkal és finomabb felosztással is megismerkedünk.

#### Értékadás a változóknak

A fordítóprogram a deklaráció hatására helyet foglal a változónak a memóriában, de értéket nem rendel hozzá. Ezt az értékadó utasítások segítségével tehetjük meg. Az értékadó utasítások legegyszerűbb formája egy változónévből, egy egyenlőségjelből és a változónak adott értékből áll. Ha az érték karakterlánc, akkor idézőjelbe kell tenni:

```
Magassag = 50
Kutya_nev = "Bodri"
```

Az első utasítás hatására a `Magassag` névvel jelölt memóriaterület tartalma 50 lesz. Ha a programba leírjuk valahova ezt a változónevet, akkor az interpreter kiolvassa a memóriából az értékét, és azt használja fel az utasítás végrehajtásánál. A `Magassag + 10` kifejezés értéke például 60. A második utasítás a `Kutya_nev` jelű memóriaterületen a `Bodri` szót tárolja (az idézőjelek nem tartoznak hozzá a karakter-sorozathoz).

A változók értéke a memóriába típusuktól függő kódolással kerül be, de ezzel nem kell foglalkoznunk.

Vegyük észre, hogy az előző fejezetben az objektumok tulajdonságainak változtatásánál szintén értékadó utasításokat használtunk. Az értékadó utasításokban az egyenlőségjel bal oldalán változónév helyett egy objektum valamely tulajdonságneve is szerepelhet:

```
document.bgColor = "yellow"
```

#### A változók értékének megjelenítése

A **3–6. példában** értéket adunk a változóknak, a kutya nevét egy üzenetablakkal megjelenítjük, a magasságot pedig a `document.write` metódussal beírjuk a weblap szövegébe. A `document.write` metódusnál az eljáráshívásokhoz hasonlóan a paramétert (a weblapra kerülő szöveget) zárójelbe tesszük:

```
<HEAD>
  <SCRIPT language = "VBS">
    Option Explicit
    Dim Kutya_nev, Magassag
    Magassag = 50
    Kutya_nev = "Bodri"
    window.alert(Kutya_nev)
    document.write("Magasság: ")
    document.write(Magassag)
  </SCRIPT>
</HEAD>
<BODY>
</BODY>
```



Figyeljük meg, hogy az üzenet megjelenítésénél még üres a weblap! A *write* metódusok csak az OK gombra való kattintás után lépnek működésbe, mert a böngésző a kód feldolgozásánál egymás után hajtja végre az utasításokat.

A változókra való hivatkozáskor a változónevet nem tesszük idézőjelbe. A

```
document.write(Magassag)
```

utasítás nem a „Magassag” karaktersorozat, hanem a *Magassag* nevű változó memóriában őrzött értékét (itt éppen 50-et) jeleníti meg. Magyaratzként az előtte lévő `document.write("Magasság: ")` az idézőjelben lévő szöveget (sztringet) írja ki. Figyeljük meg, hogy a kettőspont után szóközt hagytunk a szöveg és a szám elválasztásához!

Az előző példában a *document.write* metódust használtuk. Már említettük, hogy ennek a metódusnak a weblap összeállítása után történő meghívása törli a teljes forráskódot. Itt azonban még nem fejeződött be a dokumentum elkészítése, hiszen a böngésző egymás után dolgozza fel a HTML-kód sorait. Ha megnézzük a böngészővel a forráskódot (Nézet/Forrás menüpont), akkor láthatjuk az eredeti tartalmat. A BODY azonban üres, mert a felíratot nem a törzs HTML-kódja foglalja magában, hanem a szkript végrehajtásakor jön létre.

## Megjelenítés az *innerText* tulajdonsággal

A változók értékét hozzárendelhetjük a HTML-kóddal előállított weblap valamely objektumának *innerText* tulajdonságához is, amely számot és karaktersorozatot tartalmazhat. A következő példában a kutya nevét és a magasság értékét egy-egy SPAN-objektumban jelenítjük meg az *innerText* tulajdonság megadásával.

A 3–7. példában először deklaráljuk a változókat, és értéket adunk nekik:

```
<HEAD>
  <SCRIPT language = "VBS">
    Option Explicit
    Dim Kutya_nev, Magassag
    Magassag = 50
    Kutya_nev = "Bodri"
  </SCRIPT>
</HEAD>
```

Majd elkészítjük a weblap szövegét az egyelőre üres SPAN-objektumokkal:

```
<BODY>
  <P>A kutya neve: <SPAN id = "Nev"></SPAN></P>
  <P>Magasság: <SPAN id = "Magas"></SPAN> &nbsp; centiméter</P>
```

Végül elhelyezünk egy újabb szkriptet, amely beleírja a SPAN-objektumokba a változók értékét:

```
<SCRIPT>
  Nev.innerText = Kutya_nev
  Magas.innerText = Magassag
</SCRIPT>
</BODY>
```

A második szkriptet a SPAN-objektumok után kellett elhelyeznünk, mert előtte még nem lehetett hivatkozni az *innerText* tulajdonságukra. A deklarációkat és a kezdeti értékadást a HEAD-objektumba írtuk, bár tehettük volna a második szkriptbe is.

A példában alkalmazott módszernek az a hátránya, hogy ha nem kerül azonnal végrehajtásra a szkript, akkor adatok nélkül jelenik meg a weblap szövege. Erről meggyőződhetünk, ha a második szkript elején kiírunk egy üzenetet (**3–8. példa**):

```
<SCRIPT>
  window.alert("Még hiányoznak az adatok!")
  Nev.innerText = Kutya_nev
  Magas.innerText = Magassag
</SCRIPT>
```

### Objektumok elrejtése és megjelenítése

Sokszor zavaróan hat, ha az előző példához hasonlóan a magyarázó szöveg hiányosan, adatok nélkül olvasható a weblapon. Máskor is szükségünk lehet bizonyos objektumok elrejtésére, majd megjelenítésére. Ezt az objektum *style* tulajdonságával szabályozhatjuk. A *style* használatát már az előző fejezetben ismertettük. Az objektum megjelenését a *visibility* (láthatóság) stíluselem határozza meg. Értéke lehet:

<i>hidden</i> (rejtett):	az objektum nem látható
<i>visible</i> (látható):	az objektum megjelenik a weblapon.

A stíluselem nevét és értékét idézőjelek között, kettősponttal elválasztva kell megadni:

```
style = "visibility: hidden"
```

Az előző feladatban az adatok kiírása előtt a szöveget a bekezdés elrejtésével tudjuk eltüntetni. A későbbi megjelenítés miatt a bekezdést azonosítóval kell ellátni:

```
<P id = "Adatok" style = "visibility: hidden">
```

Az *innerText* tulajdonságok értékadása után a *visibility* stíluselemet *visible*-re kell állítani. Ennek szintaxisa a szkriptben kissé eltér a nyitó tagban használt formától:

```
Adatok.style.visibility = "visible"
```

A teljes kódot a **3–9. példa** tartalmazza. Töltsük be a böngészőbe, és figyeljük meg, hogy az üzenet kiírásakor a weblapon még nem látszik semmilyen szöveg! A bekezdés megjelenítését csak az adatok beírása után engedélyezzük.

A továbbiakban az éppen tárgyalt ismeretekre koncentrálunk, mondandónkat minél rövidebb és egyszerűbb példákkal szeretnénk illusztrálni. Ezért nem alkalmazzuk a *visibility* stíluselem beállítását. Az Olvasónak azonban javasoljuk, hogy az adatok és a magyarázó szövegek esztétikus megjelenítéséhez használja fel az itt bemutatott fogást!

### A HTML-kód előállítás

Az előző példában az adatok magyarázó szövegét a HTML-kód adta. A szkript utasításai ebbe az előre elkészített kódba írták be a megfelelő értékeket. A **3–10. példában** a *document.write* metódussal magát a HTML-kódot állítjuk elő. Ekkor nincsen szükségünk a SPAN-objektumra:

```

<HEAD>
  <SCRIPT language = "VBS">
    Option Explicit
    Dim Kutya_nev, Magassag
    Magassag = 50
    Kutya_nev = "Bodri"
    document.write("<P>A kutya neve: ")
    document.write(Kutya_nev)
    document.write("<BR>")
    document.write("Magasság: ")
    document.write(Magassag)
    document.write(" centiméter</P>")
  </SCRIPT>
</HEAD>
<BODY>
</BODY>

```

A szkriptet a HEAD-be tettük, a BODY-objektum teljesen üres. A dokumentum betöltésekor az interpreter egyesével végrehajtja a szkriptben szereplő utasításokat, és előállítja a böngésző számára a megjelenítendő HTML-kódot.

## Karakterláncok összefűzése

Ha a HTML-kód hosszú, akkor túl sok *document.write* utasítást kellene beírni a szkriptbe. Helyette megtehetjük, hogy összefűzött sztringeket adunk meg paraméterként. A karaktersorozatok összefűzése egyszerűen azt jelenti, hogy egymás után írjuk őket. Ehhez a sztringek közé & jelet kell írni:

*Művelet:*

"A kutya neve: " & "Bodri"

*Eredmény:*

A kutya neve: Bodri

Az összefűzésnél változóneveket is alkalmazhatunk. Ekkor az interpreter a változó értékét fűzi hozzá a sztringhez:

*Művelet:*

"A kutya neve: " & Kutya\_nev

*Eredmény:*

A kutya neve: Bodri

Figyeljük meg, hogy a változó nevét nem tettük idézőjelbe, különben a „Bodri” szó helyett a „Kutya\_nev” karaktersorozat jelent volna meg!

Egy sztringhez számot, illetve számot tartalmazó numerikus változót is fűzhetünk. Ekkor az interpreter a számjegyeket sorra hozzáírja a karaktersorozathoz:

*Művelet:*

"Magasság: " & 50  
 "Magasság: " & Magassag

*Eredmény:*

Magasság: 50  
 Magasság: 50

A sztringek összefűzését gyakran konkatenációnak nevezik.

Konkatenáció: sztringek összefűzése (egymás után írása).

A 3–10. példa kódját a sztringek összefűzésének alkalmazásával rövidebben tudjuk felírni (**3–11. példa**):

```
document.write("<P>A kutya neve: " & Kutya_nev & "<BR>")
document.write("Magasság: " & Magassag & " centiméter</P>")
```

A kód begépelésénél figyeljünk az idézőjelek és zárójelek használatára!

Megtehattuk volna azt is, hogy csak egyetlen *document.write* utasítást alkalmazunk. Így azonban túl hosszú sort kapnánk a forráskódban. Igyekezzünk minél áttekinthetőbb kódot írni. Ezzel csökkentjük a hibalehetőségek számát.

#### Változók értékének beolvasása

A változók értékét gyakran a felhasználótól kérjük be. A dokumentum felépítése közben ezt az *InputBox* függvénnyel érjük el. (A függvényekkel a későbbiekben ismerkedünk meg részletesebben.) Az *InputBox* megjelenít egy párbeszédablakot a képernyőn, melynek szövegmezőjébe tetszőleges értéket lehet beírni. A begépelte karakter-sorozatot felhasználhatjuk a változók értékadásánál.

Az *InputBox* után zárójelben megadjuk a felhasználót tájékoztató üzenetet és a párbeszédablak címét. Ezeket a metódusoknál felsorolt adatokhoz hasonlóan paramétereknek nevezzük:

```
InputBox(Üzenet, Ablakcím)
```

Az alábbi utasítás hatására megjelenik egy ablak „A név beolvasása” címmel, illetve az „Írja be a nevét!” üzenettel. Az ablakba egy *OK* és egy *Mégse* gomb is kerül:

```
InputBox("Írja be a nevét!", "A név beolvasása")
```

A felhasználó által begépelte adatot egy értékadó utasítással adhatjuk át valamelyik változónak:

```
Nev = InputBox("Írja be a nevét!", "A név beolvasása")
```

**A 3–12. példa** bekéri a felhasználó nevét, majd udvariasan köszön neki:

```
Option Explicit
Dim Nev
document.write("<H3>Köszönő weblap</H3>")
Nev = InputBox("Írja be a nevét:", "A név beolvasása")
document.write("<P>Üdvözlöm, kedves " & Nev & "!</P>")
```

Az *OK* gombra való kattintás után a szövegmező értéke a *Nev* változóba került, amit a továbbiakban tetszőlegesen felhasználhatunk. Figyeljük meg, hogy az *InputBox* ablaka a „Köszönő weblap” felirat kiírása után jelent meg, pontosan abban a sorrendben, ahogy az utasítások elhelyezkednek a szkriptben!

## Az InputBox kezdőértéke

Az *InputBox* paraméterei között megadhatjuk a szövegmező kezdőértékét is, amit a felhasználó elfogadhat, vagy szükség szerint megváltoztathat:

```
InputBox(Üzenet, Ablakcím, Kezdőérték)
```

A **3–13. példában** kezdőértéket adunk az *InputBox*-nak. Ez jó gyakorlat annak megelőzésére, hogy a felhasználó üresen hagyja a szövegmezőt, ami hibákhoz vezethet a program végrehajtásánál:

```
Option Explicit
Dim Magassag
Magassag = InputBox("Írja be a magasságot (cm):", _
    "A magasság beolvasása", 50)
document.write("<P>A magasság: " & Magassag & " cm</P>")
```

Az *InputBox* paraméterei nem fértek ki egy sorban, ezért az első sor végén egy szóköz és egy aláhúzásjel szerepel. Mint tudjuk, így lehet folytatni egy VBScript utasítást a következő sorban.

Az *InputBox* kezdőértéke numerikus változó és sztring is lehet. A szövegmező tartalmát a böngésző kijelöli, azaz kék háttérrel jeleníti meg. Ha elkezdünk begépelni egy új értéket, akkor a kezdőérték automatikusan törlődik.

Az interpreterrel feltétlenül közölni kell, hogy mit tegyen a felhasználó által beírt adattal. Az *InputBox* tehát csak értékadó utasításban szerepelhet. Ha önálló utasításként alkalmazzuk, akkor hibaüzenetet kapunk.

Az *InputBox*-nál az üzenetet kötelező megadni, de az ablak címe és a kezdőérték elmaradhat:

```
Magassag = InputBox("Írja be a magasságot (cm):")
```

A kezdőérték hiánya azonban futási hibákhoz vezethet, hiszen a felhasználó akkor is rákattinthat az OK gombra, ha nem írt be semmit. Az ablak címe pedig egységessé teszi a megjelenítést. Ezért elhagyásukat nem javasoljuk.

Megjegyezzük, hogy a *window*-objektum *prompt* metódusa az *InputBox*-hoz hasonló szerepet játszik, de jóval kevesebb paramétert lehet megadni.

## Beolvasás szövegmezőkkel

Az *InputBox* meglehetősen rugalmatlanul használható adatok bevitelére. Több érték beolvasása kényelmetlen, az OK gombra való kattintás után a javítás csak a dokumentum frissítésével végezhető el. A weblapon elhelyezett szövegmezők sokkal barátságosabb adatbevitelt tesznek lehetővé. A szövegmezők *value* tulajdonsága felhasználható a változók értékadásánál.

Az értékadást valójában a parancsgombokhoz rendelt eseménykezelés során hajtjuk végre. A **3–14. példa** a beolvasás után megjeleníti a felhasználó nevét, illetve testmagasságát. Ehhez két szövegmezőt, egy parancsgombot és egy bekezdésobjektumot helyezünk el a weblapon:

```
Név:<BR>
<INPUT type = "text" id = "NevBeolvas" value = "Név"><BR>
Magasság:<BR>
<INPUT type = "text" id = "MagassagBeolvas" value = 0> cm<BR>
<INPUT type = "button" id = "Gomb" value = "Kattintson ide!"><BR>
<P id = "Adatok"></P>
```

A beolvasott adatokat változóban tároljuk, és az üres bekezdésobjektumban jelenítjük meg. Ehhez deklarálunk egy *Nev* és egy *Magassag* nevű változót:

```
<SCRIPT language = "VBS">
  Option Explicit
  Dim Nev, Magassag
</SCRIPT>
```

A parancsgombhoz eseménykezelő szkriptet rendelünk:

```
<SCRIPT event = "onclick" for = "Gomb">
```

Az esemény bekövetkeztekor eltesszük a változóba a szövegmezők tartalmát:

```
Nev = NevBeolvas.value
Magassag = MagassagBeolvas.value
```

Az üres bekezdés szövegének előállításához összefűzzük a változók értékét és a magyarázatot:

```
Adatok.innerText = "Üdvözlöm, kedves " & Nev & "! " & _
                  "Magassága: " & Magassag & " cm."
```

Az idézőjelben lévő szöveg változtatás nélkül jelenik meg a weblapon, az idézőjel nélküli változóneveket az interpreter az értékükkel helyettesíti. Figyeljünk az elválasztó szóközökre az idézőjeleken belül!

A szövegmezőkkel történő beolvasás előnye, hogy a beírt értékek utólag is javíthatók, és tetszőlegesen megismételhető az adatbevitel. A továbbiakban általában ezt a módszert alkalmazzuk.

### Input és output a VBScriptben

Az eddigiekben már többféle beviteli és kiviteli lehetőséggel megismerkedtünk. Szükségesnek tartjuk összefoglalni ezeket az eszközöket.

*Input:*

- az *InputBox* függvénnyel. Akkor használjuk, ha csak egy-két értéket kell bekérni, vagy ha a weblap összeállításához van szükségünk adatokra.
- szövegmezőkkel. Kényelmes, megismételhető és könnyen módosítható adatbevitelt tesznek lehetővé. A beírt értékek feldolgozását valójában egy parancsgombra való kattintás indítja el, végrehajtva a hozzárendelt eseménykezelő szkript utasításait.

*Output:*

- a *window.alert* metódussal (vagy később az *MsgBox* függvénnyel). Főleg rövid, figyelmeztető üzeneteket küldünk segítségével a felhasználónak.
- a *document.write* metódussal. A HTML-kód összeállításánál használjuk, amikor a készülő weblap egyes elemeit is egy szkript hozza létre. A teljes dokumentum megjelenítése után alkalmazott *document.write* törli a kódot!
- SPAN-, DIV- vagy bekezdés (P) objektumokkal. Az egyik leggyakoribb adatkiviteli módszer. A HTML-kód alapján megjelenített weblap üres SPAN-, DIV- vagy bekezdésobjektumokat tartalmaz, melyek *innerText* vagy *innerHTML* tulajdonságát a szkriptek adják meg, illetve módosítják. A felsoroltak helyett bármilyen más objektumot felhasználhatunk, amely alkalmas a szövegek megjelenítésére. Megváltoztathatjuk akár egy parancsgomb feliratát is.

A klasszikus, karakteres felületen dolgozó programok először beolvassák az adatokat, majd a számítások elvégzése után közlik az eredményeket, és befejezik a működést. Jó esetben megkérdezik, hogy új értékekkel is számoljanak-e. A VBScripthez hasonló eseményvezérelt programok a képernyőn megjelenő objektumok és a Windows grafikus felhasználói felületének segítségével sokkal kényelmesebb vezérlést tesznek lehetővé. Működésüket eseményekkel irányíthatjuk. Az ilyen programok kódja legnagyobb részben eseménykezelő szkriptekből áll.

### 3.3. Aritmetikai műveletek

#### Numerikus változók

A továbbiakban egyelőre elbúcsúzunk a karaktersorozatoktól, és a numerikus változók kezelésével ismerkedünk meg. A numerikus változók tetszőleges számot tartalmazhatnak. Értékük lehet pozitív, negatív, egész vagy tört. A törteknél a szkriptekben az angolszász gyakorlatnak megfelelően tizedespontot kell írni:

3,14 helyett 3.14

A weblapon a szövegmezőkben az operációs rendszer területi beállításainál megadott jelet, a magyar nyelvű Windows esetén általában tizedesvesszőt kell használni. Ez némi keveredést okozhat, de ha összecserejlük a kétféle karaktert, akkor úgymint hibajelzést kapunk.

A túl nagy vagy túl kicsi abszolút értékű számokat úgynevezett lebegőpontos formában adhatjuk meg. A lebegőpontos alakban egy szám és egy 10 hatvány szorzata szerepel. A 10-es alapot egy E betű jelzi:

*A szám értéke:*

$5 \cdot 10^7$   
 $-3 \cdot 10^{-5}$   
 $278,6 \cdot 10^{144}$

*Lebegőpontos forma:*

5E7  
-3E-5  
278.6E144

A 10 kitevője +308 és -324 közé eshet. Figyeljünk arra, hogy az E elé nem teszünk szorzásjelet!

A numerikus változók kódolásának módját az interpreter maga dönti el. Ezzel egyelőre nem kell foglalkoznunk.

A **3–15. példában** értéket adunk néhány változónak, majd kiíratjuk őket a *document.write* metódussal. A görög  $\pi$  betűt entitásának segítségével jelenítjük meg, ami egyszerűen: *&pi;*.

```
Option Explicit
Dim Pi, Nagy, Kicsi, Negativ
Pi = 3.14159265
Nagy = 789E300
Kicsi = 123E-308
Negativ = -111E-77
document.write("A &pi; értéke: " & Pi & "<BR>")
document.write("Nagyon nagy szám: " & Nagy & "<BR>")
document.write("Nagyon kicsi szám: " & Kicsi & "<BR>")
document.write("Egy negatív szám, negatív kitevővel: " & _
    Negativ & "<BR>")
```

Töltsük be a példát a böngészőbe, és nézzük meg, milyen írásjellel választja el a  $\pi$  tizedesjegyeit az egészrésztől. Figyeljük meg, hogy a nagy és kis abszolút értékű számokat a böngésző normálalakban írta ki!

## Operátorok

A programok nem csak tárolják az adatokat, hanem műveleteket is végeznek velük. A számtani alpműveleteket és a hatványozást aritmetikai műveleteknek nevezzük. A továbbiakban másfajta (például logikai) műveletekkel is megismerkedünk. A műveleti jeleket a programozásban operátoroknak hívják.

Operátor: műveleti jel.

A VBScript aritmetikai operátorai:

Név:	Jel:	Példa:	Eredmény:
Összeadás:	+	3.6 + 5.9	9.5
Kivonás:	–	2 – 7	-5
Szorzás:	*	8 * 4	32
Osztás:	/	16 / 5	3.2
A maradékos osztás hányadosa:	\	20 \ 7	2
A maradékos osztás maradéka:	Mod	20 Mod 7	6
Hatványozás:	^	2 ^ 5	32

A fenti műveleteket változók használata nélkül a **3–16. példa** szemlélteti. Figyeljük meg, hogy a forráskódba nem a végeredményt, hanem a műveletet írtuk be:

```
document.write("3,6 + 5,9 = " & 3.6 + 5.9)
```

A weblapon már az eredmény jelenik meg.



## A műveletek sorrendje

Az előző példa *document.write* metódusának paraméterét magyarázó karaktersorozatok és aritmetikai műveletek alkották. A böngésző a weblap összeállításánál először kiszámítja a művelet eredményét, majd hozzáfűzi az előtte szereplő sztringhez. Ezt a sorrendet az úgynevezett precedencia-szabály határozza meg.

Precedencia: a műveletek elvégzésének előírt sorrendje.

A VBScript precedencia-szabálya az aritmetikai műveletekre vonatkozóan a hatványozástól eltekintve megfelel a matematikában szokásos sorrendnek:

ellentettképzés	-
hatványozás	^
szorzás, osztás	*, /
maradékos osztás hányadosa	\
maradékos osztás maradéka	Mod
összeadás, kivonás	+, -
sztringek összefűzése	&

Először a listában feljebb álló műveleteket kell elvégezni. Ha azonos precedenciájú műveletek vannak egymás mellett, akkor balról jobbra történik a kifejezés kiértékelése. Mivel az aritmetikai műveletek mellett előfordulhat a sztringek összefűzése is, ezért ez a művelet szintén helyet kapott a precedencia-táblázatban. Legalul áll a sorban, tehát az interpreter legutoljára végzi el.

Számoljunk utána az alábbi műveletek eredményének:

$5 + 4 * 6$	$= 5 + (4 * 6)$	$= 29$
$10 + 6 / 3 - 1$	$= 10 + (6 / 3) - 1$	$= 11$
$10 / 2 - 6 / 3$	$= (10 / 2) - (6 / 3)$	$= 3$
$12 * 3 ^ 2 / 2 ^ 2$	$= (12 * (3 ^ 2)) / (2 ^ 2)$	$= 27$
$5 * 7 \text{ Mod } 10 \setminus 3$	$= (5 * 7) \text{ Mod } (10 \setminus 3)$	$= 2$
$5 * 7 \setminus 3 \text{ Mod } 10$	$= ((5 * 7) \setminus 3) \text{ Mod } 10$	$= 1$
$3 \& 9 * 5 \& 6$	$= 3 \& (9 * 5) \& 6$	$= 3456 (!)$

Az utolsó példában az interpreter először elvégzi a szorzást, melynek eredménye 45. Majd balról hozzáfűz egy 3-ast, jobbról pedig egy 6-ost. Így alakul ki a 3456 eredmény.

A precedenciában előírt sorrend megváltoztatására a matematikában megszokott zárójeleket használjuk:

```
(5 + 4) * 6 = 54
(10 + 6) / (3 - 1) = 8
(10 / 2 - 6) / 3 = -1/3
((12 * 3) ^ 2 / 2) ^ 2 = 419 904
5 * ((17 Mod 10) \ 3) = 10
5 * (17 \ (3 Mod 10)) = 25
```

Felhívjuk a figyelmet arra, hogy a hatványozás és az ellentett képzés sorrendje eltér a szokásostól. Mi lesz az eredménye a  $-3^2$  műveletnek? Mivel az ellentett képzését

előbb kell elvégezni, mint a hatványozást, ezért +9-et kapunk. Bár az interpreter számára egyértelmű a sorrend, használjunk itt is zárójelet:  $(-3)^2$ . Ha a  $3^2$  ellentettjét akarjuk meghatározni, akkor *kötelező* a zárójel használata:  $-(3^2) = -9$ .

A negatív előjel precedenciája miatt alkalmazhatjuk az alábbi kifejezéseket is:

$2 * -3, 5 + -6$  stb.

Célszerű azonban zárójelek használatával a matematikában szabályos formát alkalmazni:

$2 * (-3), 5 + (-6)$  stb.

## A műveletek operandusai

A műveleteket operandusokkal végezzük el.

Operandus: olyan érték, amivel műveletet végzünk.

Az operandus lehet szám, változónév (beleértve egy objektum tulajdonságnevét) vagy valamilyen más műveletsorozat eredménye:

$3 + 5, \text{Magassag} * 2, \text{Tavolsag} / 6 - 91$  stb.

Az utolsó példában két művelet, egy osztás és egy kivonás szerepel. Az osztás operandusai a *Tavolsag* nevű változó és a 6, a kivonás operandusai az osztás eredménye és a 91.

A negatív előjelet, illetve egy kifejezés ellentettjét is a  $-$  jel jelöli:

$-37, -\text{Magassag}$

## Kifejezések

Az operátorokból és operandusokból kifejezéseket képezhetünk.

Kifejezés: operandusok és operátorok szabályos sorozata.

Az operandusok között szerepelhetnek a későbbiekben ismertetésre kerülő függvényhívások is.

A kifejezésekben a műveletek precedenciáját zárójelek segítségével változtathatjuk meg. Ha csak aritmetikai műveleteket végzünk, akkor aritmetikai kifejezést kapunk. Aritmetikai kifejezés például:

$3 * (8 - (\text{Magassag} + 6)) / (\text{Tavolsag} + 2.9)^2$

A továbbiakban egyetlen számot vagy változónevet szintén kifejezésnek fogunk tekinteni. Ezzel egyszerűsítjük a fogalmazást. Ahol kifejezésről beszélünk, ott szerepelhet egy bonyolult képlet, de akár egyetlen konstans is.

A kifejezések eredményét értékadó utasítások segítségével írhatjuk be egy változóba, az értékadó utasítások jobb oldalán kifejezések szerepelnek. A

$\text{Magassag} = 5 + 4 * 6$

utasítás hatására például a *Magassag* nevű változó értéke 29 lesz, előző értéke pedig elvész a számunkra.

A **3–17. példa** rákérdez egy kör sugarára, majd a *Kerulet* és a *Terulet* azonosítójú SPAN-objektumok segítségével megjeleníti az adott sugarú kör kerületét és területét:

```
<BODY>
  <H3>Kör kerületének és területének számítása</H3>
  A kör kerülete = <SPAN id = "Kerulet"></SPAN><BR>
  A kör területe = <SPAN id = "Terulet"></SPAN>
  <SCRIPT>
    Option Explicit
    Dim Sugar
    Sugar = InputBox("Írja be a kör sugarának értékét!", _
                     "Kör kerülete és területe", 1)
    Kerulet.innerText = Sugar * 2 * 3.14159265
    Terulet.innerText = Sugar^2 * 3.14159265
  </SCRIPT>
</BODY>
```

A szkriptet a SPAN-objektumok után kellett elhelyeznünk, hogy hivatkozhattunk rájuk. Újabb számítást a weblap frissítésével tudunk kezdeményezni.

## A kifejezések egyszerűsítése

A szkriptek értékadó utasításaiban nagyon bonyolult kifejezések is szerepelhetnek. A húsvét időpontjának meghatározásához például ki kell számolni a következő matematikai kifejezés értékét (a \ és a Mod a maradékos osztást, illetve a hányados meghatározását jelöli):

$$\begin{aligned}
 x &= \{19 \cdot (Ev \bmod 19) + Ev \setminus 100 - (Ev \setminus 100) \setminus 4 - \\
 &\quad - [Ev \setminus 100 - (Ev \setminus 100 + 8) \setminus 25 + 1] \setminus 3 + 15\} \bmod 30 \\
 y &= \{32 + 2 \cdot (Ev \setminus 100) \bmod 4 + 2 \cdot (Ev \bmod 100) \setminus 4 - x - (Ev \bmod 100) \bmod 4\} \bmod 7 \\
 z &= (Ev \bmod 19 + 11 \cdot x + 22 \cdot y) \setminus 451
 \end{aligned}$$

ahol *Ev*-vel jelöltük az évszámot. Ekkor a hónapot és a napot az alábbi képletekből kapjuk meg:

$$\begin{aligned}
 \text{hónap} &= (x + y - 7 \cdot z + 114) \setminus 31 \\
 \text{nap} &= (x + y - 7 \cdot z + 114) \bmod 31 + 1
 \end{aligned}$$

Láthatjuk, hogy a képletek még így, több lépésben megadva is eléggé bonyolultak. Vegyük észre, hogy ugyanazt a mennyiséget, például  $Ev \setminus 100$  értékét sokszor ki kell számolni! Ezzel lassítjuk a program futását. Gyűjtsük ki, és tároljuk segédváltozókban a többször előforduló részeredményeket:

```
A = Ev Mod 19
B = Ev \ 100
C = Ev Mod 100
```

Így:

```
X = (19*A + B - B\4 - (B - (B + 8)\25 + 1)\3 + 15) Mod 30
Y = (32 + 2*(B Mod 4) + 2*(C\4) - X - C Mod 4) Mod 7
Z = (A + 11*X + 22*Y)\451
```

Az utolsó lépéshez pedig bevezetjük a *D* segédváltozót:

```
D = X + Y - 7*Z + 114
```

A hónapot és a napot a fenti két maradékos osztással határozhatjuk meg.

Egyetlen mennyiséget sem számoltunk ki kétszer, és a képlet is sokkal áttekinthetőbbé vált. Jutalmul a **3–18. példa** meghatározza ezeket az értékeket és megadja a hús-vét időpontját egy tetszőleges, 1582 utáni évben.<sup>11</sup>

Bonyolult kifejezéseket akkor is célszerű több lépésben kiszámolni, ha nincsenek közös részek. Mindenképpen áttekinthetőbbé válik a program, és kevesebb hibát követünk el a megírásánál.

#### Konstansok deklarálása

A kör területét és területét számoló példában kétszer is be kellett írunk a  $\pi$  értékét néhány tizedesjegyre. Ez kényelmetlen módszer, és hibalehetőségeket rejt magában. A többször szereplő mennyiségeket célszerű külön deklarálni. Ha a program elején megadjuk az értéküket, akkor a későbbiekben egyszerűbbé válik a felhasználásuk.

A VBScript a *Const* utasítással lehetővé teszi az ilyen mennyiségek, az úgynevezett konstansok deklarációját és értékadását:

```
Const Név = érték, Név = érték, ...
```

A konstansok nevére ugyanazok a szabályok vonatkoznak, mint a változókéra.

Konstansként nem csak numerikus változót, hanem sztringet is deklarálhatunk. A *Const* utasítás tetszőleges számszor és helyen megismétlődhet a programban, de célszerű a szkript elejére tenni. A következő utasítások konstansokat deklarálnak:

```
Const Pi = 3.14159265
Const Valuta = "Euró", Arfolyam = 235
```

A konstansokat a programban a változókhoz hasonlóan értékadó utasításokban használhatjuk. Az interpreter azonban nem engedi megváltoztatni egy konstans értékét. Ezzel csökkenti a hibalehetőségeket. A következő kód végrehajtásakor például hibaüzenetet kapunk:

```
Const Valuta = "Euró", Arfolyam = 235
Arfolyam = 245
```

Nem lehet ugyanaz a neve egy konstansnak és egy változónak, így a konstansnevek nem szerepelhetnek a *Dim* utasításban.

---

<sup>11</sup> A jelenleg használatos Gergely-naptár bevezetésének éve.

## Előre deklarált konstansok

A VBScript több előre deklarált konstanssal is rendelkezik. Ezek neve vb-vel kezdődik. A **3–19. példa** bemutatja a soremelést jelentő *vbNewLine* (új sor) konstans használatát,<sup>12</sup> amellyel többsoros üzeneteket állíthatunk össze:

```
window.alert("Első sor. " & vbNewLine & "Második sor.")
```

A konstans nevét a változónevekhez hasonlóan nem kell idézőjelbe tenni.

A **3–20. példában** a *vbNewLine* segítségével elérjük, hogy az *InputBox* üzenete közvetlenül a szövegmező fölé kerüljön. Rövid szöveg esetén többször alkalmazzuk:

```
Sugar = InputBox(vbNewLine & vbNewLine & vbNewline & vbNewline & _  
vbNewLine & "Írja be a kör sugarának értékét!", _  
"Kör kerülete és területe", 1)
```

A párbeszédablakok üzenet paramétere olyan sztring, melynek a soremeléssel együtt minden karaktere kikerül a képernyőre. A weblap szövegénél azonban a tagoló karakterek nem jelennek meg. Így a

```
document.write("Első sor." & vbNewLine & "Második sor.")
```

hatására csak a forráskódban jön létre soremelés, a képernyőn a két mondat ugyanabba a sorba kerül. A weblapon történő soremeléshez a forráskódba a `<BR>` objektumot kell beilleszteni.

## Az aritmetikai műveletek alkalmazása

Az aritmetikai műveletek és az értékadó utasítások segítségével már nagyon hasznos programokat írhatunk. Készítsünk egy olyan szkriptet, amely forintot számít át euróra az aktuális árfolyam alapján!

A **3–21. példában** az átszámítandó forintértéket egy *ForintBe* azonosítójú szövegmezőben kérjük be:

```
Írja be a forint értéket:<BR>  
<INPUT id = "ForintBe" type = "text" value = 0>
```

Az eredményt egy SPAN-objektumban helyezzük el:

```
forint = <SPAN id = "Euro"></SPAN> &nbsp; euró.<BR>
```

A számítást egy parancsgomb *onclick* eseménykezelője végzi el, amely a beírt forint értéket elosztja az árfolyammal, és az eredményt megjeleníti a SPAN-objektumban:

```
<INPUT type = "button" value = "Átszámítás"  
onclick = "Euro.innerText = ForintBe.value / 232.5">
```

Ennél a megoldásnál nem is használtunk SCRIPT-objektumot, mert a „program” bekerült az *onclick* tulajdonság értékebe.

---

<sup>12</sup> Értéke az operációs rendszertől függően `Chr(10)` vagy `Chr(10) & Chr(13)`.

A **3–22. példában** az árfolyamot konstansként tároljuk:

```
<SCRIPT language = "VBS">
  Const Arfolyam = 232.5
</SCRIPT>
```

Az eseménykezelést külön szkripttel végezzük. Ehhez a parancsgombot azonosítóval kell ellátni, de nincs szükség az *onclick* tulajdonságára:

```
<INPUT id = "Szamitas" type = "button" value = "Átszámítás">
```

Az eseménykezelő szkript elvégzi az átszámítást, és értéket ad az *innerText* tulajdonságnak:

```
<SCRIPT event = "onclick" for = "Szamitas">
  Euro.innerText = ForintBe.value / Arfolyam
</SCRIPT>
```

Az árfolyamkonstanst az eseménykezelő szkriptben is deklarálhattuk volna, csak az áttekinthetőség kedvéért emeltük ki.

Programunk szépséghibája, hogy esetenként nagyon sok tizedesjegyet jelenít meg. Ennek szabályozására hamarosan sort kerítünk.

A **3–23. példa** egy harmadik megoldást is bemutat az átszámításra, amelyben egy újabb szövegmező segítségével az árfolyam értékét a felhasználótól kéri. Az árfolyam napról-napra változhat, és kényelmetlen módszer lenne mindig megváltoztatni a kódot.

Mindkét szövegmezőnek kezdőértéket adunk, hogy ne vezessen hibás működéshez, ha a felhasználó a kitöltésük nélkül kattint a parancsgombra:

```
Írja be az árfolyamot:<BR>
1 euró = <INPUT id = "ArfolyamBe" type = "text"
           value = 232,5> forint.<BR>
Írja be a forint értéket:<BR>
<INPUT id = "ForintBe" type = "text" value = 0>
```

Az árfolyamot beolvasó szövegmező kezdeti értékénél a területi beállításoknak megfelelő tizedesjelet kell használni, a magyar nyelvű Windowsnál általában vesszőt.

Az átváltásnál az *onclick* eseménykezelőben az *Arfolyam* konstans helyett most az *ArfolyamBe* azonosítójú szövegmező *value* tulajdonságával osztunk:

```
Euro.innerText = ForintBe.value / ArfolyamBe.value
```

A program a megfelelő árfolyam megadásával bármely két pénznem átváltására, sőt, bármely két szám osztására alkalmas.

Megjegyezzük, hogy ha árfolyamként nullát írunk be, akkor az eredmény *Infinity* (végtelen) lesz, hiszen nullával nem lehet osztani. Ha véletlenül (vagy szándékosan) betűket is begépelünk a szövegmezőkbe, akkor pedig NaN (Not a Number, nem szám) lesz az eredmény. Ha telepítettük a Script Debuggert, akkor a „Nullával való osztás” illetve Típuseltérés hibaüzenetek mellett elindíthatjuk a hibakeresőt. A beolvasási hibák megelőzésével a későbbiekben részletesen foglalkozunk.

## Az onkeydown esemény

Bár az előző példák jól működnek, mégis adódhat furcsának tűnő eredmény. Ha egy átszámítás után beírjuk az új forint értéket, de nem nyomjuk le az Átszámítás gombot, akkor a szövegmezőben már az új adat, az egyenlőségjel után viszont még az előző eredmény látszik. Ha valaki ilyenkor pillant a képernyőre, azt hiheti, hogy rosszul számoltunk. Ennek a nem kívánt hatásnak a megelőzésére használjuk a szövegmező *onkeydown* eseményét.

Az *onkeydown* esemény akkor következik be, ha a felhasználó lenyom egy billentyűt, mert megváltoztatja a szövegmező tartalmát. A **3–24. példa** kódjában ekkor kitöröljük az *Euro*-objektumba írt számot. Ezt úgy tehetjük meg, hogy az *innerText* tulajdonságnak üres karakterláncot adunk értékül. Az üres sztring két idézőjelet jelent egymás után: "".

```
<SCRIPT event = "onkeydown" for = "ForintBe">
  Euro.innerText = ""
</SCRIPT>
```

A CD-n lévő példában ugyanilyen eseménykezelőt készítettünk az *ArfolyamBe* szövegmezőhöz is. Ha kipróbáljuk a szövegmező tartalmának módosítását, láthatjuk, hogy azonnal eltűnik az előző eredmény. Az átváltáshoz ismét rá kell kattintatni a parancsgombra.

A programok írásánál mindig törekedjünk a félre nem érthető megjelenítésre!

## Numerikus értékek összegezése

Az értékadó utasítás első pillantásra szokatlan formájában a változó új értékének a megadásakor az egyenlőségjel jobb oldalán felhasználjuk az előző értéket:

```
Magassag = 6
Magassag = Magassag + 2
```

A második utasítás végrehajtásakor az interpreter előveszi a *Magassag* nevű változóban tárolt értéket, a 6-ot, hozzáad 2-t, majd az új értéket beírja a *Magassag* változó által lefoglalt memóriaterületre. A két utasítás végrehajtása után tehát a magasság értéke 8 lesz. Néhány programnyelvben az értékadást `:=`-vel jelölik, hogy megkülönböztessék az egyenlőségtől.

A **3–25. példában** olyan programot készítünk, amely összeadja a weblapon lévő szövegmezőbe írt értékeket, és folyamatosan kiírja az összeget. Ehhez egy szövegmezőt, egy parancsgombot és a megjelenítéshez egy SPAN-objektumot helyezünk el a dokumentumban. Beleírjuk a SPAN-objektumba a nulla kezdőértéket is:

```
<INPUT id = "AdatBe" type = "text" value = 0>
<INPUT id = "Szamitas" type = "button" value = "Összegezés"><BR>
Összeg: <SPAN id = "OsszegKi">0</SPAN>
```

Az összeg nyilvántartásához deklarálunk egy *Osszeg* nevű változót, melynek 0 kezdőértéket adunk:

```
Dim Osszeg
Osszeg = 0
```

A parancsgomb eseménykezelőjében a felhasználó által beírt értéket hozzáadjuk az *Osszeg* előző értékéhez, majd megjelenítjük az eredményt:

```
<SCRIPT event = "onclick" for = "Szamitas">
  Osszeg = Osszeg + AdatBe.value
  OsszegKi.innerText = Osszeg
</SCRIPT>
```

Nagyon fontos, hogy az *Osszeg* nevű változót külön szkriptben deklaráljuk. Az eseménykezelő szkriptekben deklarált változók a szkript végrehajtása után törlődnek, értékük nem áll a rendelkezésünkre! A változóknak ezt az úgynevezett hatókörét a későbbiekben részletesen tárgyaljuk.

A **3–26. példa** kiegészíti a weblapot egy *Törlés* gombbal, hogy újra lehessen kezdeni az összegezést. A gomb *onclick* eseménykezelője törli a szövegmező és a SPAN tartalmát, illetve nullázza az *Osszeg* változót:

```
AdatBe.value = 0
Osszeg = 0
OsszegKi.innerText = 0
```

Ha a beírt értékeket továbbra is látni akarjuk, akkor egészítsük ki a dokumentumot egy újabb SPAN-objektummal, melynek *innerText* tulajdonságához a *Szamitas* parancsgomb eseménykezelőjében hozzáfűzünk egy pluszjelet és a beírt értéket:

```
Beírt számok: <SPAN id = "Sorozat">0</SPAN>
...
Sorozat.innerText = Sorozat.innerText & " + " & AdatBe.value
```

A törlés gomb eseménykezelőjét ki kell egészíteni a *Sorozat*-objektum *innerText* tulajdonságának a nullázásával:

```
Sorozat.innerText = 0
```

A módosított dokumentumot a **3–27. példa** mutatja be.

#### Hibák az adatbevitelnél

Már említettük, hogy ha a felhasználótól adatot kérünk be egy programban, akkor hibákra kell számítanunk. Ezekben az esetekben a böngésző hibaüzenetet ad, és megszakítja a szkript utasításainak a végrehajtását. Az Internet Explorer az állapotsor bal szélén egy felkiáltójeles ikonnal, és a „Hiba az oldalon” üzenettel figyelmeztet. Ha bekapcsoltuk a parancsfájlhibák üzeneteinek megjelenítését, akkor kijelzi a hiba keletkezésének helyét és típusát.

Ha véletlenül vagy szándékosan betűk, illetve más karakterek is kerülnek a szövegmezőbe, akkor programunk nem tudja számként értelmezni a karaktersorozatot. Ekkor a Típuseltérés hibaüzenetet kapjuk, általában megjelenítve a begépelt sztringet.

Ha telepítettük a Microsoft Script Debuggert, akkor egy ilyen hiba előfordulásánál a hibaüzenet mellett a böngésző rákérdez, hogy elindítjuk-e a Hibakeresőt. Igen válasz esetén betölti a Script Debuggert, megnyitja a forrásfájlt, és sárgával jelzi a hibás sort.



A továbbiakban fokozatosan megismerkedünk ezeknek a hibáknak a kezelési mód-szereivel. Egy programot a lehetőségekhez képest „bolondbiztosra” kell készíteni! Vizsgálja meg a felhasználó által begépelte adatokat. Figyelmeztessen az elkövetett hibákra, és adjon módot a javításra.

Példaprogramjaink a VBScript utasításainak bemutatását, a programozási fogások elsajátítását szolgálják. Az éppen tárgyalt ismeretanyagra koncentrálnak, ezért gyakran elhagyják a beviteli hibák kezelését.

### 3.4. Függvények

A bonyolultabb számítások elvégzéséhez a matematikában függvényeket használnak. Függvények adják meg egy szám négyzetgyökét, egészrészét, vagy egy szög szinuszt. A programozási nyelvek is függvényekkel határozzák meg ezeket az értékeket. A függvények fontos szerepet játszanak a programozásban. Már az eddigiekben is többször hivatkoztunk rájuk.

Függvény:  
egy konkrét értéket előállító, névvel ellátott, elkülönített utasítássorozat.

A VBScript számos előre elkészített függvénnyel rendelkezik, de mi magunk is készíthetünk függvényeket.

#### Matematikai függvények

A VBScript legfontosabb matematikai függvényei:

Abs(x)	az x abszolút értéke
Atn(x)	az x arkusz tangense (a tangensből visszakeresi a radiánban mért szöget)
Cos(x)	a radiánban mért szög koszinusza
Exp(x)	az $e^x$ értéke ( $e \approx 2,72\dots$ )
Fix(x)	az x tizedesjegyeinek elhagyásával kapott egész szám
Int(x)	az x egészrésze (integer: egész) (a legnagyobb egész szám, amely még nem nagyobb az x-nél)
Log(x)	$\ln x$ (az x természetes alapú logaritmus)
Sgn(x)	előjelfüggvény (sign: előjel) $\text{Sgn}(x) = 1$ , ha $x > 0$ ; $\text{Sgn}(x) = 0$ , ha $x = 0$ ; $\text{Sgn}(x) = -1$ , ha $x < 0$
Sin(x)	a radiánban mért szög szinusza
Sqr(x)	az x négyzetgyöke (square root: négyzetgyök)
Tan(x)	a radiánban mért szög tangense

Az x helyére az utasításokban egyetlen számot, változónevet, újabb függvény nevét vagy ezek kombinációjából álló tetszőleges aritmetikai kifejezést írhatunk.

Figyeljünk arra, hogy a szögfüggvényeknél a szöget radiánban kell mérni! Mint ismeretes,  $1 \text{ radián} = 180/\pi \text{ fok}$ , illetve  $1 \text{ fok} = \pi/180 \text{ radián}$ . 10-es alapú exponenciális és logaritmusfüggvény nem áll a rendelkezésünkre, de könnyen elő tudjuk állítani:

$$10^x = \text{Exp}(x * \text{Log}(10))$$

$$\log_{10} x = \text{Log}(x) / \text{Log}(10)$$

Megjegyezzük, hogy az e-alapú exponenciális és logaritmusfüggvénnyel ugyanúgy lehet számolni, mint a 10-es alapúval, az átváltásra tehát többnyire nincs szükség.

A matematikában általában használatos függvények mellett kiemeljük az egészrész képző *Int*, és a törtrészt elhagyó *Fix* függvényeket. A két függvény csak a negatív számok esetén tér el egymástól:

$$\begin{aligned} \text{Int}(3.2) &= 3, \text{Int}(3.8) = 3, \text{Int}(-3.2) = -4, \text{Int}(-3.8) = -4 \\ \text{Fix}(3.2) &= 3, \text{Fix}(3.8) = 3, \text{Fix}(-3.2) = -3, \text{Fix}(-3.8) = -3 \end{aligned}$$

Egy szám törtrészét a következőképpen határozhatjuk meg:

$$\text{Törtrész} = \text{Szám} - \text{Int}(\text{Szám}), \text{ például: } 6.2 - \text{Int}(6.2) = 0.2$$

A kifejezés a negatív számokra is jól működik, mert a törtrész matematikai definíciója szerint:

$$(-6.2) - \text{Int}(-6.2) = +0.8$$

Ha -0,2-et akarunk kapni, akkor az *Int* helyett a *Fix* függvényt használjuk:

$$(-6.2) - \text{Fix}(-6.2) = -0.2$$

## Függvények a szkriptekben

A függvény az általa meghatározott érték kiszámításához adatokat, úgynevezett paramétereket használ. A matematikai függvények felsorolásánál x-szel jelöltük a paramétert. A függvények több paraméterrel is rendelkezhetnek. Ekkor a paraméterek a függvény neve után állnak zárójelben, egymástól vesszővel elválasztva:

$$\text{FüggvényNév}(\text{Paraméter}_1, \text{Paraméter}_2, \dots)$$

Az interpreterrel feltétlenül közölni kell, hogy mit kezdjen a függvény által meghatározott értékkel. Ezért függvények csak kifejezésekben szerepelhetnek, különben hibüzenetet kapunk. Egy függvény nevét önmagában is kifejezésnek tekintjük, ezért bárhová leírhatjuk, ahová egyébként kifejezést írhatnánk (például értékadó utasítás jobb oldalára).

Ha a függvény nevét beillesztjük valamilyen kifejezésbe, akkor azt mondjuk, hogy *meghívjuk* a függvényt. A paraméterek segítségével *átadunk* értékeket a függvénynek. A függvényhívás hatására végrehajtódnak a függvényértéket előállító utasítások, a függvény *visszaadja* a függvényértéket.

A `Negyzetgyok = Sqr(49)` utasítás végrehajtásakor például meghívjuk a négyzetgyök függvényt. Átadjuk neki a 49-et. A függvény visszatérési értéke a 49 négyzetgyöke, azaz 7. Ezt kapja meg a *Negyzetgyok* nevű változó.

A **3–28. példában** olyan weblapot készítünk, amely meghatározza egy szám négyzetgyökét. A beolvasáshoz az *AdatBe* azonosítójú szövegmezőt, az eredmény kiírásához pedig a *GyokKi* azonosítójú SPAN-objektumot használjuk. A függvényhívást végző parancsgomb eseménykezelője:

```
<SCRIPT language = "VBS" event = "onclick" for = "Gomb">
  GyokKi.innerText = Sqr(AdatBe.value)
</SCRIPT>
```

Az eddigiekhez hasonlóan az eredményt megjelenítő SPAN-objektum tartalmát töröljük, ha a felhasználó új adatot ír a szövegmezőbe. Az *onkeydown* eseménynek azonban nem készítünk külön szkriptet, hanem az *AdatBe*-objektum nyitó tagjában helyezzük el az értékadást, ahogyan eleinte is tettük. Egy-egy rövid utasítás esetén ez a legegyszerűbb megoldás az eseménykezelésre.

Ha a szövegmezőbe negatív számot gépelünk be, akkor az „Érvénytelen eljárás hívás vagy argumentum” hibaüzenetet kapjuk. Egy „bolondbiztos” program esetén a gyökvonás előtt meg kell vizsgálnunk, hogy elvégezhető-e a művelet. Ennek módszerével hamarosan megismerkedünk.

## Az eredmények kerekítése

A számítások eredményeinek megjelenítése során zavaró lehet, hogy az interpreter esetenként sok tizedesjegyet ír ki. A kerekítést a *Round* függvénnyel végezhetjük el, amely két paraméterrel rendelkezik. Hívásakor meg kell adni a kerekítendő értéket és a kívánt tizedesjegyek számát:

```
Round(Kerekítendő_érték, Tizedesjegyek_száma)
```

A paraméterek helyén tetszőleges aritmetikai kifejezés állhat.

Ha egész számra akarjuk az eredményt kerekíteni, akkor 0-t adunk meg a tizedesjegyek számaként. Ilyenkor a második paraméter el is hagyható.

Néhány példa a *Round* alkalmazására:

```
Round(3.14159, 2) = 3,14
Round(-9.654, 1) = -9,7
Round(45.392, 0) = 45
Round(14.96) = 15
Round(3.6, 5) = 3,6
```

Mint az utolsó példából láthatjuk, ha a szám kevesebb tizedesjegyből áll, mint a paraméterként megadott érték, akkor a *Round* az eredeti számjegyeket adja vissza.

Ha a 3–28. példában a négyzetgyök értékét 4 tizedesjegyre kívánjuk megjeleníteni, akkor az értékadó utasítást a következőképpen kell módosítani:

```
Gyok.innerText = Round(Sqr(Adat.value), 4)
```

A *Round* függvény eléggé szokatlan módon a 0,5-et páros egészrésznél abszolút értékben lefelé, páratlannál pedig felfelé kerekíti:

```
Round(-6.5, 0) = -6, Round(-1.5, 0) = -2
Round(6.5, 0) = 6, Round(1.5, 0) = 2
```

## Formázott megjelenítés

Kerekítéssel elhagyhatjuk egy numerikus érték felesleges tizedesjegyeit. Az adatok megjelenítésénél azonban szükségünk lehet előírt számú tizedesjegyre akkor is, ha a szám kevesebbet tartalmaz. A nagy számok kiírásánál pedig célszerű hármassával cso-

portosítani a számjegyeket, hogy könnyebben áttekinthető legyen az eredmény. Mind-ezeket a kívánalmakat a *FormatNumber* függvény segítségével teljesíthetjük:

```
FormatNumber(Kifejezés, Tizedesjegyek_száma, Vezető_nulla, _  
Negatív_zárójel, Ezresek_elválasztása)
```

A *FormatNumber* paraméterei:

<i>Kifejezés:</i>	a formázásra kerülő aritmetikai kifejezés
<i>Tizedesjegyek_száma:</i>	a szükséges tizedesjegyek száma (szükség esetén nullákkal egészíti ki, -1 esetén az operációs rendszer területi beállításainál megadott értéket veszi)
<i>Vezető_nulla:</i>	kiírja-e a nullát az egynél kisebb számoknál
<i>Negatív_zárójel:</i>	zárójelbe tegye-e a negatív számokat
<i>Ezresek_elválasztása:</i>	csoportosítsa-e hármassával az egészrész számjegyeit

A három utolsó paraméter úgynevezett háromállapotú (tristate) konstans. Értékeit megadhatjuk számokkal vagy a szkriptben deklarált konstansokkal. Szokásos elnevezésük:

<i>Konstans:</i>	<i>Érték:</i>	<i>Jelentés:</i>
TristateTrue:	-1	igen (igaz)
TristateFalse:	0	nem (hamis)
TristateUseDefault:	-2	a területi beállításokat veszi figyelembe

A paraméterek közül csak a kifejezést kötelező megadni. Bármely másik elhagyható, kimaradó helyét vesszővel kell jelölni. A **3–29. példa** különböző paraméterek esetén mutatja be a *FormatNumber* viselkedését.

### Függvények alkalmazása

A **3–30. példában** meghatározzuk két pont távolságát a koordináta-rendszerben. Ha a pontok koordinátái  $x_1, y_1$  és  $x_2, y_2$ , akkor a Pitagorasz tétel alapján a távolság:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Programunk négy szövegmező segítségével beolvassa a pontok koordinátáit, majd megjeleníti a távolságot. A számítás forráskódja egyszerűsödik, ha a szövegmezők *value* tulajdonságát változóknak tároljuk:

```
X1 = X1_be.value : Y1 = Y1_be.value  
X2 = X2_be.value : Y2 = Y2_be.value
```

Majd behelyettesítjük ezeket az értékeket a képletbe, és megjelenítjük az eredményt:

```
Tavolsag.innerText = Sqr((X2 - X1)^2 + (Y2 - Y1)^2)
```

A számításokat végző utasítások természetesen egy parancsgomb eseménykezelőjébe kerülnek.

## Sztringek átalakítása numerikus értéké

Az *InputBox* és a szövegmezők mindig sztringeket olvasnak be. Ha a begépelte karakterlánc számként értelmezhető és aritmetikai műveletben vesz részt, akkor az interpreter automatikusan numerikus értéké alakítja át. Ezért nem okozott hibát a 3–23. példában a `ForintBe.value / ArfolyamBe.value` kifejezés. Sőt, a hányadost átadhattuk egy objektum szöveg tulajdonságának, mivel az interpreter visszaalakította a számot sztringgé:

```
Euro.innerText = ForintBe.value / ArfolyamBe.value
```

A VBScript (és a Visual Basic) régebbi változataiban a sztringek összefűzésére a `+` jelet használták, amit kompatibilitási okokból meghagytak a jelenlegi változatban is. Ezért ha a két beolvasott értéket össze akarjuk adni, akkor nem használhatjuk az

```
EgyikSzam.value + MasikSzam.value
```

kifejezést, mert az interpreter – egyébként logikusan – összefűzi a két sztringet. Ilyenkor először át kell alakítanunk a karakterláncot numerikus értéké. Ezt az átalakítást az úgynevezett típuskonverziós függvények végzik el.

Típuskonverzió: az egyik típusú érték átalakítása más típusúvá.

Az átalakítást többféle típusra is elvégezhetjük. A választott típustól függ a program hatékonysága és a helyfoglalás a memóriában. Amíg nem tárgyaljuk részletesen a változók típusait, addig a *CSng* (convert to single: átalakítás egyszeres pontosságúra) függvényt fogjuk használni. Paramétereként az átalakítandó kifejezést kell megadni, ami egyszerűbb esetben egy sztring, például egy szövegmező *value* tulajdonsága lehet. Ha a kifejezés nem alakítható át numerikus értéké (például betűket is tartalmaz), akkor hibaüzenetet kapunk.

A fent említett két érték összeadását tehát a következő utasítással érhetjük el:

```
CSng(EgyikSzam.value) + CSng(MasikSzam.value)
```

A többi matematikai műveletben, vagy ha az összeadás egyik operandusa numerikus érték, akkor a konverzió automatikusan megtörténik. Az

```
EgyikSzam.value + 5
```

kifejezés már számokat ad össze.

Emlékezzünk vissza arra, hogy az azonos precedenciájú műveleteket az interpreter balról jobbra haladva végzi el. Ezért a következő esetben:

```
EgyikSzam.value + MasikSzam.value + 5
```

először összefűzi a két szövegmezőbe beírt karaktersorozatot, majd – mivel egy szám következik, – konvertálja numerikus értéké, és így végzi el az összeadást. Ha az egyik szövegmezőbe 12-t, a másikba 34-et írtunk, akkor az összefűzés eredménye 1234, az összeadásé pedig:  $1234 + 5 = 1239$  lesz. Az egyes eseteket a **3–31. példa** szemlélteti.

Ha programjainkban – főleg összeadás esetén – furcsa eredményre jutunk, akkor a számítások előtt a *CSng* függvény segítségével végezzük el a beolvasott értékek konverzióját.

#### Véletlenszámok előállítása

Az eddig megismert függvények mellett rendelkezésünkre áll néhány speciális lehetőség is. Az *Rnd* (random, véletlen) függvény például véletlenszerűen előállít egy számot, amely kisebb, mint 1 és nagyobb vagy egyenlő, mint 0. Paramétere speciális célokat szolgál, mi nélküle fogjuk használni:

```
VeletlenSzam = Rnd()
```

Bár az üres zárójelet el is lehet hagyni, kiírásával jelezzük, hogy függvényt hívtunk.

Egy tetszőleges, a *Minimum* és a *Maximum* egész számokkal határolt zárt intervallumban a következő kifejezéssel állíthatunk elő véletlenszámot:

```
VeletlenSzam = Int((Maximum - Minimum + 1) * Rnd() + Minimum)
```

Az *Int* függvénnyel a szám törtrészét hagytuk el.

A következő kifejezés 1 és 90 között választ egy számot az ötöslottó kitöltéséhez:

```
LottoSzam = Int(90 * Rnd() + 1)
```

Ha ezt egy parancsgomb eseménykezelőjébe írjuk, és megjelenítjük az eredményt, akkor egy nagyon egyszerű programot kapunk, amely véletlenszerűen fog lottószámokat választani. A teljes kódot a **3–32. példa** mutatja. (Itt még nincs módunk az azonos számok választásának elkerülésére, ebben az esetben meg kell ismételni az eljárást.)

Ha frissítjük a dokumentumot, és kattintgatunk az OK gombra, akkor észrevehetjük, hogy mindig ugyanazt a véletlenszám-sorozatot kapjuk. Ennek elkerüléséhez használjuk a *Randomize* utasítást. Nem kell beletenni az eseménykezelő szkriptbe, elegendő egyszer, a weblap betöltésénél végrehajtani. Ezt úgy érhetjük el, hogy külön szkriptbe írjuk. Hatására mindig új véletlenszám-sorozatot kapunk. A módosított kódot a **3–33. példában** láthatjuk.

#### A szín-függvény

A VBScript függvényei nem csak a matematikával kapcsolatos értékeket tudnak meghatározni. Az RGB függvény például egy szín kódját állítja elő.

A képernyőn megjelenő színek három alapszín, a vörös (R, red), a zöld (G, green) és a kék (B, blue) keverékéből állnak össze. Az RGB rendszerben mindhárom színösszetevő értékét egy 0 és 255 közé eső számmal adhatjuk meg. Így összesen  $256 \cdot 256 \cdot 256 = 16\,777\,216$  szín keverhető ki. (A megjelenő színek száma már függ a képernyő és a grafikus kártya minőségétől.) Ha mindhárom érték 0, akkor feketét kapunk, ha mindegyikük 255, akkor pedig fehéret.

A színeket a böngésző számára hexadecimálisan kell megadni. Az RGB függvény a három, decimálisan beírt értékből meghatározza a szükséges hexadecimális kódot, így sokkal könnyebben tudunk színeket kikeverni.<sup>13</sup>

```
Szin = RGB(kék_összetevő, zöld_összetevő, vörös_összetevő)
```

Mint említettük, mindhárom paraméternek 0 és 255 közé kell esnie. Az RGB függvény a 255-nél nagyobb értékeket 255-nek veszi. A **3–34. példa** bemutatja a függvény használatát, és lehetővé teszi, hogy kipróbáljuk az alapszínek keverését.

Változatos megjelenést érhetünk el, ha a szöveg színét a háttérszín komplementerére állítjuk. Egy szín komplementerét úgy kapjuk meg, hogy az alapszínek értékét kivonjuk 255-ből:

```
KomplementerSzin = RGB(255 - kék_összetevő, _  
                        255 - zöld_összetevő, 255 - vörös_összetevő)
```

A hatást a **3–35. példa** mutatja be.

A későbbiekben szükségünk lesz a színek hexadecimális kódjára. Egy hexadecimális értéket idézőjelek között és #-tel kezdve kell megadni, például:

```
SzinKod = "#ffa089"
```

Jónéhány szín kódját megtaláljuk a CD-melléklet Dokumentumok mappájában, de magunk is meghatározhatjuk, ha a 3–34. példát kiegészítjük a kód kiírásával. Ehhez egyszerűen a *document.bgColor* értékét kell megjeleníteni. A hexadecimális színek kódokat a **3–36. példa** segítségével állapíthatjuk meg. A színkódot szövegmezőbe írtuk, hogy sötét háttér esetén is olvasható legyen.

## Saját függvények definiálása

A VBScript lehetővé teszi saját függvények definiálását. Ha ugyanazt a képletet többféle számmal is ki kell értékelni, vagy egy összetett számítási folyamatot akarunk egyszerűsíteni, akkor célszerű erre külön függvényt készíteni.

A függvénydefiníció szintaxisa:

```
Function FüggvényNév(Paraméter_1, Paraméter_2, ...)  
    utasítások  
End Function
```

Figyeljük meg, hogy a strukturált elrendezés elveinek megfelelően a függvényhez tartozó utasításokat beljebb kezdjük!

A függvény nevénél a változónévre vonatkozó előírásokat kell betartanunk. A függvényhívásnál a paraméterek helyére – a matematikai függvényekhez hasonlóan – írhatunk konkrét értékeket, változóneveket vagy egyéb kifejezéseket. A paraméterek nevének tetszőleges változónevet választhatunk, ezekkel hivatkozhatunk az értékükre.

A böngésző a weblap megjelenítésekor nem hajtja végre a *Function* és az *End Function* közé eső utasításokat, csak szintaktikus ellenőrzést végez.

---

<sup>13</sup> Az Internet Explorer számára a vörös összetevő a legnagyobb helyiértékű bájt, ezért kell az RGB rövidítéshez viszonyítva fordított sorrendben megadni a színeket.

A függvény utasításai között kell szerepelnie egy olyan értékadó utasításnak, melynek bal oldalán a függvény neve áll zárójelek és paraméterek nélkül. Ezzel adjuk meg a függvény visszatérési értékét:

```
FüggvényNév = Kifejezés
```

A függvényértéket tetszőleges műveletsorozatban felhasználhatjuk. A következő függvény átszámítja a fokban megadott szöget radiánra:

```
Function Fok2Rad(X)
    Fok2Rad = X * 0.01745329252
End Function
```

A függvényt aztán felhasználjuk az értékadó utasításokban:

```
Dim FokSzog, RadianSzog, SzinusSzog
FokSzog = 32
RadianSzog = Fok2Rad(FokSzog)
SzinusSzog = Sin(RadianSzog) ' vagy Sin(Fok2Rad(FokSzog))
```

Függvényeink több paraméterrel is rendelkezhetnek. A következő függvény egy derékszögű háromszög befogóiból a Pitagorasz-tétel alapján kiszámítja az átfogót:

```
Function Atfogo(A, B)
    Atfogo = Sqr(A^2 + B^2)
End Function
```

A függvény alkalmazását a **3–37. példa** mutatja be. Az eseménykezelést külön szkripttel kell elvégeznünk, de a függvényt definiáló szkript egyéb utasításokat is tartalmazhatna. A függvény paramétereit a két szövegmező *value* tulajdonsága alkotja.

Saját függvényeinknél sem kötelező paramétereket használni. A következő függvény például megadja a  $\pi$  nagyon pontos értékét:<sup>14</sup>

```
Function Pi()
    Pi = 4 * Atn(1)
End Function
```

A függvény hívásakor – a beépített függvényekhez hasonlóan – elhagyhatjuk a zárójelet, de ezt nem javasoljuk:

```
Kerulet = 2 * R * Pi ' Így félreérthető a hivatkozás!
Terulet = R^2 * Pi()
' Ne hagyjuk el a zárójelet a függvény neve után!
```

A **3–38. példa** megjeleníti a  $\pi$  pontos értékét, majd kiszámítja a felhasználó által megadott sugarú kör kerületét és területét. Figyeljük meg, hogy a  $\pi$  értéke még az *InputBox* előtt megjelent a weblapon! Hasonlítsuk össze a kódot a 3–17. példáéval.

Eddigi függvényeink definíciója egyetlen utasítást tartalmazott. A továbbiakban jóval bonyolultabb függvényeket is készítünk.

---

<sup>14</sup> Az  $\arctan 1 = \pi/4$  alapján.



## Formális és aktuális paraméterek

A függvény paraméterei formálisan változónevek lehetnek, melyeket a függvény utasításaiban használhatunk fel. A függvény hívásakor a paraméterek helyére tetszőleges, a paraméter típusának megfelelő kifejezés írható:

```
Függvénynév(Kifejezés_1, Kifejezés_2, ...)
```

Az interpreter a függvény hívásánál kiszámítja a kifejezések értékét, és a paraméterek nevének megfelelően behelyettesíti a függvényértéket előállító utasításokban a paraméterek helyére.

A függvény definíciójában szereplő paramétereket formális, a hívásnál megadott kifejezéseket pedig aktuális paramétereknek nevezzük. Az aktuális paramétereket gyakran a függvény argumentumainak hívják.

Formális paraméter: a függvény definíciójában szereplő paraméter (változónév).  
Aktuális paraméter (argumentum): a függvény hívásában szereplő kifejezés.

Az A és B befogójú derékszögű háromszög átfogóját meghatározó

```
Function Atfogo(A, B)
```

függvényünk esetén az *A* és *B* változók a függvény formális paraméterei, melyek az átfogót kiszámító utasításban szerepelnek. Ha programunkban az egyik befogó 3, a másik pedig  $4 * (2 + \text{Magassag})$ , akkor a háromszög átfogóját meghatározó függvényhívás:

```
Atfogo(3, 4 * (2 + Magassag))
```

A 3 és a  $4 * (2 + \text{Magassag})$  kifejezések a függvény aktuális paraméterei (argumentumai). Ezeket helyettesíti be az interpreter a függvény utasításainak végrehajtásakor az *A*, *B* változók helyére.<sup>15</sup>

## Kód csatolása a dokumentumhoz

A bonyolult szkriptek hosszadalmassá és nehezen áttekinthetővé teszik a HTML-kódot. Ha több dokumentumban is felhasználjuk ugyanazt az utasítássorozatot, akkor minden egyes fájlba be kell gépelni vagy másolni. Ha változtatunk a kódon, akkor meg kell keresnünk a szkript összes előfordulását, ami sok hiba elkövetéséhez vezet.

Ezt a munkát a szkript csatolásával takaríthatjuk meg. Az utasítássorozatot egy külön fájlban tároljuk, és a SCRIPT-objektum *src* (source, forrás) tulajdonságának megadásával csatoljuk a dokumentumhoz. A fájl tetszőleges szövegszerkesztővel elkészíthetjük, de formázás nélküli szöveggként kell elmenteni. A fájl kiterjesztése is tetszőleges lehet, mi a továbbiakban az elterjedt gyakorlatnak megfelelően a .vbs kiterjesztést használjuk. Ne felejtjük el megadni a HTML-kódban a SCRIPT-objektum záró tagját! Ilyen módon előre elkészített vagy máshonnan átvett függvényeket is csatolhatunk a dokumentumhoz.

<sup>15</sup> A cím és érték szerinti paraméterátadást később tárgyaljuk.

A csatolt fájlba csak a VBScript utasításokat kell beírni a SCRIPT nyitó és záró tagja nélkül. Másoljuk át a 3–37. példa *Atfogo* függvényét a Jegyzetömbbe, és mentjük el Háromszög.vbs néven (a fájl a CD-melléklet Fejezet03 mappájában is megtalálható). Egy másik dokumentumhoz egy üres szkript segítségével csatolhatjuk, melynek *src* (source, forrás) tulajdonságánál megadjuk az elérési utat. Az elérési út megadását a 2.2. fejezetben, a hivatkozások beillesztésénél ismertettük. Ha a vbs fájl ugyanabban a mappában helyezkedik el, mint a HTML-dokumentum, akkor a következő szkript végzi el a csatolást:

```
<SCRIPT language = "VBS" src = "Háromszög.vbs"> </SCRIPT>
```

Ezek után ugyanúgy használhatjuk az *Atfogo* függvényt, mintha valóban beírtuk volna a kódját a dokumentumba.

A kód csatolását a **3–39. példa** mutatja be. Nézzük meg a böngészőből a forráskódot (Nézet/Forrás). Maga a függvény nem szerepel benne, csak a hivatkozás.

A szkriptet tartalmazó fájl akár az Interneten is lehet. Ebben az esetben élő Internet-kapcsolattal kell rendelkezünk az eléréséhez.

### Helyiértékes megjelenítés

Táblázatok megjelenítésénél a későbbiekben szükségünk lesz arra, hogy a számok azonos helyiértékei egymás alá kerüljenek. Sajnos nincs olyan VBScript függvény, amely ezt a formázást elvégezné, a kisebb értékek esetén például kellő számú szóközt hagyna ki a szám előtt. Ezért készítettünk egy olyan függvényt, amely kialakítja a megfelelő formátumot. Bár a működését eddigi ismereteink alapján még nem értjük, magát a függvényt így is felhasználhatjuk.

A függvény kódját a CD-melléklet Fejezet03 mappájában, a *Formaz.vbs* fájlban találjuk, amit csatolnunk kell a dokumentumhoz. A függvényhívás formája:

```
Formaz(Kifejezés, Egészrész_karakterek_száma, Tizedeshelyek_száma)
```

A függvény az első paraméterként szereplő kifejezés egészrészét nem törhető szóközzel a megadott számú karakterre egészíti ki. Ha a kifejezés egészrésze több jegyből áll, mint a második paraméter, akkor formázás nélkül az eredeti értéket adja vissza. Negatív számok esetén az előjel az egészrésztől vesz el egy karaktert. A kifejezés értékének tizedesrészét az utolsó paraméterként megadott számú jegyre kerekíti. Ha a tizedeshelyek számaként 0-t adunk meg, akkor egészre kerekíti a számot, és nem ír ki tizedesvesszőt.

A *Formaz* függvény működését, használatát és csatolását egy dokumentumhoz a **3–40. példa** mutatja be. A helyiértékes megjelenítéshez olyan betűtípust kell választanunk, amelyben egyforma szélességűek a karakterek. Ezt a célt szolgálja a PRE-objektum. (Ismertetését lásd a CD-melléklet 2–19. példájában.)

Megjegyezzük, hogy ezt a függvényt könyvünkhöz készítettük el, így nem tud szélsőséges eseteket kezelni. Ha a szám abszolút értéke nagyobb, mint  $10^{10}$ , vagy kisebb, mint  $10^{-10}$ , de nem nulla, akkor nem végzi el a formázást, az eredeti értéket adja vissza. Mind az egész, mind a tizedes karakterek száma legfeljebb 15 lehet.

## 3.5. Eljárások

### Eljárások a szkriptekben

A függvények elkülönítenek egy konkrét értéket meghatározó összetett utasítássorozatot, így áttekinthetővé teszik a programot. Ezt az egyszerűsítési lehetőséget akkor is használhatjuk, ha az utasítások nem valamilyen értéket számítanak ki, hanem más feladatot hajtanak végre. Ekkor eljárást (szubrutint) készítünk az utasítássorozatból.

Eljárás (szubrutin): névvel ellátott, elkülönített utasítássorozat.

Az eljárások a függvényekkel együtt a programok fontos építőelemei. Segítségükkel az algoritmust több kisebb, különálló részre lehet felbontani, ami nagymértékben elősegíti az áttekintést, a hibakeresést.

Az eljárásokat a változókhoz hasonló szabályok szerint kell elnevezni. Az eljárásnak is lehetnek formális paraméterei, melyek felhasználhatók az utasításokban. A paraméterek segítségével értékeket adunk át az eljárásnak. A formális és aktuális paraméterekre ugyanazok a tudnivalók vonatkoznak, mint amiket a függvényeknél ismertettünk.

Az eljárások szintaxisa:

```
Sub EljárásNév(Paraméter_1, Paraméter_2, ...)
    utasítások
End Sub
```

A böngésző a weblap betöltésekor nem hajtja végre az eljárások utasításait, csak szintaktikus ellenőrzést végez. Ezért az eljárásokban hivatkozhatunk olyan objektumokra is, melyeket csak később fogunk beleírni a HTML-kódba.

Az eljárások hasznos szerepet töltenek be, ha ugyanazt az utasítássorozatot a program több, különböző részén kell végrehajtani. Elegendő szubrutint készíteni az utasításokból és meghívni az eljárást. Az eljáráshívást a *Call* utasítással végezzük:

```
Call EljárásNév(Kifejezés_1, Kifejezés_2, ...)
```

A programban az eljárás hívása egyenértékű az utasítássorozat beillesztésével.

A *Call* kulcsszó használata a Visual Basic régebbi változataiból maradt meg. Ha az eljárásnak nincs paramétere, vagy csak egy értéket adunk át, akkor a *Call* elhagyható:

```
Eljárásnév() vagy Eljárásnév(Kifejezés)
```

Ezt az egyszerűsítési lehetőséget a továbbiakban ki fogjuk használni.<sup>16</sup>

Figyeljünk arra, hogy az eljáráshívások a programunk önálló utasításai, míg a függvényhívások kifejezésekben szerepelnek! A függvények által visszaadott értéket a függvény definícióján belül meg kell határoznunk, a függvény nevének szerepelnie kell egy értékadó utasítás bal oldalán. Az eljárásoknak nincsen visszatérési értékük, így nem vonatkozik rájuk ez a szabály.

<sup>16</sup> A *Call* elhagyása, de zárójelek használata esetén a paraméterátadás az alapértelmezéssel ellentétben érték szerint történik!

Megjegyezzük, hogy mind az eljárások, mind a függvények hívhatnak újabb eljárásokat, illetve függvényeket is.

A továbbiakban az eljárásokat és a függvényeket együttesen alprogramoknak fogjuk nevezni.

Alprogram: az eljárások (szubrutinok) és függvények összefoglaló neve.

#### Eljárás készítése

Az eljárások készítését a **3–41. példában** mutatjuk be, amelynek weblapján három parancsgomb segítségével három különböző háttérszint tudunk beállítani. Természetesen minden parancsgombhoz megírhatnánk a megfelelő eseménykezelő szkriptet. Ehelyett azonban egyetlen eljárást használunk, amely paraméterként megkapja a kívánt háttérszint:

```
Sub Szinez(Hatterszin)
    document.bgColor = Hatterszin
End Sub
```

Az egyes parancsgombok *onclick* eseménykezelőjében meghívjuk a *Szinez* eljárást a megfelelő színnel:

```
<INPUT type = "button" value = "Piros"
        onclick = 'Szinez("red") '>
<INPUT type = "button" value = "Kék"
        onclick = 'Szinez("blue") '>
<INPUT type = "button" value = "Sárga"
        onclick = 'Szinez("yellow") '>
```

Ha egy parancsgombra kattintunk, akkor az interpreter az eljárás definíciójában szereplő *Hatterszin* paramétert behelyettesíti a hívásnál beírt karakterlánccal, azaz a szín angol nevével. Ezért az eljárás végrehajtása során a *bgColor* tulajdonság a megfelelő értéket kapja.

A szubrutin hívása a függvényekkel ellentétben nem kifejezésekben szerepel, hiszen nem ad vissza semmilyen értéket. Önálló utasításként írjuk a szkriptbe. A **3–42. példa** a weblap betöltésekor egy figyelmeztető üzenet után beállítja a háttérszint:

```
<SCRIPT language = "VBS">
    Sub Szinez
        document.bgColor = "yellow"
    End Sub

    window.alert("Sárga lesz a háttér!")
    Szinez
</SCRIPT>
```

Mivel csak egyféle színt állítottunk be, nem használtunk paramétereket. Ekkor elhagyhatjuk az eljárás neve mellől a zárójeleket. Így mintegy új VBScript utasításokat is létrehozhatunk.

Figyeljük meg, hogy a szkriptben előre írtuk a szubrutin definícióját, majd egy üres sorral elválasztva következtek a betöltésnél végrehajtandó utasítások! Az interpreter

azonban a betöltésnél először a *window.alert* metódust hajtotta végre (még fehér volt a háttérszín), csak utána következett a *Szinez* eljárás meghívása. Az üres sor csak az áttekinthetőséget szolgálja, alkalmazása nem kötelező.

## Eseménykezelő eljárások

Az eddigiekben már háromféle módon végeztünk eseménykezelést. Bár mindegyikükre szükségünk lesz a továbbiakban, eljárások alkalmazásával egy újabb lehetőség áll a rendelkezésünkre, amit gyakran alkalmazunk. Előbb azonban foglaljuk össze az eddig megismert módszereket!

A programozási alapismeretek elsajátítása előtt az utasításokat beírtuk az objektum nyitó tagjába az eseménynév tulajdonság értékeként:

```
<Osztálynév ... eseménynév = "utasítások">
```

Mivel az eseménykezelés a nyitó tag sorában szerepel, ezt inline (a sorba beillesztett) módszernek hívják. Ez nagyon kényelmetlen technika, és csak egy (vagy legfeljebb kettő) utasítás, általában valamilyen értékadás esetén használjuk. Hosszabb utasítássorozat áttekinthetatlenné teszi a kódot.

Következő módszerünk külön szkriptet rendelt minden egyes objektum minden egyes eseményéhez:

```
<SCRIPT event = "eseménynév" for = "ObjektumNév">
    utasítások
</SCRIPT>
```

Ez a megoldás nagyon széttördeli a programunkat. Annyi szkriptet kell elhelyeznünk a dokumentumban, ahányféle objektum és esemény kezelését el akarjuk végezni. Elég ritkán használjuk, főleg akkor, ha az objektum közvetlenül nincs benne a HTML-kódban. Ilyen például a *window* vagy az úgynevezett külső objektumok. Ezek tárolják a böngésző adatait, a képernyő beállításait vagy az eddig felkeresett weblapok listáját.

A harmadik lehetőséget a 3–41-es példában alkalmaztuk. Ebben egy eljáráshívás szerepel az objektum nyitó tagjában. Ez hasonlít az inline módszerhez, de az esemény tulajdonság értékeként az utasítások felsorolása helyett csak a szubrutin nevét kell megadni:

```
<Osztálynév ... eseménynév = "eljáráshívás">
```

Az eljárás definíciója a HEAD-ben lévő szkriptbe került. Ez a módszer nagyon rugalmas lehetőségeket biztosít, hiszen több objektum különböző eseményeit is egyetlen eljárással kezelhetjük. A paraméterek segítségével lehet az eljárást hozzáigazítani a megfelelő objektumhoz és eseményhez.

Az ilyen közös eljárás alkalmazásának az a hátránya, hogy ha egy objektumnál sokféle esemény következhet be, akkor a nyitó tag a felsorolás miatt nagyon hosszú lesz a HTML-kódban, ami megnehezíti az áttekintést.

Mint említettük, az események kezelésére leggyakrabban egy negyedik módszert fogunk használni. Itt az objektum nyitó tagja nem utal az eseményre. A szkriptünkben lévő szubrutin nevébe kerül az objektum azonosítója, amit egy aláhúzásjel után az esemény megjelölése követ:

```
Sub ObjektumAzonosító_eseménynév
    utasítások
End Sub
```

Ennél a módszernél is külön eljárást kell írunk minden egyes objektum minden egyes eseményéhez, de az eseménykezelő szubrutinok a HEAD-ben sorakoznak egymás után, jól áttekinthetően.

A HTML-kód feldolgozásakor a böngésző feljegyzi, hogy mely objektumok rendelkeznek eseménykezelő eljárásokkal, és azok milyen eseményekre vonatkoznak. Ha bekövetkezik egy esemény, akkor végignézi ezt a listát, és meghívja a megfelelő eljárást, azaz végrehajtja az utasításait.

#### Példa az eseménykezelő eljárásokra

Az események kezelésének négyféle lehetőségét a **3–43. példában** mutatjuk be. Megjelenítünk négy parancsgombot. Kattintásra mindegyikük meghívja a *window*-objektum *alert* metódusát, amely kiírja az eseménykezelés módját.

Az inline eseménykezelés a parancsgomb elhelyezéséből és *onclick* tulajdonságának megadásából áll. Mivel nem hivatkozunk rá, nem kell azonosítóval ellátni a parancsgombot:

```
<INPUT type = "button" value = "Inline"
        onclick = 'window.alert("Eseménykezelés inline módon.")'>
```

A külön szkripttel történő eseménykezelés bemutatásához elhelyezünk egy azonosítóval ellátott parancsgombot a BODY-ban:

```
<INPUT id = "SajatSzkript" type = "button"
        value = "Sajat szkript">
```

illetve a neki megfelelő szkriptet a HEAD-ben:

```
<SCRIPT event = "onclick" for = "SajatSzkript">
    window.alert("Eseménykezelés saját szkripttel.")
</SCRIPT>
```

Közös eljárás használata esetén a parancsgomb nyitó tagjában hívjuk meg az eljárást, azonosítóra nincs szükség:

```
<INPUT type = "button" value = "Közös eljárás"
        onclick = "KozosEljaras">
```

Az eljárás a HEAD-be kerül, a szkript más eljárásokat és egyéb utasításokat is tartalmazhat:

```
Sub KozosEljaras
    window.alert("Eseménykezelés közös eljárással.")
End Sub
```

Ezt az eljárást bármely más objektum is meghívhatná saját eseményeinek kezelésénél.

A negyedik, eddig még nem használt módszernél az INPUT-objektum nyitó tagjában az eseményre vonatkozóan semmilyen megjelölés nem szerepel:

```
<INPUT id = "SajatEljaras" type = "button"
        value = "Sajat eljárás">
```

Az eseménykezelő eljárás a HEAD-ben lévő közös szkriptbe kerülhet. Neve kötelezően a parancsgomb azonosítója, azaz *SajatEljaras* lesz, melyet egy aláhúzásjellel elválasztva az esemény megjelölése követ:

```
Sub SajatEljaras_onclick
    window.alert("Eseménykezelés saját eljárással.")
End Sub
```

Mint említettük, ennek hatására a böngésző a dokumentum betöltésekor feljegyzi, hogy ha a felhasználó rákattint a *SajatEljaras* azonosítójú parancsgombra (azaz bekövetkezik az *onclick* esemény), akkor el kell indítania az így megjelölt eljárást.

Az eseménykezelés módszerei az előzőekben felsorolt előnyök és hátrányok mellett lényegében egyenértékűek.

Még egyszer kiemeljük, hogy az eseménykezelő eljárásokat nem a mi programunk aktivizálja, hanem a böngésző indítja el a megfelelő esemény bekövetkezésekor. Ettől függetlenül a többihez hasonló eljárásként viselkednek, tehát szükség esetén az esemény létrejötte nélkül a szkriptjeink is hívhatják őket.

### Metódusok: függvények és eljárások

Az objektumok metódusai is utasítássorozatot hajtanak végre, így az alprogramokkal egyenértékű szerepet töltenek be. Ha egy metódus visszaad valamilyen értéket, akkor függvényként, egyébként eljárásként használjuk.

Az eddig megismert metódusoknak (*alert*, *close*, *resizeTo*, *write*) nem volt visszatérési értéke. A szkriptekben külön sorban álltak, az eljárásokhoz hasonlóan önálló utasításként szerepeltek.

Néhány metódus a függvényekhez hasonlóan meghatároz valamilyen értéket, amit felhasználhatunk az értékadó utasításokban. Példaként megemlíjtük a *window*-objektum *confirm* (megerősítés) metódusát, amely egy OK és *Mégse* gombokkal ellátott üzenetablakot jelenít meg:

```
window.confirm("Üzenet_szövege")
```

A programnak tudnia kell, hogy a felhasználó melyik parancsgombra kattintott. A metódus visszatérési értéke igaz (*True*), ha az OK és hamis (*False*) ha a *Mégse* gombra kattintottunk. A *confirm* használatát a **3–44. példa** mutatja be.

A későbbiekben a *confirm* helyett a VBScript sokkal hatékonyabb *MsgBox* függvényét alkalmazzuk.

### A változók hatóköre

Az alprogramoknak lehetnek saját változóik, melyeket a *Sub* és az *End Sub*, illetve a *Function* és *End Function* utasítások között deklarálunk. Ezek csak az alprogramon belül használhatók, a kilépés után eltűnnek, törlődnek a memóriából. Az ilyen változókat lokális változóknak nevezzük. A formális paraméterek felsorolása is lokális deklarációnak számít.

**Lokális változók:** egy függvényen vagy eljáráson belül deklarált változók és a formális paramétereik. Csak abban az alprogramban használhatók, ahol deklaráltuk őket.

A szkriptek más helyein (nem alprogramban) deklarált változókra a dokumentumban bárhol hivatkozhatunk. Az értékeket felhasználhatjuk a HTML-kód más szkriptjeiben, és az alprogramokon belül is. Az ilyen változókat globális változóknak hívjuk.

**Globális változók:** a függvényeken vagy eljárásokon kívül deklarált változók. A dokumentum tetszőleges helyén felhasználhatóak.

Egy lokális változó eredeti értékét akkor sem érhetjük el, ha újból meghívjuk az őt létrehozó alprogramot. Teljes egészében újonnan deklarált változóként fog viselkedni. A változók felhasználásánál tehát figyelemmel kell lennünk arra a tartományra, ahonnan elérhetjük az értéküket.

**Hatókör:** az a tartomány (szkriptek, függvények, eljárások), ahol egy változót felhasználhatunk, értékére hivatkozhatunk. A hatókört elhagyva a változó eltűnik a memóriából.

Ha egy lokális változót olyan néven deklarálunk, amilyen néven már létezik globális változó, akkor ez utóbbi az alprogramon belül nem lesz elérhető. Értéke azonban megmarad, amint visszatérünk az alprogramból, újra felhasználhatóvá válik. A változók hatóköre néhány további kérdést is felvet, amire a későbbiekben visszatérünk.

Emlékezzünk vissza arra, hogy egy szkripten belül egyszerre legfeljebb 127 változót használhatunk. Ez a korlátozás egy hatókörre érvényes. Ha kiléptünk a hatókörből, akkor a változók eltűnnek, újabbakat lehet helyettük deklarálni.

Megjegyezzük, hogy a dokumentum objektummodell összes objektuma, tulajdonsága (és metódusa) globális, tehát a HTML-kódban bárhol felhasználható.

#### **Példa a változók hatókörére**

A fent elmondottakat a **3–45. példával** világítjuk meg:

```
<SCRIPT>
  Dim Elso, Masodik, Harmadik
  Elso = 1
  Masodik = 2

  Sub EgyikEljaras
    Dim Alfa, Beta
    Alfa = 10
    Beta = 20
  End Sub
```



```
Sub MasikEljaras
    Dim Masodik
    Masodik = 30
End Sub

EgyikEljaras
MasikEljaras
Harmadik = Masodik + 1
</SCRIPT>
```

Az *Elso* és a *Masodik* változó az egész szkriptre nézve globális, értékeik például az *EgyikEljaras* szubrutinban is felhasználhatók. Az *Alfa* és a *Beta* az *EgyikEljaras* lokális változói, csak az eljáráson belül használhatjuk őket. A *MasikEljaras* szubrutinban deklaráltunk egy olyan változót, amely globálisként már létezett. A szubrutinon belül értéke 30 lesz, és a másik nem érhető el. Amint kiléptünk a *MasikEljaras*-ból, ismét a globális *Masodik* változóra vonatkoznak a hivatkozások, így a *Harmadik* változó értéke az utolsó utasítás után  $2 + 1 = 3$  lesz, nem pedig  $30 + 1 = 31$ .

A CD-n lévő program néhány üzenetablakot is megjelenít, melyek segítségével nyomon követhetjük a változók használatát.

### Az onload esemény

Az előző példában a szkriptet a BODY végére tettük, hogy az interpreter a weblap szövegének megjelenítése után hajtsa végre az utasításait. Akkor is ezt az elhelyezést kell választanunk, ha hivatkozni akarunk valamely DHTML-objektumra. A szkriptek szétszórása a kódban megnehezíti programjaink áttekinthetőségét. A *window*-objektum *onload* eseménye lehetővé teszi, hogy az összes szkriptet a HEAD-be írjuk.

Az *onload* esemény akkor következik be, amikor a böngésző teljes egészében betöltötte a dokumentum kódját. A továbbiakban az *onload* eseménykezelő eljárásába fogjuk beleírni azokat az utasításokat, melyeket a weblap megjelenítése után kell végrehajtani. Előnye az is, hogy benne már hivatkozhatunk a DHTML-objektumokra. Így az eseménykezelőt a HEAD-ben lévő szkriptbe tehetjük, nem kell a dokumentum végén egy újabb szkriptet elhelyeznünk.

A **3–46. példa** az *onload* működését mutatja be. Az eseménykezelő megjelenít egy üzenetet, majd a deklarált konstansok segítségével megváltoztatja a háttér színét, és a címsor igazítását. Figyeljük meg, hogy elhelyezése ellenére végrehajtásakor már ismeri a BODY és a címsorobjektumokat!

A **3–47. példa** ugyanezt a tevékenységet végzi el, de az *onload* nélkül. Az utasításokat kénytelenek voltunk a HTML-kód végére helyezni, hogy hivatkozhattunk az objektumokra. Ezzel nagyon szétszórtuk a szkripteket a dokumentumban, megnehezítve az áttekintést. Javasoljuk az Olvasónak, hogy hasonlítsa össze egymással a két megoldás forráskódját!

## A VBScript programok szerkezete

Mint láttuk, szkriptjeinket a HEAD-be és a BODY-ba is tehetjük. Ez utóbbi megoldást akkor alkalmazzuk, ha betöltés közben akarunk hivatkozni a DHTML-objektumokra, vagy a *document.write* metódussal írunk a HTML-kódba. A BODY-ban lévő szkriptek megnehezítik a programok áttekintését, de kényelmesebbé teszik a kód megírását. Ilyen elhelyezést használunk akkor is, ha sok egyforma DHTML-objektumot kell elkészíteni.

A dokumentum betöltése közben a böngésző ellenőrzi a szkriptekben lévő alprogramok kódjának szintaxisát, de nem hajtja végre az utasításait. Az alprogramokon kívül lévő utasítások azonban végrehajtásra kerülnek. Ezek az utasítások alkotják a globális szkripteket.

Globális szkript: a betöltéskor végrehajtásra kerülő utasítások halmaza.

Vegyük észre, hogy a globális változókat éppen a globális szkriptekben deklaráljuk.

A globális szkriptek a HEAD-ben és a BODY-ban is elhelyezkedhetnek. Mind a két helyen tartalmazhatnak alprogramokat. Növeljük a kód áttekinthetőségét, ha az alábbi szerkezetet alkalmazzuk.

A HEAD-be tesszük egymás után:

- a külső kód csatolását végző szkripteket (ha szükséges),  
és azokat a szkripteket, melyek tartalmazzák:
- a globális változók és konstansok deklarációját,
- a globális változók értékadásához szükséges utasításokat,
- további, a dokumentum betöltésénél végrehajtásra kerülő utasításokat.

Ezután következnek a HEAD szkriptjeiben:

- a saját függvények és eljárások,
- az eseménykezelő eljárások, köztük a *window onload* eseménykezelője.

A BODY tartalmazza:

- a weblap megjelenését kialakító DHTML-objektumokat, szövegeket, képeket, szövegmezőket, parancsgombokat stb.,
- azokat a szkripteket, melyek a betöltésnél írnak a HTML-kódba (például egy-egy számított értéket, táblázatot stb.), ezért egyszerűbb az elhelyezésük a BODY-ban.

A CD-n található **3–48. példa** ezt a szerkezetet mutatja be az eddig megismert szkriptek felhasználásával. A csatolt fájlban definiált *VeletlenSzin()* függvény véletlenszerűen állít elő egy színt, és visszaadja annak hexadecimális kódját. Felhasználtuk azt, hogy a megjeleníthető objektumok háttérszínét a *style.backgroundColor* tulajdonsággal állíthatjuk be, például:

```
Gomb.style.backgroundColor = "red"
```

### 3.6. HTML-alkalmazások (HTA)

Programjainkat eddig egy-egy weblap forráskódjában helyeztük el, és a böngésző segítségével futtattuk. A dokumentum a böngésző ablakában jelent meg, rendelkezésünkre állt a szokásos menüsor, eszköztár, állapotsor.

A böngésző ellenőrzi a szkriptek futását. Figyelmeztető üzenetet ad az ablak bezárásakor, az erőforrások lefoglalásakor, a háttértárak kezelésekor. Ez esetenként megnehezíti a munkát. Ha nem az Internetre készítünk weblapokat, akkor fölöslegesnek találhatjuk a böngésző felhasználói felületét is.

#### HTML-alkalmazások készítése

A HTML-alkalmazások használatával dokumentumaink egy szokványos Windows-ablakban jelennek meg, a böngésző által nyújtott felhasználói felület nélkül. Az ablakban csak a programozó által megírt menü, eszköztár, címsor látható.

HTML-alkalmazás készítéséhez egyszerűen változtassuk meg a fájl kiterjesztését .htm-ről .hta-ra. A hta fájlok forráskódját az eddigieknek megfelelően kell megírni, alkalmazhatjuk a DHTML-objektumok és a VBScript teljes eszköztárát. Bármilyen HTML-fájl megnyitható .hta kiterjesztéssel.

A **3–49. példa** bemutatja egy HTML-alkalmazás megjelenítését. A forráskódban csak már ismert HTML-objektumokat használtunk fel. Javasoljuk az Olvasónak, hogy a példafájlokat mentse el .hta kiterjesztéssel, és próbálja ki így is a működésüket!

A hta-állományokat az iexplore (Internet Explorer) helyett az mshta futtatja, így a biztonsági felügyelet is megfelel a szokásos Windows-alkalmazásoknak. Kikerülünk a böngésző szigorú ellenőrzése alól, kényelmesebben használhatók a programok. Nem kapunk figyelmeztetést, ha a szkript be akarja zárni az ablakot vagy hozzányúl a háttértárakhoz. A hta fájlok egyenértékűek a többi Windows alatt futó programmal.

Hta fájlokat az Internetre is fel lehet tenni, de letöltésük során a böngésző a többi fájlhoz megfelelően megkérdezi, hogy megnyissa vagy mentse az állományt. Ha egy ismeretlen helyről származó fájl esetén a megnyitást választjuk, akkor a vírusokhoz hasonlóan kárt tehet a rendszerben. Ez természetesen nem vonatkozik könyvünk példaállományaira.

A hta-fájlokat a későbbiekben a szkriptek futtatásának egyszerűsítésére használjuk.

#### Az ablak tulajdonságainak megadása

A HTML-alkalmazások esetén megadhatjuk a megjelenő ablak tulajdonságait. Ehhez a *HTA:Application*-objektumot használjuk fel, melynek kódját a HEAD-ben kell elhelyezni. Az objektum nem tartalmaz semmilyen elemet, de kötelező kiírni a záró tagját. Ezt helyettesíthetjük a nyitó tag „/>” lezárásával is. Ekkor a „/” jel elé szóközt kell írni:

```
<HTA:Application tulajdonságok />
```

A tulajdonságok értéke sztring, ezért idézőjelet használunk. Az alapértelmezett értékeket nem kell megadni. A tulajdonságokat a szkriptekből is elérhetjük, de nem tudjuk megváltoztatni. A hivatkozás a *HTA:Application*-objektum azonosítójával (*id*) történik.

A tulajdonságokat a **3–50. példa** mutatja be. Felsoroltuk a lehetséges értékeket, és megadtuk az alapértelmezést. Javasoljuk az Olvasónak, hogy próbálja ki a különböző beállításokat! Az ellenőrzéshez elhelyeztünk egy parancsgombot, amely kattintásra megjeleníti az aktuális értékeket.

Megemlítjük, hogy ha letiltjuk a címsort, akkor az Alt + F4 funkcióbillentyűvel lehet bezárni az ablakot.

A *HTA:Application*-objektumot csak az ablak tulajdonságainak megadására használjuk. Ha megfelelnek számunkra az alapértelmezés szerinti értékek, akkor nincs szükség az alkalmazására.

#### Indítás a parancssorból

A Windowsban többféle módon indíthatunk el egy programot, vagy nyithatunk meg egy dokumentumot. Duplán kattinthatunk a fájlra az Intézőben, felvehetjük a Start menübe, elhelyezhetjük a parancsikont az Asztalon, vagy használhatjuk az úgynevezett parancssort (MS-DOS parancssort), melyet a Start/Programok menüből érhetünk el. A kiadott parancsokat egy .bat kiterjesztésű fájlban szintén felsorolhatjuk. Hta fájlok parancssori vagy parancsfájlból történő indítása a Windows 2000-től kezdve lehetséges, parancsikont az előző változatokban is létrehozhatunk.

Parancsikont vagy parancssor segítségével történő indításnál megadhatunk olyan paramétereket, melyeket a HTML-alkalmazás felhasznál a működése közben. A paraméterek használatát a teljesség kedvéért részletezzük. Ha az Olvasó nem kívánja alkalmazni a parancssori indítást, akkor kihagyhatja ezt a részt.

A parancssorba gépelt karaktereket (paramétereket) a *HTA:Application*-objektum *commandLine* tulajdonsága tartalmazza az elérési úttal együtt. A tulajdonság értéke egyetlen sztring.

A **3–51. példa** megjeleníti a *commandLine* tulajdonság értékét. Ha a programot a szokásos módon, az Intézőből indítjuk el, akkor az elérési utat kapjuk meg idézőjelbe téve. Kattintsunk az Intézőben a jobb egérgombbal a fájlra, majd válasszuk a Küldés parancsot. Célként jelöljük meg az Asztalt (parancsikont létrehozása). Így a dokumentumot jelölő ikont megtaláljuk az Asztalon is, ahonnan dupla kattintással tudjuk megnyitni.

Paraméterek megadásához kattintsunk a jobb egérgombbal az Asztalon lévő ikonra, és válasszuk a Tulajdonságok menüpontot. A Cél szövegmezőben az elérési út után hagyjunk egy szóközt, majd gépeljünk be karaktereket. Az OK gombbal zárjuk be a tulajdonságlapot. Ha most megnyitjuk a dokumentumot, akkor a *commandLine* tulajdonság értékénél viszontlátjuk a begépelte karaktersorozatot. Ezt a sztringet felhasználhatjuk a szkriptekben.

Ha a parancssorból indítjuk a fájlt, akkor hasonló módon adhatunk meg, és érhetünk el paramétereket. Szétválasztásuk módját a sztringfüggvényeknél mutatjuk be.

## 4. VEZÉRLŐSZERKEZETEK

Eddigi programjaink utasításait az interpreter sorban *egymás után* végrehajtotta. A legtöbb feladat megoldásánál azonban adódhatnak olyan utasítások, melyeket csak bizonyos feltételek teljesülése, vagy éppen nem teljesülése esetén szükséges feldolgozni. Előfordulhat az is, hogy egy utasításcsoportot többször megismétlünk. Nehézkes lenne mindannyiszor begépelni a kódot. Ráadásul nem mindig tudjuk előre az ismétlések számát.

A program végrehajtását tehát vezérelni kell. Ezt a vezérlőszerkezetek teszik lehetővé.

### 4.1. Szelekciók

A szelekció válogatást jelent. A kiválasztott utasításcsoportot csak egy bizonyos feltétel teljesülése esetén hajtjuk végre. A szelekciót gyakran feltételes elágazásnak hívják.

Szelekció (feltételes elágazás): az utasításcsoport egy meghatározott feltétel teljesülésétől függő végrehajtása.

#### Egyszerű feltételek

Az utasítások végrehajtását szabályozó feltételek a legegyszerűbb esetben két mennyiség összehasonlítását jelentik:

`Magassag < 50, Maradek = 0, ...`

Ezekben a feltételekben relációs (összehasonlító) operátorokat használunk.

*Relációs operátorok:*

*Példa:*

<code>&lt;</code>	kisebb	<code>-2 &lt; 16</code>
<code>&gt;</code>	nagyobb	<code>5 &gt; 3</code>
<code>&lt;=</code>	kisebb vagy egyenlő	<code>4 &lt;= 15 és 6 &lt;= 6</code>
<code>&gt;=</code>	nagyobb vagy egyenlő	<code>-8 &gt;= -9 és 6 &gt;= 6</code>
<code>=</code>	egyenlő	<code>8 = 3 - (-5)</code>
<code>&lt;&gt;</code>	nem egyenlő	<code>2 &lt;&gt; 7</code>

Az operátorok mindkét oldalán tetszőleges kifejezés (tehát önmagában egy változó vagy egy konkrét érték is) állhat. Ezt a szerkezetet relációnak nevezzük.

Reláció: két kifejezés, relációs operátorral összekapcsolva. Értéke igaz (*True*) vagy hamis (*False*).

A relációk alkotják a szelekciós utasítások egyszerű feltételeit.

### Karakterláncok összehasonlítása

A VBScriptben a többi programozási nyelvhez hasonlóan nem csak numerikus értékeket, hanem karaktersorozatokot is összehasonlíthatunk. A sztringek összehasonlítása az ábécérendnek megfelelően történik, így a következő relációk mind igazak:

```
"alma" < "cseresznye", "patak" < "tenger", "barna" < "piros"
```

Az összehasonlítást az interpreter a karakterek ANSI-kódja szerint végzi. A nagybetűk ANSI-kódja kisebb (!), mint a kisbetűké, így a nagybetűk előbb állnak a sorban:

```
"G" < "g", "Virág" < "virág"
```

Az azonos betűkkel kezdődő sztringek esetén az első eltérő karakter számít. Ha az egyik sztring közben véget ér, akkor a hosszabb lesz a nagyobb:

```
"pongyola" < "pont", "Kati" < "Katica", "kaTica" < "katIca",  
de: "Katica" < "kati"
```

Az összehasonlításra kerülő sztringek nem csak betűket, hanem számokat és más karaktereket is tartalmazhatnak. Sorrendjüket ekkor is az ANSI-kódok határozzák meg. Ilyen kódtáblázat a CD-melléklet Dokumentumok mappájában található.

Ha számokból álló sztringeket hasonlítunk össze, akkor meglepő eredményeket kaphatunk. Az interpreter ugyanis karakterenként végzi el az összehasonlítást. Amelyik sztringben először talál nagyobbat, az a nagyobb (az idézőjellel hangsúlyozzuk, hogy nem numerikus értékekről, hanem karaktersorozatokról van szó):

```
"12" < "3", "12356" < "124", "" < "0", de "1234" < "12345"
```

A sztringek összehasonlítását fel lehet használni az ábécé szerinti rendezéshez. Sajnos az ANSI-kódok sorában a magyar ékezetes magánhangzók (á, é, í, ó, ő, ö, ú, ü, ű) az összes többi betű után helyezkednek el. Az ékezetes karakterek megszokott rendezéséhez az *StrComp* függvényre van szükség.

A relációs operátorok alkalmazásánál arra is figyelniünk kell, hogy az adatbevitel szöveges formában történik. A VBScript a numerikus változók bármely értékét kisebbnek tekinti a karaktersorozatoknál, így például a

```
Szam = 1234
```

esetén a

```
Szam < Szovegmezo.value
```

mindig igaz, akármit írtunk a szövegmezőbe. Ezért a beolvasott adatot át kell alakítani numerikus értéké:

```
Szam < CSng(Szovegmezo.value)
```

Megjegyezzük, hogy a szövegmező tartalmát számokkal már közvetlenül össze lehet hasonlítani, és aritmetikai műveletet szintén végezhetünk vele, így az

```
1234 < Szovegmezo.value vagy a  
Szovegmezo.value * 2 > Szam
```

utasítások konverzió nélkül is helyes eredményre vezetnek. Célszerű azonban mindig átalakítani számmá a szövegmező numerikus tartalmát!

## Összehasonlítás az StrComp függvénnyel

A magyar ékezetes magánhangzókat tartalmazó karakterláncok összehasonlítására az *StrComp* (string compare, sztring-összehasonlítás) függvényt használhatjuk:

```
StrComp(Sztring_1, Sztring_2, mód)
```

A *mód* paraméter értéke és jelentése:

- 0: bájttonkénti összehasonlítás, megkülönbözteti a kis- és nagybetűket, sorrend az ANSI-kódok szerint
- 1: szövegszerű összehasonlítás, nem különbözteti meg a kis- és nagybetűket, sorrend a magyar ábécé szerint

A függvény visszatérési értéke és jelentése:

- 1: *Sztring\_1* < *Sztring\_2*
- 0: *Sztring\_1* = *Sztring\_2*
- 1: *Sztring\_1* > *Sztring\_2*

A függvényt a **4–1. példa** weblapján próbálhatjuk ki. Néhány példa az alkalmazására:

StrComp("Katica", "katica", 0) = -1	"Katica" < "katica"
StrComp("Katica", "katica", 1) = 0	"Katica" = "katica"
StrComp("Ádám", "Béla", 0) = 1	"Ádám" > "Béla"
StrComp("Ádám", "Béla", 1) = -1	"Ádám" < "Béla"
StrComp("ádám", "Béla", 1) = -1	"ádám" < "Béla"

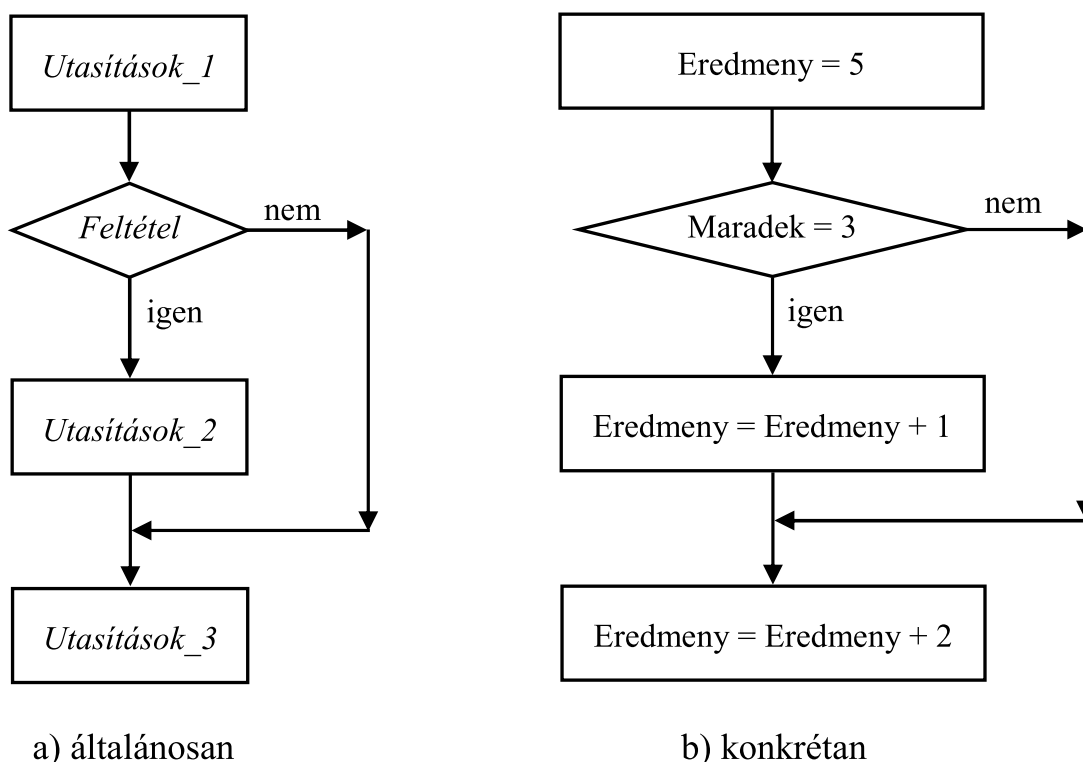
Mint látjuk, a szövegszerű összehasonlítás a magyar ábécének megfelelő sorrendben történik, de ekkor nem tudjuk megkülönböztetni a kis- és nagybetűket egymástól.

## Egyágú szelekció

Az egyágú szelekció azt jelenti, hogy ha igaz a feltétel, akkor a kiválasztott utasítássorozatot végre kell hajtani, egyébként pedig nem. A szelekciós rész előtt és után következő utasítások mindkét esetben végrehajtódnak. Az egyágú szelekciót folyamatábrával szemléltethetjük, ahol az utasításokat téglalapokba, a feltételt pedig egy csúcsára állított rombuszba írjuk (4–1. ábra).

Ha a 4–1. a) ábrán látható esetben a feltétel teljesül, akkor az *Utasítások\_1* után az *Utasítások\_2* hajtódik végre, majd az *Utasítások\_3*. Ha a feltétel nem teljesül, akkor az *Utasítások\_2* kimarad, és az *Utasítások\_1* után rögtön az *Utasítások\_3* következik.

Ha a 4–1. b) ábrán szereplő konkrét példában a *Maradek* = 3, akkor az *Eredmeny*-hez először hozzáadunk 1-et, majd a szelekció után még 2-t. Értéke tehát 8 lesz. Ha a *Maradek* nem egyenlő 3-mal, akkor csak az utolsó összeadás hajtódik végre, így értéke 7 lesz.



4–1. ábra. Az egyágú szelekció folyamatábrája

### Az If ... Then utasítás

Az egyágú szelekciót a VBScriptben az *If ... Then* (ha ... akkor) utasítással valósíthatjuk meg. Szintaxisa:

```

If Feltétel Then
    Utasítások
End If

```

Ha a *Feltétel* teljesül, akkor a *Then* és az *End If* között lévő utasítások végrehajtódnak, egyébként pedig nem.

A 4–1. b) folyamatábra VBScript kódja például:

```

Eredmeny = 5
If Maradek = 3 Then
    Eredmeny = Eredmeny + 1
End If
Eredmeny = Eredmeny + 2

```

Figyeljük meg, hogy a szelekcióban szereplő utasítást beljebb kezdtük, így jól látszik az *If* és az *End If* által közrefogott rész!

A feltételes elágazást áttekinthetőbbé tehetjük egy mondatszerű leírással, amelyben megtartjuk a VBScript szerkezetét, de magyarul fogalmazunk:

Utásítások_1	Eredmeny = 5
Ha Feltétel Akkor	Ha Maradek = 3 Akkor
Utásítások_2	Eredmeny = Eredmeny + 1
Elágazás vége	Elágazás vége
Utásítások_3	Eredmeny = Eredmeny + 2



Az ilyen úgynevezett pszeudokód bonyolultabb szerkezetek esetén nagymértékben segíti a program megtervezését, a hibák felderítését és javítását.

### Példa az egyágú szelekcióra

A 4–2. példa olyan weblapot mutat be, amely a felhasználó által megadott értékkel kiszámítja az  $x^2$ , a  $\sqrt{x}$ , a  $2x+1$  és az  $1/x$  függvények értékét.

A beolvasáshoz és megjelenítéshez egy szövegmezőt, egy parancsgombot és egy üres DIV-objektumot helyezünk el a HTML-kódban:

```
x = <INPUT id = "X_be" type = "text" value = 0>
<INPUT id = "Gomb" type = "button" value = "Számítás!">
<DIV id = "ListaKi"></DIV>
```

Az eseménykezelő szkriptben egy változóba írjuk a szövegmező értékét, hogy rövidebben hivatkozhatunk rá:

```
X = X_be.value
```

A kiszámított függvényértékek felsorolását a *Lista* nevű sztringbe gyűjtjük össze, amit a DIV segítségével jelenítünk meg. A sort az  $x^2$ -tel kezdjük. A függvényérték elé elhelyezzük a magyarázó feliratot, a tagoláshoz pedig bekezdésobjektumot alkalmazunk:

```
Lista = "<P>x&sup2; = " & X^2 & "</P>"
```

A négyzetgyök értékét ehhez a sztringhez fűzzük hozzá. Az *&radic;* a gyökjel karakter entitása:

```
Lista = Lista & "<P>&radic;x = " & Sqr(X) & "</P>"
```

Negatív számból azonban nem tudunk négyzetgyököt vonni, ezért ezt a műveletet csak akkor végezzük el, ha a felhasználó nem negatív számot írt a szövegmezőbe:

```
If X >= 0 Then
    Lista = Lista & "<P>&radic;x = " & Sqr(X) & "</P>"
End If
```

Az *If* utasítás hatására a négyzetgyökvonás csak akkor kerül végrehajtásra (majd kiírásra), ha a szám nagyobb vagy egyenlő, mint nulla.

Következik a  $2x+1$  értékének a hozzáfűzése:

```
Lista = Lista & "<P>2x + 1 = " & 2*X + 1 & "</P>"
```

Végül az  $1/x$  értékét akkor tudjuk hozzáfűzni, ha az  $x$  nem 0:

```
If X <> 0 Then
    Lista = Lista & "<P>1/x = " & 1/X & "</P>"
End If
ListaKi.innerHTML = Lista
```

Próbáljuk ki a programot különböző pozitív, negatív értékekkel és nullával!

A 4–2. példa kódjában a szövegmező nyitó tagját kiegészítettük az *onkeydown* esemény kezelőjével, amely az  $x$  változtatásakor törli a DIV-objektum tartalmát, hogy ne

legyen félrevezető a megjelenítés (a régi függvényértékek az új x-szel). Mivel az eseménykezelés egyetlen értékadó utasításból áll, nem készítettünk külön szkriptet.

Megjegyezzük, hogy az *X* és a *Lista* nevű változókat a szubrutinban deklaráltuk (lokális változók). Ezek nem érhetők el az eljárásen kívül, de itt nincs is erre szükség.

### Számkitaláló játék

A feltételes elágazás segítségével a **4–3. példában** versenyre hívjuk a felhasználót, hány próbálgatással tud kitalálni egy 1 és 100 közé eső, véletlenszerűen választott számot. Weblapunk a *TippBe* szövegmezőt, a *Gomb* parancsgombot és a *Valasz* bekezdést tartalmazza. A globális szkript elején előállítunk egy véletlenszámot:

```
Randomize  
Véletlenszam = Int(100 * Rnd() + 1)
```

Emlékezzünk vissza arra, hogy a szkriptben szereplő utasításokat a böngésző a weblap megjelenítésekor hajtja végre. Így új véletlenszám választásához, azaz új játékhoz frissíteni kell a dokumentumot.

A parancsgomb eseménykezelőjében megvizsgáljuk, hogy a tipp értéke nagyobb-e, mint a program által előállított véletlenszám. (Mivel a szövegmező tartalma sztring, ezért numerikus értékke alakítjuk.) Ha a tipp nagyobb, akkor túl nagy számot választottunk:

```
Tipp = CSng(TippBe.value)  
If Tipp > Veletlenszam Then  
    Valasz.innerText = "Túl nagy!"  
End If
```

Ha a tipp kisebb, akkor túl kicsi a szám:

```
If Tipp < Veletlenszam Then  
    Valasz.innerText = "Túl kicsi!"  
End If
```

Ha a tipp egyenlő a véletlenszámmal, akkor éppen eltaláltuk:

```
If Tipp = Veletlenszam Then  
    Valasz.innerHTML = "<B>ELTALÁLTA!</B>"  
End If
```

Egyszerű játékprogramunk hibátlanul működik, de felesleges vizsgálatokat végez. A szelekciók feltételei ugyanis kizárják egymást, egyszerre csak az egyik lehet igaz közülük. Ha például túl nagy értéket adunk meg, akkor a második és a harmadik *If* utasítás feltételét már egyáltalán nem kellene ellenőrizni. Ez lassítja a programfutást, és bonyolultabbá teszi a kódot. Ezt a hiányosságot a többágú szelekcióval fogjuk kiküszöbölni.

### A találgatások számlálása

Az előző példa programját a **4–4. példában** úgy alakítjuk át, hogy számlálja a próbálgatásainkat. Ehhez egy újabb, *SzamlaloKi* azonosítójú SPAN-objektumot helyezünk el a weblapon, a betöltéskor pedig létrehozunk egy *Szamlalo* nevű változót, aminek 0 kezdőértéket adunk:

```
Szamlalo = 0
```

Az eseménykezelő szubrutin minden egyes hívásakor megnöveljük a *Szamlalo* értékét:

```
Szamlalo = Szamlalo + 1
```

Ezek után már csak a megjelenítéséről kell gondoskodnunk.

A programozás során gyakran kell számlálást végeznünk. Az algoritmus hasonló a példában alkalmazott eljáráshoz:

```
Szamlalo = 0
```

```
...
```

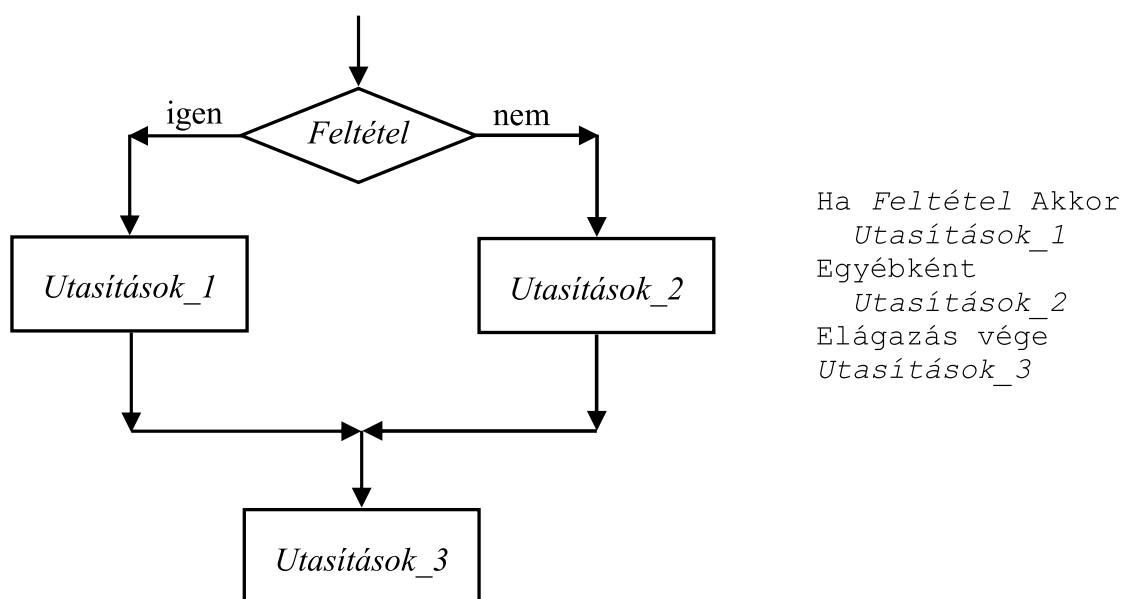
```
Szamlalo = Szamlalo + 1
```

A számláló növelését a számlálandó tevékenység elé és után is helyezhetjük, de csak a növelés után mutatja a helyes értéket. Ne feledkezzünk meg az elején a nullázásról. Az ilyen kezdőértékadást a programozásban gyakran a változó inicializálásának is nevezik.

Inicializálás: a változó kezdőértékének megadása.

## Kétágú szelekció

A feltételes elágazásoknál gyakran előfordul, hogy a feltétel teljesülésekor az egyik, nem teljesülésekor pedig egy másik utasításcsoportot kell végrehajtani, de a kettőt együtt soha. Ekkor kétágú szelekcióról beszélünk, melynek folyamatábrája és pszeudokódja:



4–2. ábra. A kétágú szelekció folyamatábrája és pszeudokódja

Ha a feltétel teljesül, akkor az *Utasítások\_1* csoport hajtódik végre, ha nem teljesül, akkor pedig az *Utasítások\_2*. Mindkét esetben az *Utasítások\_3*-mal folytatódik a program.

A VBScriptben a kétágú szelekciót az *If ... Then ... Else* (ha ... akkor ... egyébként) utasítás valósítja meg:

```
If Feltétel Then
    Utasítások_1
Else
    Utasítások_2
End If
```

Figyeljük meg a kód strukturálását! Az *If*, az *Else* és az *End If* ugyanabban az oszlopban kezdődik, az utasításokat viszont beljebb írjuk. Ezzel egyértelműen kiemeljük a kétágú szelekció szerkezetét.

### A kétágú szelekció alkalmazása

A kétágú szelekció alkalmazását az osztás példáján mutatjuk be. Minden osztásnál a végrehajtás előtt meg kell vizsgálni, hogy a nevező nulla-e. Ebben az esetben ugyanis nem szabad elvégezni a műveletet, mert hibaüzenettel leáll a programunk. A **4–5. példában** létrehozunk két szövegmezőt és egy parancsgombot. A parancsgomb eseménykezelője hibaüzenetet küld, ha a nevező 0, egyébként pedig elvégzi az osztást:

```
If Nevezo.value = 0 Then
    window.alert("Nullával nem lehet osztani!")
Else
    Hanyados.innerText = Szamlalo.value / Nevezo.value
End If
```

A **4–6. példában** egyetlen parancsgomb segítségével váltogatjuk a háttér színét. Ehhez definiáljuk a két színnek megfelelő hexadecimális konstanszt:

```
Const Kek = "#0000ff"
Const Voros = "#ff0000"
```

A parancsgomb eseménykezelőjében a szelekció feltétele az aktuális háttérszín vizsgálata lesz. Ha kék színű, akkor vörösre váltjuk, egyébként pedig (vagyis ha vörös, akkor) kékre:

```
If document.bgColor = Kek Then
    document.bgColor = Voros
Else
    document.bgColor = Kek
End If
```

### Az elágazások egyszerűsítése

A szelekciók elengedhetetlen kellékei a programírásnak, de megnehezítik a kód áttekintését, és a hibák felismerését, ezért törekednünk kell a szerkezet egyszerűsítésére. Ha például két szám közül kell kiválasztani a nagyobbát, akkor nyilvánvalónak tűnik a kétágú szelekció alkalmazása:

```
If ElsoSzam < MasodikSzam Then
    Maximum = MasodikSzam
Else
    Maximum = ElsoSzam
End If
```

A feladatot azonban egyágú szelekcióval is megoldhatjuk. Válasszuk először az első számot maximumnak, majd hasonlítsuk össze a második számmal. Ha rosszul döntöttünk, akkor vegyük maximumnak a második számot:

```
Maximum = ElsoSzam
If ElsoSzam < MasodikSzam Then
    Maximum = MasodikSzam
End If
```

Ez a szerkezet nem csak egyszerűsíti a kódot, hanem gyorsítja a végrehajtást is, mert a szelekciók fordítása időigényes folyamat az interpreter számára. Fenti példánk egy gyakran alkalmazott programozói fogást mutatott be. Az értékadás, illetve szükség esetén a módosítás áttekinthetőbbé és hatékonyabbá teszi a programot.

A **4–7. példában** megvizsgáljuk, hogy a felhasználó által a szövegmezőbe írt szám osztható-e 7-tel. Egy szám akkor osztható 7-tel, ha az osztás maradéka 0. Mint tudjuk, a maradékot a *Mod* művelet állítja elő. A programírásnál megint kétágú szelekcióval próbálkozunk:

```
If Szam.value Mod 7 = 0 Then
    Valasz.innerText = "osztható 7-tel"
Else
    Valasz.innerText = "nem osztható 7-tel"
End If
```

Vegyük azonban észre, hogy az oszthatóságnak megfelelő üzenet a „nem” szóval kiegészítve megadja a másik üzenetet. Így először feltételezzük, hogy a szám osztható, majd egyágú szelekcióval szükség esetén hozzáfűzzük az üzenethez a módosítást. A választ célszerű először egy külön sztringben tárolni:

```
Valasz = "osztható 7-tel"
If Szam.value Mod 7 <> 0 Then
    Valasz = "nem " & Valasz
End If
ValaszKi.innerText = Valasz
```

Figyeljünk a „nem” szócskát követő szóközre az üzenet összeállításánál!

A szelekcióban szereplő értékadást ennél az egyszerű példánál természetesen helyettesíthettük volna a `Valasz = "nem osztható 7-tel"` utasítással is. De ha a *Valasz* értékét egy hosszabb algoritmus határozza meg, akkor már nem tudjuk ezt az értékadást könnyen elvégezni.

## A szelekciós ágak cseréje

A kétágú szelekciónál a feltétel megfogalmazásától függ, hogy melyik utasítássorozat kerül a *Then*, és melyik az *Else* ágba. Ha ellentétesre változtatjuk a feltételt (azaz akkor lesz hamis, amikor eredetileg igaz volt), fel kell cserélnünk a két ágot. Az alábbi kódok például egymással egyenértékűek:

<pre>If Nevező = 0 Then     hibaüzenet küldése Else     számítások elvégzése End If</pre>	<pre>If Nevező &lt;&gt; 0 Then     számítások elvégzése Else     hibaüzenet küldése End If</pre>
---	--

Ha az egyik ág jóval rövidebb, mint a másik, akkor célszerű a *Then* után tenni. Így könnyebben megtaláljuk az *Else* ágot, áttekinthetőbb lesz a kód. A fenti példa esetén a hibaüzenet valószínűleg rövidebb, mint a számításokat végző utasítássorozat, így az első megoldást részesítjük előnyben.

## Egymásba ágyazott szelekciók

A szelekciók egyes ágaiban tetszőleges utasítássorozatok szerepelhetnek, akár további szelekciókat is írhatunk. Ennek bemutatására a **4–8. példában** megoldjuk az  $a \cdot x = b$  elsőfokú egyenletet. Ha az „a” együttható értéke nem 0, akkor oszthatunk vele, és a megoldás:

$$x = \frac{b}{a}$$

Ha az „a” értéke 0, akkor két esetet kell megkülönböztetnünk. A  $0 \cdot x = 6$  típusú egyenleteknek (tehát ha a „b” nem 0) nincs megoldása, a  $0 \cdot x = 0$  egyenlet esetén pedig az  $x$  értéke tetszőleges lehet. Mivel a felhasználó bármely értéket megadhat az együtthatóknál, minden esetre fel kell készülnünk. A megoldás pszeudokódja és VBScript programja:

<pre>Ha a &lt;&gt; 0 Akkor     x = b/a Egyébként     Ha b = 0 Akkor         minden szám megoldás     Egyébként         nincs megoldás     Elágazás vége Elágazás vége</pre>	<pre>If A.value &lt;&gt; 0 Then     X = "x = " &amp; B.value / A.value Else     If B.value = 0 Then         X = "Minden szám megoldás."     Else         X = "Nincs megoldás."     End If End If</pre>
---	--

Weblapunkra két szövegmező és egy parancsgomb kerül. A változóneveknél megtartottuk a fenti képletekben alkalmazott jelöléseket. Az egymásba ágyazott szelekciókat a parancsgomb *onclick* eseménykezelője tartalmazza.

A kiíráshoz az  $X$  változót egyszerűen át kell adni egy bekezdés *innerText* tulajdonságának.

## 4.2. Többágú szelekciók

### Többágú szelekció

Az elsőfokú egyenlet megoldásánál a külső szelekció relációs feltétele az „a” együttthatóra vonatkozott. Ha értéke 0 volt, akkor a megoldáshoz meg kellett vizsgálnunk egy másik változó, a „b” értékét. Sok esetben van szükség egymás után többféle feltétel kiértékelésére úgy, hogy csak az egyikhez tartozó utasítások kerüljenek végrehajtásra. Ekkor az egymásba ágyazott feltételes elágazások helyett célszerű többágú szelekciót alkalmazni.

A többágú szelekciót az *If ... ElseIf* (ha ... egyébként ha) utasítás valósítja meg, melynek pszeudokódja és VBScript szintaxisa:

Ha <i>Feltétel_1</i> Akkor	<i>If Feltétel_1 Then</i>
<i>Utasítások_1</i>	<i>Utasítások_1</i>
Egyébként ha <i>Feltétel_2</i> Akkor	<i>ElseIf Feltétel_2 Then</i>
<i>Utasítások_2</i>	<i>Utasítások_2</i>
...	...
Egyébként	<i>Else</i>
<i>Utasítások_n</i>	<i>Utasítások_n</i>
Elágazás vége	<i>End If</i>

Ha a *Feltétel\_1* teljesül, akkor az *Utasítások\_1* csoport hajtódik végre. Egyébként, ha a *Feltétel\_2* teljesül, akkor az *Utasítások\_2* és így tovább. A végén álló *Else* ág *Utasítások\_n* csoportjára akkor kerül a vezérlés, ha előtte egyetlen feltétel sem teljesült. Felhívjuk a figyelmet arra, hogy az *ElseIf* kulcsszót egybe kell írni, mert különben a többágú szelekció helyett egymásba ágyazott elágazásként kerülne kiértékelésre!

Az *Else* ág alkalmazása nem kötelező. Hiánya esetén ha egyetlen feltétel sem teljesül, akkor egyetlen utasítás sem hajtódik végre.

Ha a többágú szelekció bármelyik feltétele teljesül, akkor a többit már nem vizsgálja meg az interpreter. Így csak egyetlen ág utasításcsoportja kerül végrehajtásra. Ezért a többágú szelekciót általában egy kifejezésre vonatkozó, egymást kizáró feltételek esetén szoktuk alkalmazni.

A 4–9. példában a számkitaláló játék három feltételes elágazását egyetlen többágú szelekcióval helyettesítjük. Ezzel elkerüljük, hogy egy feltétel teljesülése esetén a többi fölöslegesen kiértékelődjön:

```
If Tipp > Veletlenszam Then
    Valasz.innerText = "Túl nagy!"
ElseIf Tipp < Veletlenszam Then
    Valasz.innerText = "Túl kicsi!"
Else
    Valasz.innerHTML = "<B>ELTALÁLTA!</B>"
End If
```

Az utolsó elágazásnál felesleges lett volna a relációs feltételt kiírni, mert ha a különbség nem nagyobb és nem is kisebb, mint a *Veletlenszam*, akkor csak egyenlő lehet vele. Ezért *Else* ágot alkalmaztunk.

### Példa a többágú szelekcióra

A többágú szelekció egyes feltételeinek egyszerűsítéséhez kihasználhatjuk azt a viselkedést, hogy csak az egyik ág utasításai hajtódik végre. Egy intervallum esetén például gyakran elegendő csak az egyik végponttal összehasonlítani a változó értékét.

A **4–10. példában** olyan programot készítünk, amely bizonyítványunk átlageredménye alapján értékeli a teljesítményünket. Az értékelést a következő táblázat mutatja:

Átlag:	Értékelés:
kisebb, mint 2,0	Sajnos megbukott!
$2,0 \leq \text{átlag} < 2,9$	Elég gyöngé a teljesítménye.
$2,9 \leq \text{átlag} < 3,9$	Ez bizony csak közepes.
$3,9 \leq \text{átlag} < 4,5$	Egyre jobb lesz.
$4,5 \leq \text{átlag} < 5$	Gratulálok a jeleshez!
5	Kitűnő a bizonyítvány!

Mivel több összehasonlítást kell végeznünk, a beolvasáshoz használt szövegmező értékét itt is átadjuk egy változónak, így egyszerűbben tudunk hivatkozni rá:

```
Atlag = AtlagBe.value
```

Az összehasonlításokat egy összetett *If ... ElseIf* utasítással végezzük:

```
If Atlag < 2 Then
    Uzenet = "Sajnos megbukott!"
ElseIf Atlag < 2.9 Then
    Uzenet = "Elég gyöngé a teljesítménye!"
ElseIf Atlag < 3.9 Then
    Uzenet = "Ez bizony csak közepes."
ElseIf Atlag < 4.5 Then
    Uzenet = "Egyre jobb lesz."
ElseIf Atlag < 5 Then
    Uzenet = "Gratulálok a jeleshez!"
Else
    Uzenet = "Kitűnő a bizonyítvány!"
End If
```

Egy reláció teljesülése esetén a többit már nem vizsgálja meg az interpreter, így ha az átlag 3,2, akkor a *Szoveg* az „Ez bizony csak közepes.” értéket kapja. Bár a további relációk is teljesülnének, a vezérlés már nem kerül rájuk. Az utolsó ágban kihasználtuk, hogy ha egyik addigi feltétel sem teljesül, akkor az átlag már csak 5-ös lehet.

Az értékadás után egy bekezdésobjektum segítségével megjelenítjük az *Uzenet* karakterláncot:

```
UzenetKi.innerText = Uzenet
```

Megjegyezzük, hogy az *Uzenet* változó alkalmazása helyett a szelekció egyes ágaiban is beírhattuk volna az üzenetet a bekezdésbe.



## A bevitel ellenőrzése az Else ág segítségével

A programozónak mindig számítania kell arra, hogy a begépett értékek nem felelnek meg az elvárásoknak. Az előző példában a felhasználó beírhat -3-at is a szövegmezőbe, amire a „Sajnos megbukott!” üzenetet kapja. Ha 1300-at ad meg átlagként, akkor pedig közöljük vele, hogy kitűnő a bizonyítványa.

A meg nem engedett értékek kezeléséhez egészítsük ki az *If ... ElseIf* utasítást az elején az

```
If Atlag < 1 Then
    Uzenet = "Ennyire rossz nem lehet!"
ElseIf Atlag < 2 Then
    Uzenet = "Sajnos megbukott!"
```

a végén pedig az

```
ElseIf Atlag = 5 Then
    Uzenet = "Kitűnő a bizonyítvány!"
Else
    Uzenet = "Elég furcsán osztályoznak!"
End If
```

sorokkal. Így már csak valószínű eredményeket fogad el. A módosított kódot a **4–11. példa** tartalmazza.

## A vizsgálatok sorrendje

A fenti példákban nem cserélhettük fel a vizsgálatok sorrendjét, mert a feltételek átfedték egymást. Ha az elejére tesszük az *Atlag < 5* feltételt, akkor a bukott tanulót is jelesre értékelnénk. A fenti megoldásban azonban a "Sajnos megbukott!" értékelés után a vezérlés már nem kerül át a többi *ElseIf* ágra.

Az egymásba ágyazott szelekciók szerkezetét gyakran lehet egyszerűsíteni a vizsgálatok sorrendjének megváltoztatásával. Sok esetben a *Then* után álló *If* majdnem biztosan logikai bukfencre utal.

Egy nagykereskedő például a következő kód szerint többféle kedvezményt ad üzletfeleinek a vásárolt darabszám után:

```
Kedvezmeny = 0
If Darab > 100 Then
    If Darab > 200 Then
        If Darab > 500 Then
            Kedvezmeny = 20
        Else
            Kedvezmeny = 10
        End If
    Else
        Kedvezmeny = 5
    End If
End If
```

Végezzük el fordított sorrendben a vizsgálatokat, és mindjárt világosabb szerkezetű programot kapunk. Ha a darabszám túllépi az 500-at, akkor a kereskedő 20 % kedvez-

ményt ad. 200 és 500 között a kedvezmény 10 %, 100 és 200 között pedig 5 %. Ezt tipikusan egy *If ... ElseIf* utasítással fejezhetjük ki:

```
If Darab > 500 Then
    Kedvezmeny = 20
ElseIf Darab > 200 Then
    Kedvezmeny = 10
ElseIf Darab > 100 Then
    Kedvezmeny = 5
Else
    Kedvezmeny = 0
End If
```

Általában is törekedjünk arra, hogy a *Then* utasítás után ne egy újabb *If* álljon, hanem olyan tevékenység, melyet a feltétel teljesülésénél el kell végezni. A vizsgálatot cselekvés kövesse, ne pedig egy újabb vizsgálat!

A fentihez hasonló besorolások esetén a határookra is gondot kell fordítani. Ha a darabszám éppen 200, akkor a kedvezmény még 20 %, vagy már csak 10 %? A megkülönböztetéshez a szigorúan nagyobb, illetve a nagyobb vagy egyenlő relációs operátort alkalmazzuk.

### A másodfokú egyenlet megoldása

A szelekciók gyakorlásaként a **4–12. példában** elkészítünk egy nagyobb lélegzetű programot, amely megoldja az  $ax^2 + bx + c = 0$  másodfokú egyenletet az ismert

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

megoldóképlet segítségével. Ha  $a = 0$ , akkor nem másodfokú az egyenlet. Ekkor az elsőfokú egyenlet megoldásának kódját használjuk fel, az együtthatók jelölésének átírásával. Ha az „a” nem nulla, a gyökvonás elvégzése előtt további vizsgálatot kell végeznünk. A matematikában a gyökjel alatt lévő kifejezést diszkriminánsnak hívják. Ha negatív, akkor nincs (valós) megoldása az egyenletnek, ha 0, akkor pedig csak egyetlen megoldás létezik.

A fentiek alapján könnyen elkészíthetjük a pszeudokódot. A megjegyzéseket aposztróffal jelöltük:

```
Ha a <> 0 Akkor ' másodfokú az egyenlet
    a Diszkrimináns meghatározása
    Ha Diszkrimináns < 0 Akkor ' nincs megoldás
        üzenet kiírása
    Egyébként ha Diszkrimináns = 0 Akkor ' egy megoldás van
        az egyetlen gyök meghatározása és kiírása
    Egyébként ' két megoldás van
        a két gyök meghatározása és kiírása
    Elágazás vége
Egyébként ' nem másodfokú az egyenlet
    az elsőfokú egyenlet megoldása
Elágazás vége
```

A strukturált elrendezésből jól látható, hogy minden egyes elágazásnak megvan a *Ha*, *Egyébként* és *Elágazás vége* része. A VBScript forráskód megtalálható a CD-mellékleten. Javasoljuk az Olvasónak, hogy alaposan gondolja át minden sorát!

## A Select Case utasítás

Ha a többágú szelekcióban szereplő relációk csak egyenlőségjelet tartalmaznak, akkor az *If ... ElseIf* helyet egy jobban áttekinthető utasítást is alkalmazhatunk. A *Select Case* (esetkiválasztás) pszeudokódja és VBScript szintaxisa:

Elágazás <i>TesztKifejezés</i> szerint	<i>Select Case TesztKifejezés</i>
<i>Kifejezéslista_1</i> esetén	<i>Case Kifejezéslista_1</i>
<i>Utasítások_1</i>	<i>Utasítások_1</i>
<i>Kifejezéslista_2</i> esetén	<i>Case Kifejezéslista_2</i>
<i>Utasítások_2</i>	<i>Utasítások_2</i>
...	...
Egyébként	<i>Case Else</i>
<i>Utasítások</i>	<i>Utasítások</i>
Elágazás vége	<i>End Select</i>

Az interpreter megkeresi azt a kifejezéslistát, amelyben szerepel a teszt kifejezés konkrét értéke, és a kifejezéslistának megfelelő utasításokat hajtja végre. Ha egyik ág sem felel meg a teszt kifejezésnek, akkor a *Case Else* (egyébként) ág utasításaira tér rá. A *Case Else* ág használata nem kötelező – ebben az esetben ha nincs találat, akkor a szelekció hatástalan!

A kifejezéslista elemeit vesszővel kell egymástól elválasztani.

A *Select Case* végrehajtása során csak egyetlen ág utasításai hajtnak végre, ezután a feldolgozás az *End Select*-et követő utasítással folytatódik. Így ha több kifejezéslista tartalmazza a teszt kifejezés értékét, akkor is csak az először sorra kerülő ág utasításai lesznek végrehajtva.

Ha egy *Case* ágba nem írunk utasításokat, akkor a feltétel teljesülése esetén az *End Select* után folytatódik a program. Ezt akkor használjuk ki, ha a teszt kifejezés bizonyos értékeit ki akarjuk hagyni a feldolgozásból.

A *Select Case* utasítás bemutatásához a **4–13. példában** olyan programot készítünk, amely egyetlen parancsgomb segítségével egymás után négyféle háttérszín között váltogat. A betöltésnél deklaráljuk a *Kék*, *Vörös*, *Zöld*, *Sarga* konstansokat. A parancsgomb eseménykezelőjében már csak módosítani kell egymás után a színeket:

```
Select Case document.bgColor
Case Kek
    document.bgColor = Vörös
Case Vörös
    document.bgColor = Zöld
Case Zöld
    document.bgColor = Sarga
Case Else
    document.bgColor = Kek
End Select
```

Az interpreter összehasonlítja a teszt kifejezésként megadott *document.bgColor* értékét a *Case* kulcsszavak után álló értékekkel. Az első egyezés esetén végrehajtja az utána következő értékadó utasítást, azaz megváltoztatja a színt.

Az utolsó ágban kihasználtuk, hogy ha egyik feltétel sem teljesül, akkor a háttérszín már csak sárga lehet. Ezt a feladatot tömbök alkalmazásával egy kicsit egyszerűbben is meg tudjuk majd oldani.

### A Select Case alkalmazása

A 4–14. példában olyan programot készítünk, amely megadja egy hónap napjainak a számát. A hónap nevét a *Honap* azonosítójú szövegmezőbe gépeljük be. A *Select Case* utasítás teszt kifejezése a szövegmező *value* tulajdonsága lesz. A *Case*-ek kifejezéslistája az egyforma hosszúságú hónapokat sorolja fel, az utánuk következő utasítás pedig megadja a napok számát:

```
Select Case Honap.value
  Case "február"
    NapokSzama = "28 vagy 29"
  Case "április", "június", "szeptember", "november"
    NapokSzama = 30
  Case "január", "március", "május", "július", "augusztus", _
    "október", "december"
    NapokSzama = 31
  Case Else
    NapokSzama = "???"
End Select
```

Mivel a sztringek összehasonlításánál az interpreter megkülönbözteti a kis- és nagybetűket, a szövegmezőbe kisbetűkkel kell beírni a hónapok nevét.

Az *Else* ágot célszerű felhasználni a begépelte karaktersorozat ellenőrzésére. Ha a szövegmező tartalma egyik hónapnak sem felel meg, akkor programunk ezt négy kérdőjel kiírásával jelzi.

### A focus metódus

Az előző példában kissé kényelmetlen volt egy új adat begépeléséhez rákattintani a szövegmezőre, kijelölni és törölni a benne lévő karaktersorozatot. A *window*-objektum *focus* és *select* metódusainak használatával megkönnyíthetjük a felhasználó dolgát.

A Windows operációs rendszerben egyszerre több programot is futtathatunk, de csak az egyik alkalmazás lehet aktív. Az aktív alkalmazáson belül pedig általában egérr kattintással jelölhetjük ki azt az elemet (objektumot), amely elsőként reagál a bekövetkező eseményekre.

Ha egy szövegmezőre kattintunk, akkor egy villogó kurzor jelenik meg benne, jelezve, hogy a odakerül a begépelte szöveg. Parancsgombok esetén vastagabb keret mutatja a kijelölést. Ekkor az Enter billentyű lenyomása egyenértékű a gombra való egérr kattintással.

Az objektumoknak a fentiekben leírt állapotát úgy fejezzük ki, hogy megkapták a fókuszt. A bekövetkező eseményeket az az objektum érzékeli, amelyikre a fókuszt került.<sup>17</sup>

Fókuszt: egy objektum kijelölt állapota.

Mint ismeretes, a Windows-ban a tabulátor billentyű lenyomásával is lépegethünk az objektumok (például a szövegmezők vagy parancsgombok) között, továbbadhatjuk a fókuszt. A sorrendet a *tabIndex* tulajdonsággal befolyásolhatjuk – ezzel majd később foglalkozunk.

A felhasználó dolgát nagymértékben megkönnyítjük, ha a fókuszt állítását nem bízuk a böngészőre. Egy eseménykezelő eljárásból való kilépésnél célszerű a fókuszt ráállítani arra az objektumra, amellyel feltehetően folytatódik a bevitel. Ezt az objektumok *focus* metódusa segítségével tehetjük meg:

```
ObjektumAzonosító.focus()
```

A kezdeti beállításhoz a metódust a *window onload* eseménykezelőjében hívjuk meg.

A **4–15. példában** kiegészítettük az előző példa kódját a *focus* metódussal. Így a betöltésnél már a szövegmezőben villog a kurzor.

A **4–16. példában** kezdőértéket adtunk a szövegmezőnek, az *onload* eseménykezelőben pedig a parancsgombra állítottuk a fókuszt. Ezzel elértük, hogy a dokumentum betöltése után a felhasználó egérműveletek nélkül, az Enter billentyű lenyomásával tudja megjeleníteni a hónap napjainak a számát. Figyeljük meg, hogy a parancsgomb vastagabb kerettel jelent meg a weblapon!

## A select metódus

Az előző példákban hiába helyeztük a fókuszt a napok kiírása utána a szövegmezőre, ha a felhasználó egy másik hónapot akar begépelni, akkor először törölnie kell a tartalmát. Ezt a munkát a szöveg kijelölésével könnyíthetjük meg. Ha egy szöveget kijelölünk, akkor az új karakterek begépelésekor nem kell külön törölnünk, ez automatikusan megtörténik.

A szövegmező tartalmát a *select* metódussal jelölhetjük ki:

```
ObjektumAzonosító.select()
```

A **4–17. példában** a parancsgomb eseménykezelőjének végén meghívtuk a szövegmező *select* metódusát is. A betöltés után, üres szövegmező esetén természetesen nem történik kijelölés, csak a kurzor villog a szövegmezőben.

A továbbiakban az eseménykezelő eljárások végén a *focus* és a *select* metódusok alkalmazásával gondoskodunk a folytatásnak megfelelő objektum kijelöléséről.

Megjegyezzük, hogy a *select* metódusnak és a *Select Case* utasításnak a hasonló elnevezés ellenére semmi köze egymáshoz!

<sup>17</sup> A fókuszt csak a billentyűzettel kapcsolatos eseményekre vonatkozik.

### 4.3. Összetett feltételek

Eddigi példáinkban a szelekciós utasítások feltételei relációk voltak. Bonyolultabb esetekben az egyszerű feltételeket logikai műveletekkel kapcsolhatjuk össze. A logikai műveletek eredménye igaz vagy hamis lehet.

#### Logikai változók

Az igaz vagy hamis értékeket logikai változókban tároljuk. Logikai változókkal már találkoztunk bizonyos tulajdonságok megadásánál, például egy parancsgomb használatának (*disabled*) vagy az eseménybuborék terjedésének (*cancelBubble*) engedélyezésénél, letiltásánál. Értékük igaz vagy hamis lehet.

Logikai változó: értéke igaz (*True*) vagy hamis (*False*).

A szelekciós utasítások feltételeiben szereplő relációk eredménye hasonló értékeket vehet fel, tehát szintén logikai érték lesz:

5 < 3:        igaz  
-2 >= 7:      hamis

Az *If* ág akkor hajtódik végre, ha a szelekciós feltétel igaz. Hamis feltétel esetén az *Else* vagy *ElseIf* ág kerül sorra, amennyiben szerepel a kódban.

A feltételekbe írhatunk bonyolultabb kifejezéseket is:

```
If Egysegar < AnyagKoltseg + MunkaBer Then ...
```

Ezek a kifejezések a bennük szereplő változók értékétől függően szintén igaz vagy hamis logikai értéket állítanak elő.

A logikai változók elnevezésére hasonló szabályok vonatkoznak, mint a többi változóéra. Értékadó utasítással adhatunk nekik valamilyen értéket:

```
Dim Logikai_1, Logikai_2  
Logikai_1 = True  
Logikai_2 = 3 > 4
```

Fenti példákban a *Logikai\_1* változó értéke igaz, a *Logikai\_2* változóé pedig hamis lett. Bár ez utóbbinál a használt szintaxist megengedi a VBScript, célszerű a relációt zárójelbe tenni, mert áttekinthetőbb lesz a kód:

```
Logikai_2 = (3 > 4)
```

Megjegyezzük, hogy logikai változók aritmetikai kifejezésekben is állhatnak. Ekkor az interpreter a *True*-t -1-nek, a *False*-t pedig 0-nak tekinti. Például:

```
True + 10 = 9, 3 * False = 0
```

Ennek megfelelően a logikai értékek össze is hasonlíthatók egymással, illetve más numerikus értékekkel:

```
True < False, True < 0, False > -3
```

Ilyen kifejezésekkel azonban csak elvétve találkozunk.

## Logikai műveletek

A logikai változók között logikai műveleteket végezhetünk. Mivel a relációk eredménye egy-egy logikai érték, ezeket is összekapcsolhatjuk logikai műveletekkel. Bonyolultabb szelekcióknál gyakran előfordulnak ilyen logikai kifejezések.

A logikai műveletek megfelelnek az állítások összekapcsolásához használt „és”, „vagy” stb. kötőszavaknak. A programozási nyelvek a műveletek jelölésére a kötőszavak angol megfelelőjét használják: *And* (és), *Or* (vagy). A relációs kifejezéseket az áttekinthetőség érdekében célszerű zárójelbe tenni:

*Logikai művelet:*

*Kód:*

(X1 < 5) „és” (X2 = 6)

(X1 < 5) And (X2 = 6)

(Sugar < 10) „vagy” (Sugar > 30)

(Sugar < 10) Or (Sugar > 30)

Mind az *And*, mind az *Or* műveletnél felcserélhető az operandusok sorrendje:

A „és” B = B „és” A

A „vagy” B = B „vagy” A

Speciális, egyoperandusos művelet a „nem”, egy relációs kifejezés értékének a tagadása, melynek jele: *Not*.

A felsoroltakon túl még jó néhány logikai művelet létezik. A VBScript a *Not*, az *And* és az *Or* mellett az ekvivalenciát (*Eqv*), az implikációt (*Imp*) és a kizáró vagy (*Xor*) műveletet is ismeri. Ezeket azonban könyvünkben nem tárgyaljuk.

Megjegyezzük, hogy numerikus kifejezések között szintén végezhetünk logikai műveleteket. Ekkor az interpreter bitenként határozza meg az eredményt. A bitműveletekre hely hiányában nem térünk ki.

## A logikai műveletek igazságtáblázata

A logikai műveleteket az úgynevezett igazságtáblázattal adhatjuk meg, amelyben felsoroljuk a művelet által összekapcsolt kifejezések lehetséges értékeit és a logikai művelet eredményét.

A „nem” művelet eredménye az utána álló relációs kifejezés értékének a tagadása:

Kifejezés	A „nem” művelet eredménye
hamis	igaz
igaz	hamis

*A „nem” művelet igazságtáblázata*

A *Not* (5 > 6) logikai művelet eredménye például igaz, a *Not* (2 < 3) pedig hamis.

Az „és” műveletet a hétköznapi szóhasználatnak megfelelően csak akkor tekintjük igaznak, ha mindkét kifejezés igaz. Az (5 < 3) And (6 > 2) logikai művelet eredménye hamis, a (4 < 7) And (−1 < 9) eredménye pedig igaz.

1. kifejezés	2. kifejezés	Az „és” művelet eredménye	A „vagy” művelet eredménye
hamis	hamis	hamis	hamis
hamis	igaz	hamis	igaz
igaz	hamis	hamis	igaz
igaz	igaz	igaz	igaz

*A logikai műveletek igazságtáblázata*

A „vagy” művelet a definíciója szerint akkor lesz igaz, ha legalább az egyik kifejezés igaz. A  $(3 > 4) \text{ Or } (2 < 6)$  logikai művelet eredménye igaz, a  $(4 < 3) \text{ Or } (-5 > 6)$  pedig hamis.

Intervallumok megadásánál figyeljünk a megfelelő logikai művelet alkalmazására! Ha egy változó értéke 5 és 10 közé esik, akkor ezt az

$(5 < \text{Valtozo}) \text{ And } (\text{Valtozo} < 10)$

logikai művelettel fejezhetjük ki. Az

$(5 < \text{Valtozo}) \text{ Or } (\text{Valtozo} < 10)$

kifejezés eredménye mindig igaz! Ha egy változó értéke kisebb, mint 5 vagy nagyobb mint 10, akkor természetesen a „vagy” műveletet kell használnunk:

$(\text{Valtozo} < 5) \text{ Or } (10 < \text{Valtozo})$

Ha „és” műveletet használnánk helyette:

$(\text{Valtozo} < 5) \text{ And } (10 < \text{Valtozo})$

akkor mindig hamis kifejezést kapnánk!

### **A logikai műveletek precedenciája**

Egy logikai értéket előállító kifejezés eléggé bonyolult lehet. Szerepelhetnek benne aritmetikai kifejezések, relációs operátorok és logikai műveletek. Az eredmény szempontjából nagyon fontos tekintettel lennünk a műveletek végzésének sorrendjére. A logikai műveletek precedenciája (a teljesség kedvéért feltüntettük az általunk nem tárgyalt műveleteket is):

Not  
And  
Or  
Xor  
Eqv  
Imp

Először a listában feljebb álló műveletek hajtódnak végre. Az egyforma precedenciájú műveletek balról jobbra kerülnek kiértékelésre.



A precedencia-szabály alapján a következő kifejezésben:

`Not A Or B And C`

először a `Not A`, majd a `B And C`, végül az `Or` művelet kerül kiértékelésre. Ha a logikai változók értéke:

`A = False : B = True : C = False`

akkor a kifejezés értéke *True* lesz.

Ha meg akarjuk változtatni a kiértékelés sorrendjét, akkor az aritmetikai kifejezésekhez hasonlóan zárójeleket alkalmazunk. A változók fenti értékei mellett a

`(Not A Or B) And C`

kifejezés értéke már *False* lesz.

Ha egy kifejezésben vegyesen szerepelnek aritmetikai műveletek, relációs operátorok és logikai műveletek, akkor az interpreter először az aritmetikai műveleteket, majd a relációkat, végül a logikai műveleteket értékeli ki. Így a következő kifejezés:

`3 * A > 4 And 5 - B < 9`

egyenértékű a zárójelekkel ellátott:

`((3 * A) > 4) And ((5 - B) < 9)`

kifejezéssel.

A bonyolult kifejezések esetén akkor is célszerű zárójeleket használni, ha a precedencia-szabály alapján helyes eredményt kapnánk. A zárójelek megkönnyítik a műveletek áttekintését, bár kismértékben lassítják a feldolgozást, mert többletmunkát adnak az interpreternek.

## A relációk precedenciája

Mint láttuk, a relációk igaz vagy hamis logikai értéket eredményeznek, melyek szintén összehasonlíthatók egymással. Az összehasonlító operátorok esetén azonban nem létezik a sorrendet meghatározó precedencia, az összetett kifejezések mindig balról jobbra kerülnek kiértékelésre, amit zárójelekkel módosíthatunk. Így:

`-3 < 4 = 5` hamis, mert `-3 < 4` igaz (-1), de `-1 = 5` hamis

`-3 < (4 = 5)` értéke viszont igaz, mert `4 = 5` hamis (0), viszont `-3 < 0` igaz

Ilyen szokatlan kifejezések alkalmazását célszerű elkerülni, mert nehezen felderíthető hibákhoz vezethet.

## A logikai műveletek alkalmazása

Bár a logikai műveletek leggyakrabban szelekciós feltételekben szerepelnek, önálló alkalmazásukra is sor kerülhet. Írjunk programot, amely meghatározza egy évről, hogy szökőév-e.

Az 1582 óta használatos Gergely-naptár szerint a 4-gyel osztható évszámok számítanak szökőévnek. A 100-zal osztható évszámok közül azonban csak azok lesznek szökőévek, amelyek 400-zal is oszthatóak. Így például 1900 nem volt szökőév, de

2000 igen. Szökőév tehát az az év, amelynek évszáma 400-zal osztható, vagy 4-gyel osztható és 100-zal nem osztható. Ezt a következő logikai kifejezéssel határozhatjuk meg:

```
Szokoev = (Ev Mod 400 = 0) Or  
          ((Ev Mod 4 = 0) And (Ev Mod 100 <> 0))
```

Megjegyezzük, hogy a zárójelek itt nem befolyásolják a műveletek végzésének a sorrendjét, csak az áttekinthetőséget növelik.

Az így kapott logikai értéket használjuk fel az eredmény kiírásánál. Egy lehetséges megoldást a **4–18. példa** mutat be.

### Az Execute utasítás

A logikai műveletek alapos ismerete fontos feltétele a programozásnak. Gyakorlásukhoz ismerkedjünk meg a VBScript *Execute* utasításával. Ez az utasítás végrehajtja a paramétereként megadott sztringet. A sztringnek természetesen VBScript kódot kell tartalmaznia:

```
Execute("VBScript_utasítás")
```

A **4–19. példában** tetszőleges logikai kifejezést írhatunk a szövegmezőbe. A parancsgombra kattintás után megjelenik a kifejezés értéke. Futtassuk a programot a

```
True And False Or False And Not True
```

vagy a

```
(4 < 8) And (3 > 2) Or (5*3 + 8 < 11)
```

kifejezésekhez hasonló sztringek beírásával. Próbáljuk meghatározni a kifejezés értékét, mielőtt még rákattintanánk a parancsgombra!

Megjegyezzük, hogy az *Execute* utasítás nem csak a logikai kifejezések kiértékelésére alkalmas, hanem tetszőleges, sztringként összeállított VBScript utasítást végre tud hajtani.

### Logikai kifejezések a szelekciós feltételekben

A szelekciós utasítások feltétele általában egy logikai kifejezés.

Logikai kifejezés: logikai műveletekkel összekapcsolt relációk sorozata. Értéke igaz (*True*) vagy hamis (*False*).

Az *If* utasítás általános alakja tehát:

```
If Logikai_kifejezés Then  
    Utasítások_1  
Else  
    Utasítások_2  
End If
```

Ha a *Logikai\_kifejezés* igaz, akkor a *Then* ág, egyébként az *Else* ág hajtódik végre. Ugyanígy használjuk az *If ... ElseIf* utasítást.

Egyetlen logikai értéket szintén logikai kifejezésnek tekintünk, így a következő szelekciós utasításban:

```
If True Then
    Utasítások_1
Else
    Utasítások_2
End If
```

mindig az *Utasítások\_1* csoport hajtódik végre. A konstansként megadott *True* helyett egy logikai változó is szerepelhet:

```
Oszthato = (Szam Mod 7 = 0)
If Oszthato Then
    Utasítások_1
Else
    Utasítások_2
End If
```

A fenti szelekcióban a *Then* ág akkor érvényesül, ha a *Szam* osztható 7-tel, egyébként az *Else* ág utasításai kerülnek sorra. Az ilyen jellegű kódolás nagymértékben növeli a program áttekinthetőségét.

### Az *IsNumeric* függvény

A logikai változók segítségével ellenőrizhetjük, hogy a felhasználó numerikus értéket gépelt-e be egy szövegmezőbe. Az *IsNumeric* függvény által visszaadott logikai érték igaz, ha a paraméterként megadott kifejezés, például a szövegmező *value* tulajdonsága számmá alakítható, és hamis, ha egyéb karakterek is szerepelnek benne, vagy a felhasználó üresen hagyta (kitörölte).

A függvény felhasználásával hibaüzenetet jeleníthetünk meg, ha a karaktersorozat nem értelmezhető számként:

```
If IsNumeric(Szovegmezo.value) Then
    ' Számot gépelt be, következhet a feldolgozás.
Else
    ' Nem számot gépelt be, hibaüzenetet jelenítünk meg.
End If
```

Mivel a hibaüzenet küldése általában rövid kódot jelent, áttekinthetőbb az *If ... Then ... Else* szerkezet, ha a feltétel tagadásával megfordítjuk a két ág sorrendjét:

```
If Not IsNumeric(Szovegmezo.value) Then
    ' Nem számot gépelt be, hibaüzenetet adunk
Else
    ' Számot gépelt be, következhet a feldolgozás
End If
```

A 4–20. példában a gyökvonást végző 3–28. példa kódját alakítjuk át úgy, hogy negatív vagy nem numerikus érték begépelése esetén adjon hibaüzenetet. Mivel a szövegmező tartalmát többször is felhasználjuk, célszerű egy változóban tárolni:

```
Adat = AdatBe.value
```

A negatív előjel vizsgálata előtt először meg kell néznünk, hogy egyáltalán számot gépeltek-e be. Ha nem, akkor hibaüzenetet jelenítünk meg. Ez az ág hajtódik végre akkor is, ha a felhasználó kitörölte a kezdőértéket, de nem írt be mást, azaz a parancsgombra kattintáskor üres a szövegmező:

```
If Not IsNumeric(Adat) Then  
    window.alert("Számot gépeljen be!")
```

Negatív szám begépelése esetén is hibaüzenetet jelenítünk meg:

```
ElseIf Adat < 0 Then  
    window.alert("Negatív számból nem lehet gyököt vonni!")
```

Ha minden ellenőrzésen túljutottunk, akkor elvégezzük a gyökvonást és kiírjuk az eredményt:

```
Else  
    GyokKi.innerText = Sqr(Adat)  
End If
```

A CD-n lévő kódban még felhasználtunk néhány eddig említett fogást (a négyzetgyök törlését újabb adat beírása esetén, a fókusz visszaállítását a szövegmezőre stb.).

A további példákban a rövidebb és áttekinthetőbb kód érdekében általában nem alkalmazzuk a bevitel ilyen ellenőrzését, de javasoljuk az Olvasónak, hogy saját programjaiban sohase mulassza el!

### Az eljárás megszakítása

Az adatbevitel ellenőrzése következtében nagyon hosszadalmassá válhat a szelekciós szerkezet. Az egyes ágak a különböző előforduló hibákat ellenőrzik, és hibaüzenetet jelenítenek meg, ami nem tartozik hozzá a begépelte adatok feldolgozásának folyamatához. Ezt a szerkezetet az eljárás végrehajtásának megszakításával egyszerűsíthetjük.

A **4–21. példa** programja a bizonyítványban olvasható módon, szövegesen fogalmaz meg egy tantárgyi jegyet (jeles, jó, közepes, elégséges, elégtelen). Az értékadást egy *Select Case* utasítással hajtjuk végre:

```
Select Case Jegy  
    Case 1  
        Eredmeny = "elégtelen"  
    Case 2  
        Eredmeny = "elégséges"  
    Case 3  
        Eredmeny = "közepes"  
    Case 4  
        Eredmeny = "jó"  
    Case 5  
        Eredmeny = "jeles"  
End Select
```

Bár a *Case Else* ág segítségével is megjeleníthetnénk a hibaüzenetet, a felhasználót részletesebben szeretnénk tájékoztatni a jegy beírásakor elkövetett hibáról. Más hiba-

üzenetet írunk ki, ha üresen hagyta a szövegmezőt, és mást, ha rossz karaktert gépelt be. Az esetek szétválasztását egy *If ... ElseIf* utasítással érjük el:

```
If Jegy = "" Then
    window.alert("Írjon be egy jegyet!")
ElseIf Jegy < "1" Or Jegy > "5" Then
    window.alert("Rossz karaktert gépelt be!")
Else
    Select Case Jegy
    ...
    End Select
End If
```

Az *Else* ág tartalmazza a program lényeges részét, a már tárgyalt *Select Case*-t. Ha ez – mint általában – hosszadalmas kódot jelent, akkor a végén nehéz visszakeresni, hogy az *End If* vajon hová is tartozott.

#### *A szerkezet egyszerűsítése*

A bevitel ellenőrzését áttekinthetőbbé tehetjük, ha hiba esetén a hibaüzenet után az *Exit Sub* (kilépés az eljárásból) utasítás segítségével megszakítjuk az eseménykezelő eljárás végrehajtását. Ezzel a bonyolult, többágú szelekciót rövid, egyszerű feltételes elágazásokra bontjuk fel (**4–22. példa**):

```
If Jegy = "" Then
    window.alert("Írjon be egy jegyet!")
    Exit Sub
End If
If Jegy < "1" Or Jegy > "5" Then
    window.alert("Rossz karaktert gépelt be!")
    Exit Sub
End If
Select Case Jegy
...

```

Az ellenőrzések után következik az adatok feldolgozása, melyet így nem rejtünk el a többágú szelekció egyik ágában.

A megszakítás ellenére természetesen be kell írunk a kódba az *End If* illetve *End Sub* utasításokat még akkor is, ha mindig kiugrunk az eljárásból. A böngésző már a dokumentum betöltésekor szintaktikus hibát jelez, ha hiányzik valamelyik *End*.

Megoldásunk egyik hátránya, hogy minden egyes kilépés előtt meg kell hívnunk a *focus* és *select* metódusokat. Ezt a fenti kódban nem tettük meg, de a CD-n lévő példában igen.

A másik gondot a program áttekinthetősége jelenti. Egy szelekciós ágba eldugott *Exit* utasítást nem biztos, hogy észreveszünk a hibakeresés vagy a módosítás során. Egy bonyolult, hosszú program áttekintésénél nem merülünk el a részletekben, nem vizsgálunk meg minden ágot. Ezért célszerű az *Exit* utasításokat például feltűnően elhelyezett megjegyzésekkel kiemelni.

Nem csak a *Sub*, hanem a *Function*, a *Do* és a későbbiekben ismertetésre kerülő *For* utasítások blokkjában is állhat *Exit*, ezeket a szerkezeteket szintén meg lehet sza-

kítani. Az interpreter az *Exit* végrehajtásakor minden megkezdett részt, szelekciót, eljárást, függvényt lezár.

### A logikai kifejezések használata

A 4–23. példa programja további feldolgozás céljára élő személyek születési dátumát olvassa be. Az évet, a hónapot és a napot egy-egy szövegmezőbe kell beírni. A program hibaüzenetet ad, ha a dátum érvénytelen.

A születési év élő személyek esetén valószínűleg 1880 és 2004 közé esik, a hónap sorszáma pedig 1 és 12 közé. Az egyszerűség kedvéért fogadjunk el a hónaptól függetlenül 1 és 31 közé eső napokat. Feltételezzük, hogy a felhasználó csak egész számokat gépel be.

A beírható karaktersorozatok hosszát a szövegmezők *maxLength* tulajdonságának segítségével korlátozzuk.

Először értéket adunk a kiírandó üzenetnek, feltételezzük, hogy a begépelte érték hibás. Ezt a technikát már az előzőekben használtuk és indokoltuk:

```
Uzenet = "<B>A dátum hibás!</B>"
```

Ha a szövegmezők valamelyikébe nem numerikus értéket gépeltek be (vagy üresen hagyták), akkor a számokra vonatkozó relációk vizsgálatánál hibaüzenetet kapunk. Ezért a számértékek ellenőrzésére csak akkor kerülhet sor, ha minden begépelte sztring számként értelmezhető:

```
If IsNumeric(Ev) And IsNumeric(Ho) And IsNumeric(Nap) Then
```

A begépelte dátum érvényes, ha a következő feltétel teljesül:

```
If (Ev >= 1880) And (Ev <= 2004) And _  
    (Ho >= 1) And (Ho <= 12) And _  
    (Nap >= 1) And (Nap <= 31) Then  
    Uzenet = "A dátum érvényes."  
End If  
End If
```

A CD-n található program a parancsgombra kattintás után közli, hogy érvényes dátumot gépeltünk-e be. Mivel több szövegmező is szerepel a weblapon, az új dátum begépelésénél az előző kiírás törlését egy külön szubrutinnal oldottuk meg.

### Szelekciók és logikai kifejezések

A *Then* után álló *If* gyakran a program kusza logikai szerkezetére utal. Az egymásba ágyazott szelekciós utasítások és az összetett logikai kifejezések egyébként is szoros kapcsolatban vannak egymással. Érdemes meggondolni, hogy melyik megoldást alkalmazzuk a program szerkezetének egyszerűsítésére.

Egy pékmester például a következő programrészlettel határozza meg az üzletfeleinek adott kedvezmény százalékos értékét:

```
Kedvezmeny = 0
If KifliDarab > 200 Then
    If ZsomleDarab > 100 Then
        Kedvezmeny = 10
    End If
End If
```

A *Then* után álló *If* miatt érdemes megvizsgálni a szelekció működését. A kezdeti értékadást követő elágazásokat elemezve rájöhetünk arra, hogy akkor jár kedvezmény, ha 200-nál több kiflit és 100-nál több zsömlét vásárolunk. Ezt a feltételt egyetlen *If* utasítással is megfogalmazhatjuk:

```
Kedvezmeny = 0
If (KifliDarab > 200) And (ZsomleDarab > 100) Then
    Kedvezmeny = 10
End If
```

A szelekciók általában elbonyolítják a programok szerkezetét. Igyekezzünk mindig világosan és egyszerűen megfogalmazni a kódot. Ezzel sok bosszúságtól, hosszadalmas hibakereséstől kímélhetjük meg magunkat!

Összetett vizsgálatok esetén a fordított eset is előfordulhat, amikor célszerű a logikai kifejezést több részre bontani, és egymásba ágyazott *If* utasítások feltételeiben felhasználni. A dátum érvényességét vizsgáló példánkban ezt a következőképpen tehetjük meg (**4–24. példa**):

```
' Minden begépelt adat számként értelmezhető-e:
If IsNumeric(Ev) And IsNumeric(Ho) And IsNumeric(Nap) Then
    ' Az év ellenőrzése:
    If (Ev >= 1880) And (Ev <= 2004) Then
        ' A hónap ellenőrzése:
        If (Ho >= 1) And (Ho <= 12) Then
            ' A nap ellenőrzése:
            If (Nap >= 1) And (Nap <= 31) Then
                ' Minden feltétel teljesül, érvényes a dátum:
                Uzenet = "A dátum érvényes."
            End If
        End If
    End If
End If
```

Ez a megoldás némileg gyorsítja a feldolgozást, mert ha az egyik feltétel nem teljesül, akkor a következőket már nem kell megvizsgálni. A program szerkezetét azonban alaposan elbonyolítottuk.

A legjobban áttekinthető megoldást akkor kapjuk, ha a feltételeket egy-egy logikai változóban tároljuk, a szelekcióban pedig felhasználjuk ezeket a logikai változókat (**4–25. példa**):

```
If IsNumeric(Ev) And IsNumeric(Ho) And IsNumeric(Nap) Then
    EvJo = (Ev >= 1880) And (Ev <= 2004)
    HoJo = (Ho >= 1) And (Ho <= 12)
    NapJo = (Nap >= 1) And (Nap <= 31)
```

```
If EvJo And HoJo And NapJo Then
    Uzenet = "A dátum érvényes."
End If
End If
```

Bár itt is végzünk fölösleges tevékenységet (értékadást), amikor valamelyik változó értéke hamis, a legfontosabb alapelv mindig a program áttekinthetősége. Így a legkönnyebb ellenőrizni az algoritmust, felderíteni a hibákat, melyek esetenként csak jóval a program megírása és használatbavétele után derülnek ki.

### 4.4. Iterációk

A programozási feladatok megoldása során gyakran előfordul, hogy egy-egy utasítássorozatot többször is végre kell hajtani. Rossz megoldás lenne mindannyiszor megismételni a kódot, nem szólva arról az esetről, amikor csak menet közben derül ki a szükséges ismétlések száma. Az ismételt végrehajtáshoz iterációkat használunk. Az iterációt a programozásban gyakran ciklusnak hívják.

Iteráció (ciklus): egy utasításcsoport előírt számú, vagy egy logikai kifejezés értékétől függő ismétlése.

#### Léptető ciklusok

A programnyelvek többféle lehetőséget biztosítanak az iterációk létrehozásához. Ha a végrehajtás során már ismert az ismétlések száma, akkor előírt menetszámú ciklust készítünk. Ezt gyakran léptető vagy számláló ciklusnak nevezik.

A léptető ciklusnál egy úgynevezett ciklusváltozót deklarálunk, melynek értéke a megadott kezdőértéktől az előírt végértékig meghatározott lépésközzel változik. A lépésköz, a kezdő- és végérték tetszőleges aritmetikai kifejezés lehet.

A ciklust először a ciklusváltozó kezdőértékével hajtjuk végre. A végrehajtás után a ciklusváltozó értékét a lépésközzel megváltoztatjuk, majd ellenőrizzük, hogy túllépte-e a végértéket. Ha nem, akkor ismét végrehajtjuk az utasítássorozatot. Megjegyezzük, hogy az ellenőrzést a VBScript a ciklus első végrehajtása előtt végzi el, tehát lehet, hogy a ciklus egyszer sem fut le. A léptető ciklus pszeudokódja:

```
Ciklus Ciklusváltozó = Kezdőérték-től Végérték-ig Lépésköz-zel
    Utasítások
Ciklus vége
```

A ciklusváltozó kezdő- és végértékének, illetve a lépésköznek a megadását ciklusfejnek, az utasításokat pedig a ciklus magjának nevezzük. (A ciklusváltozót gyakran I, J, K-val jelölik, eltekintve a beszédes változónevek használatától.)

A következő pszeudokód ciklusa például 1-től 10-ig kiírja a természetes számokat és négyzeteiket:

```
Ciklus I = 1-től 10-ig, 1-esével
    Kiírás: I, I2
Ciklus vége
```

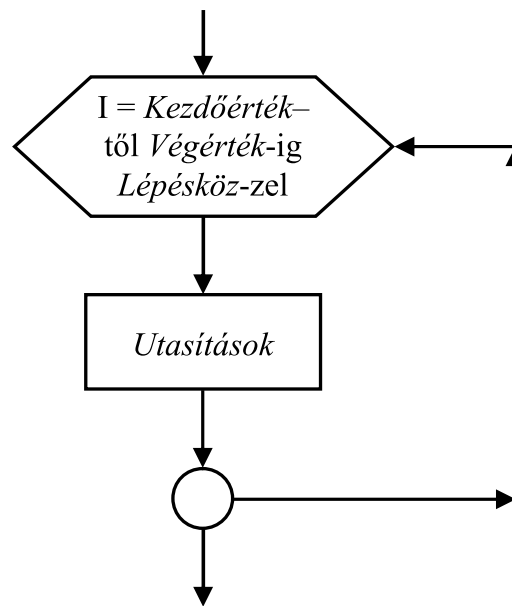


A kiírás először  $I = 1$ -gyel, majd 2-vel stb. történik, egészen  $I = 10$ -ig.

A ciklusváltozónak nem kell felvennie pontosan a végértéket. Ha a növelés után már nagyobb lesz nála, akkor nem ismétlődik meg a ciklusmag végrehajtása, kilépünk a ciklusból. A következő ciklusfej hatására például a ciklusváltozó értéke sorra 10, 13 és 16 lesz:

Ciklus  $I = 10$ -tól 17-ig, 3-asával

A léptető ciklus folyamatábráján gyakran egy elnyújtott hatszöggel jelölik a ciklusfejet, és egy körrel vagy ellipszissel a ciklus végét:



4–3. ábra. A léptető ciklus folyamatábrája

### A For ... Next utasítás

A VBScriptben a léptető ciklust a *For ... Next* utasítással adjuk meg, melynek szintaxisa:

```
For Ciklusváltozó = Kezdőérték To Végérték Step Lépésköz
    Utasítások
Next
```

A ciklus szerkezete pontosan megfelel a pszeudokódnak, csak angol szavakat használunk. A *Next* után gyakran feltüntetik a ciklusváltozót, ami főleg az egymásba ágyazott ciklusok esetén segít összepárosítani a *For* és *Next* utasításokat:

```
For I = ...
    ...
Next I
```

A **4–26. példában** megvalósítjuk az előző pontban említett ciklust, 1-től 10-ig kiírjuk a számok négyzetét. Ne felejtsünk el sort emelni a `<BR>` objektum segítségével:

```
For I = 1 To 10 Step 1
    document.write(I & "&sup2; = " & I^2 & "<BR>")
Next
```

A VBScriptben (és sok más programnyelvben) elhagyható a *Step*, ha a lépésköz értéke 1, ezért az előző ciklusfej így is felírható:

```
For I = 1 To 10
```

Bár a rövidítés csökkenti a kód áttekinthetőségét, alkalmazni fogjuk, mert a programozásban általánosan bevett szokás.

A ciklusokat bemutató példa sokkal szebb eredményre vezet a *Formaz* függvény segítségével. A megjelenítés egyszerűbb, ha a HTML-kódot nem a szkript készíti el. Hogy hivatkozassunk az objektumokra, az iterációt a *window onload* eseménykezelőjébe tettük. A kiíráshoz a táblázatot először a *Tablázat* sztringben állítottuk össze, és felhasználtuk a *TablázatKi* PRE-objektumot. A forráskódot a **4–27. példában** találjuk. (A *Formaz* függvény használatát a 3–40. példa, a PRE-objektumot pedig a 2–19. példa mutatta be.)

Megjegyezzük, hogy a programnyelvek többségében a ciklus befejezése után határozatlan a ciklusváltozó értéke, tehát nem tanácsos felhasználni. Ha szükségünk van rá, akkor a cikluson belül egy segédváltozónak adjuk át az értékét. A segédváltozó megmarad a ciklusból való kilépés után is.

### Folyamatjelző az állapotsorban

Az Internet Explorer csak akkor végzi el a weblap ablakának frissítését, a szövegek kiírását, amikor már kiléptünk az eseménykezelő eljárásból. (Kivételt csak a párbeszédablakok megjelenítése képez – ekkor frissíti a megjelenítést). Ezért felesleges a cikluson belül megadni az objektumok *innerText* vagy *innerHTML* tulajdonságát, úgylis csak a végén jelenik meg az értékük.

Hosszú ciklusok esetén azonban meglehetősen zavarja a felhasználót, ha nem történik semmi változás, mert nem tudja, hogy működik-e a program, vagy valamilyen hiba miatt „lefagyott”, és hiába vár.

A **4–28. példában** átírjuk az előző példa ciklusváltozójának végértékét 1000-re, és beletesszük a megjelenítést végző utasítást a ciklusba:

```
For I = 1 to 1000
    Tablázat = Tablázat & I & "&sup2; = " & I^2 & "<BR>"
    TablázatKi.innerHTML = Tablázat
Next
```

A ciklust most egy parancsgomb eseménykezelő eljárása hajtja végre.

A program a parancsgombra való kattintás után jó néhány másodpercig mindenféle visszajelzés nélkül dolgozik, a táblázat csak a végén jelenik meg. A felhasználók általában nem várnak néhány másodpercnél tovább, megpróbálják bezárni az ablakot. A Bezárás gombbal ez nem fog sikerülni. Ha a tálcán a jobb egérgomb segítségével pró-

bálkozunk vele, akkor – az itt teljesen félrevezető – „A program nem válaszol” üzenetet kapjuk. Ez igaz, de nem azért, mert lefagyott, hanem azért, mert még nem készült el a táblázat.

A felhasználók megnyugtatóására folyamatjelzőket szoktak használni, általában hosszú, keskeny téglalapokat, melyekben színes négyzetek jelennek meg egymás mellett. Mire a sor végére érnek, befejeződik a feladat végrehajtása. Ilyen folyamatjelzők elhelyezéséhez további ismeretekre van szükség, addig is a dokumentum ablakának állapotosorát használjuk erre a célra. A **4–29. példában** a ciklusba illesztjük a következő utasítást:

```
window.status = "I = " & I
```

Nem árt, ha a felhasználót is tájékoztatjuk arról, hogy figyelje az állapotosort.

A folyamatjelző ugyan némileg lassítja a program futását, de hosszabb eljárások, ciklusok esetén mindenképpen javasoljuk az alkalmazását.

### A ciklus futásának gyorsítása

A weblap megjelenítésének folyamatos módosítása, az *innerText* vagy *innerHTML* tulajdonságok állandó felülírása nagyon leterheli a böngészőt, és lelassítja a program futását

A **4–30. példában** visszatettük a megjelenítést végző

```
TablázatKi.innerHTML = Tablázat
```

utasítást a *Next* után. Jelentős mértékben felgyorsult a program végrehajtása! A továbbiakban az objektumok szövegét mindig egy külön sztringben állítjuk össze, és az eseménykezelő eljárás végén adjuk át az *innerText* vagy *innerHTML* tulajdonságnak.

Ha próbálgatni akarjuk a ciklusváltozó felső határának emelését, akkor hagyjuk ki a megjelenítést, mert a túl hosszú HTML-kódtól valóban lefagyhat a böngésző. A **4–31. példa** segítségével egy üres ciklust futtathatunk, ami csak az állapotosor tartalmát változtatja. Az ismétlések számát a szövegmezőben adhatjuk meg.

### Kifejezések a ciklusfejben

A ciklusfejben szereplő mennyiségek tetszőleges aritmetikai kifejezések lehetnek. A **4–32. példa** ciklusa a felhasználó által a *ParameterBe* azonosítójú szövegmezőbe gépelt értéktől kezdve írja ki a számok négyzetgyökét. Végértéknek a paraméter tízszeresét, lépésköznél a kezdőérték kétszeresének veszi:

```
Parameter = ParameterBe.value
Tablázat = ""
For I = Parameter To 10 * Parameter Step 2 * Parameter
    Tablázat = Tablázat & "&radic;" & I & " = " & Sqr(I) & "<BR>"
Next
TablázatKi.innerHTML = Tablázat
```

Futtassuk le a programot úgy, hogy 1-et adunk meg kezdőértéknek. Ekkor a lépésköz 2, a végérték 10. Miután a ciklusváltozó értéke eljut a 9-ig, a ciklus lefutása után 2-vel megnő. Ekkor már túllépi a végértéket, így a ciklusmag nem hajtodik többé végre.

### Végértékek és lépésközök

A VBScript a ciklus kezdetén hasonlítja össze a ciklusváltozó értékét a végértékkel. Ha a kezdőérték már eleve nagyobb, akkor a ciklus egyszer sem fut le. A következő ciklusfej ilyen esetet mutat:

```
For I = 10 To 8
```

Egyes programnyelvek a vizsgálatot a *For* ciklus lefutása után végzik. Ezek a ciklusok a kezdő és végértéktől függetlenül egyszer biztosan lefutnak. Erre a sorrendre a program írásánál ügyelnünk kell.

A lépésköz értéke negatív is lehet. Ekkor a ciklusváltozó értéke minden egyes ismétlésnél csökken. A **4–33. példa** 10-től 1-ig írja ki a számok négyzetét:

```
For I = 10 to 1 Step -1
    IFormaz = Formaz(I, 3, 0)
    NegyzetFormaz = Formaz(I^2, 4, 0)
    Tablazat = Tablazat & IFormaz & "<SUP>2</SUP> = " & _
        NegyzetFormaz & "<BR>"
Next
```

Negatív lépésköz esetén természetesen megváltozik a ciklus végrehajtásának a feltétele. Ha a ciklusváltozó értéke kisebb, mint a végérték, akkor befejeződik az ismétlés. A következő ciklus így egyszer sem fut le:

```
For I = 10 To 12 Step -1
```

Ha a ciklusfejben megadott paraméterek között törtszámok is előfordulnak, akkor figyelemmel kell lennünk a számítási pontosságra. A számítógépek kettes számrendszerben kódolják a számokat. Gyakran előfordul, hogy a tízes számrendszerben megadott egyszerű tört (mondjuk a 0,1) binárisan végtelen szakaszos tört lesz, így ennek csak közelítő értéke használható. A **4–34. példa** ciklusa a ciklusfej ellenére csak 1,9-ig jut el, a 2 négyzetét már nem írja ki:

```
For I = 1 to 2 Step 0.1
    Tablazat = Tablazat & I & "&sup2; = " & I^2 & "<BR>"
Next
```

Ennek oka, hogy a 0,1-et binárisan csak kerekítve tudjuk tárolni. A közelítő értékkel végzett számítások miatt az utolsó lépésben 1,9-hez hozzáadva az összeg már kicsivel túllépi a végértéket, így a ciklus befejezi működését.

Az ilyen bizonytalan kimenetek miatt a ciklusfejben csak egész számokat ajánlott használni! A cikluson belül a segítségükkel határozzuk meg a szükséges értékeket. A fenti ciklus helyes megoldását a **4–35. példa** mutatja be:

```
For I = 10 to 20 Step 1
    Szam = I/10
    Tablazat = Tablazat & Szam & "&sup2; = " & Szam^2 & "<BR>"
Next
```

A *Szam* segédváltozó bevezetésével elértük, hogy a táblázat az *I* ciklusváltozó tízedrészére, tehát 1-től 2-ig készül el, így a ciklus a kívánt értékekre fut le.

A törtszámok használatára a kerekítés miatt a programozás más területein is ügyelnünk kell. Erre később még visszatérünk.

### A ciklusfej értékeinek módosítása a ciklusban

Elvileg nincs akadálya annak, hogy egy cikluson belül megváltoztassuk a ciklusváltozó értékét. A **4–36. példa** minden egyes ismétlésnél rákérdez a ciklusváltozóra, így lehetőséget ad a módosításra:

```
For I = 1 to 4
    I = InputBox("Jelenlegi érték = " & I & "
                "Új érték:", "Ciklusváltozó", I)
Next
```

(A CD-n szereplő kód néhány üres sort is hozzáfűz az *InputBox* üzenetéhez.)

Először fogadjuk el a felajánlott értékeket (a parancsgombra való kattintás helyett használhatjuk az Enter billentyűt is). Futtassuk le újra a programot úgy, hogy  $I = 2$  esetén írjunk be 6-ot. Mivel ez nagyobb, mint a végérték, a ciklus azonnal befejeződik. Ha menet közben például egy negatív értéket írunk be a ciklusváltozóra, akkor onnan folytatódik a léptetés, megnő az ismétlések száma.

A ciklusváltozó módosítása a cikluson belül nehezen felderíthető hibákhoz vezethet. Csak kivételesen indokolt esetben használjuk ezt a programozási fogást!

Mint láttuk, a ciklusváltozó végértéke és a lépésköz tetszőleges aritmetikai kifejezés lehet. Az interpreter a ciklus kezdetén feljegyzi az értéküket. Így a cikluson belül már hiába változtatjuk meg ezeket az adatokat, a módosítás a ciklus ismétlődéseire nincs hatással.

A **4–37. példában** a végértéket és a lépésközt is egy-egy változóval adjuk meg. A változókat a cikluson belül módosítjuk, majd kiíratjuk a ciklusváltozó által felvett értékeket:

```
VegErtek = 5
Lepeskoz = 2
For I = 1 to VegErtek Step Lepeskoz
    VegErtek = 20
    Lepeskoz = 5
    Lista = Lista & I & "; "
Next
```

Mint a példa futtatásából látható, a ciklus az eredeti értékekkel került végrehajtásra. A cikluson belüli módosításoknak nem volt hatása sem a végértékre, sem a lépésközre.

### Végtelen ciklusok

A ciklusváltozó megváltoztatása egy másik kellemetlen következménnyel is járhat. A **4–38. példa** ciklusa sohasem áll meg, hiszen mindig visszaírjuk a ciklusváltozó eredeti értékét. Így végtelen hurokba kerülünk:

```
For I = 1 to 4
    I = 1
    window.alert("I = " & I)
Next
```

Ha az Olvasó elindította ezt az „alattomos” ciklust, akkor csak a Windows feladatkezelője segítségével tud kikeveredni belőle. A Windows 98 használata esetén egyszerre le kell nyomni a Ctrl, Alt és Del billentyűket, majd a megjelenő listában rá kell kattintani a Példa4-38.htm fájlra (valószínűleg legfelül helyezkedik el). Ezután kattintunk rá a „Feladat bezárása” gombra, aztán a megjelenő párbeszédablakban ismét a „Feladat bezárása” gombra. Windows 2000 és Windows XP esetén az eljárás hasonló, az üzenetek kissé különböznek.

Szerencsére az Internet Explorer figyeli a szkripteket, és ha valamelyik túl sokáig fut, akkor lehetővé teszi a leállítást. Az előző példából töröljük ki a *window.alert* hívását, majd a parancsgombbal indítsuk el a ciklust (**4–39. példa**):

```
For I = 1 to 4
  I = 1
Next
```

Az indítás után néhány másodperccel az Internet Explorer figyelmeztet a lehetséges hibára és módot ad a szkript futásának megszakítására. A böngésző nem a program logikáját elemezte, hanem az erőforrások lefoglalását figyelte. Így egy hosszú, bár jól működő ciklus esetén is üzenetet küld. Ezt a viselkedést a **4–40. példa** futtatásánál tapasztalhatjuk:

```
For I = 1 to 10000000
  ' Nem csinálunk semmit a ciklusban
Next
window.alert("Ciklus vége.")
```

Az indítás után néhány másodperccel most is megkapjuk figyelmeztetést. Ha megszakítjuk a parancsfájl (szkript) futását, akkor nem jelenik meg a „Ciklus vége” üzenet, mert a böngésző abbahagyta a szkript végrehajtását.

### Egymásba ágyazott ciklusok

A ciklusokat a szelekciókhoz hasonlóan egymásba ágyazhatjuk. Ügyeljünk arra, hogy az egyes ciklusoknál különböző ciklusváltozókat alkalmazzunk!

A **4–41. példa** két ciklusa szorzótáblát készít 1x1-től 10x10-ig. A belső ciklus minden egyes I érték esetén tízszer lefut, eközben a J értéke 1-től 10-ig változik:

```
For I = 1 To 10
  For J = 1 To 10
    document.write(I * J & " ")
  Next
  document.write("<BR>")
Next
```

Az egymásba ágyazott ciklusoknál az első *Next* az utolsó *For*-hoz tartozik, az utolsó *Next* zárja le az elsőként megkezdett ciklust.

Figyeljük meg, hogy soremelést (<BR>) csak a külső ciklusban írtunk a dokumentumba! Így a belső ciklus által kiírt értékek egy sorba kerülnek.

A CD-mellékleten található **4–42. példában** a táblázatot először a *Tablázat* sztringben állítottuk össze, majd a ciklus lefutása után írtuk ki. Az esztétikus megje-

lenés kialakításához felhasználtuk a *Formaz* függvényt. Ehhez csatolni kell a *Formaz.vbs* fájlt a dokumentumhoz. A kiírást egy PRE-objektum segítségével végezzük el. Az utasításokat a *window onload* eseménykezelőjébe tettük, hogy hivatkozhasunk a dokumentum DHTML-objektumaira.

A **4–43. példában** három egymásba ágyazott ciklus segítségével változtatjuk a weblap háttérszínét. Emlékezzünk vissza arra, hogy az egyes színösszetevők értéke 0 és 255 közé eshet. A szükséges hexadecimális kódot az RGB függvény állítja elő. A lépésközt megnöveltük, hogy gyorsabb legyen a változás. Az állapotsorban megjelenítettük a ciklusváltozók értékét. Amíg gyönyörködünk a képernyő színeiben, gondoljuk át, amit eddig tanultunk a vezérlőszerkezetekről!

### Feltételvezérelt ciklusok

A léptető ciklusok egy ciklusváltozó segítségével határozzák meg a szükséges ismétlések számát. Bár a kezdő és végérték tetszőleges aritmetikai kifejezés lehet, a ciklus végrehajtásakor már pontosan ismert az értékük. Előfordulhatnak azonban olyan esetek, amikor valamilyen logikai kifejezés igaz vagy hamis értékéhez kapcsoljuk a ciklusmag végrehajtását.

Feltételvezérelt ciklus: egy logikai kifejezés igaz vagy hamis értéke eredményezi a ciklus utasításainak végrehajtását.

Addig üsd a vasat, amíg meleg! – tartja a közmondás. Programozási szempontból feltételvezérelt ciklust kaptunk. Nem tudjuk megmondani, meddig fogjuk ütni. Az ismétlések száma a feltétel teljesülésétől függ. Az ismétlés feltétele, hogy meleg legyen a vas. Ha teljesül, végrehajtjuk a ciklusmagot, azaz ütjük. Ha már kihűlt, akkor befejezzük a munkát.

Addig jár a korsó a kútra, amíg el nem törik. Ebben a ciklusban egy feltétel tagadását találjuk. Ha eltörik a korsó, akkor nem jár többé a kútra. A kilépés feltétele, hogy eltörik a korsó. Amíg ez nem teljesül (azaz hamis), addig ismétlődik a ciklusmag.

Ismétlési feltétel: igaz értéke a ciklus végrehajtását okozza.  
Kilépési feltétel: igaz értéke befejezi a ciklust  
(ismétlésre a hamis érték esetén kerül sor).

A feltételvezérelt ciklusok pszeudokódjában az „amíg” és a „mígnem” kötőszavakkal különböztetjük meg a kétféle esetet. Az „amíg” használatakor a logikai kifejezés igaz értéke, a „mígnem” esetén pedig hamis értéke eredményezi a ciklusutasítások ismételt végrehajtását:

```
Ciklus amíg Ismétlési_Feltétel
' Ismétlés, amíg a
' feltétel igaz
  Utasítások
Ciklus vége
```

```
Ciklus mígnem Kilépési_Feltétel
' Ismétlés, amíg a
' feltétel hamis
  Utasítások
Ciklus vége
```

Vegyük észre, hogy a kétféle eset egyenértékű, a feltétel tagadásával az egyik helyett a másikat használhatjuk. Csupán kényelmi okokból áll rendelkezésünkre mind a kettő.

### Elöl- és hátultesztelő ciklusok

A feltételvezérelt ciklusokat más szempontból is csoportosíthatjuk. Az ismétléshez vezető feltételt ugyanis ki lehet értékelni a ciklus elején, és a ciklus végén. Ez utóbbi esetben a ciklus egyszer mindenképpen lefut.

Elöltesztelő ciklus: a feltételt a ciklus elején értékeli ki az interpreter.  
Hátultesztelő ciklus: a feltételt a ciklus végén értékeli ki az interpreter  
(a ciklus egyszer mindenképpen lefut).

A hátultesztelő ciklusok pszeudokódjában az „amíg” és „mígnem” szavakat a „Ciklus vége” jelzés elé tesszük:

```
Ciklus
' Ismétlés, amíg a
' feltétel igaz
  Utasítások
amíg Ismétlési_Feltétel
Ciklus vége
```

```
Ciklus
' Ismétlés, amíg a
' feltétel hamis
  Utasítások
mígnem Kilépési_Feltétel
Ciklus vége
```

Az előltesztelő ciklusok esetén előfordulhat, hogy a ciklusmag utasításai egyszer sem hajtódnak végre. A hátultesztelő ciklusok a feltétel értékétől függetlenül egyszer biztosan lefutnak, hiszen a kiértékelésre csak a ciklus végén kerül sor. A *For ... Next* ciklus ebből a szempontból az előltesztelő kategóriába tartozik.

A program tervezése során alaposan meg kell gondolnunk, hogy elől-, illetve hátultesztelő ciklust használjunk-e, és ismétlési vagy kilépési feltételt adjunk-e meg. Cél a minél egyszerűbb, világosabb kód létrehozása. Ha például az ismétlés feltételében tagadás szerepel, akkor használjunk helyette kilépési feltételt! Programozási szempontból a korszóra vonatkozó közmondást inkább így mondanánk: addig jár a korszó a kútra, mígnem eltörik. Ezzel megtakarítjuk a tagadást, a *Not* logikai művelet alkalmazását.

### A Do ... Loop utasítás

A VBScript nyelvben a *Do ... Loop* utasítással hozható létre feltételvezérelt ciklus. A feltétel igaz értékénél történő ismétléshez a *While* (amíg), a hamis értéknél történő ismétléshez az *Until* (mígnem) kulcsszót használjuk. Elöltesztelő ciklusoknál a *Do*, hátultesztelő esetben pedig a *Loop* után írjuk a kulcsszót és a feltételt.

*Elöltesztelő ciklus,*  
*ismétlés, amíg a feltétel igaz:*

```
Do While Ismétlési_Feltétel
  Utasítások
Loop
```

*Hátultesztelő ciklus,*  
*ismétlés, amíg a feltétel igaz:*

```
Do
  Utasítások
Loop While Ismétlési_Feltétel
```



*Elöltesztelő ciklus,  
ismétlés, amíg a feltétel hamis:*

```
Do Until Kilépési_Feltétel
    Utasítások
Loop
```

*Hátultesztelő ciklus,  
ismétlés, amíg a feltétel hamis:*

```
Do
    Utasítások
Loop Until Kilépési_Feltétel
```

A *Do ... Loop* utasítással tehát mind a négyféle feltételvezérelt ciklus létrehozható. Az ismétlési és a kilépési feltétel – a szelekciós utasításokhoz hasonlóan – itt is egy-egy logikai kifejezés.

A **4–44. példában** az *InputBox* segítségével bekérünk a felhasználótól egy 100-nál nagyobb számot. A bekérést mindaddig ismételjük, amíg a beírt szám nem felel meg a feltételnek.

Ennél a példánál hátultesztelő ciklust kell alkalmaznunk, hiszen amíg a felhasználó nem ír be számot, addig nem tudjuk ellenőrizni, hogy nagyobb-e 100-nál:

```
Do
    Szam = InputBox("Írjon be egy 100-nál nagyobb számot!", _
                    "Szám bekérése", 0)
Loop Until Szam > 100
```

A ciklus egyszer mindenképpen lefut, bekéri a számot – az *Until* ellenőrzi, hogy a feltétel teljesül-e. Ha igen, akkor a szám nagyobb, mint 100. Készen vagyunk a beolvasással, ismétlésre nem kerül sor. Ha kisebb vagy egyenlő, mint 100, akkor a feltétel nem teljesül. Megismételjük a ciklust, újabb számot kérünk be. Figyeljük meg, hogy a ciklus elején nem is lehetett volna a feltételt ellenőrizni, mert még nem állt rendelkezésünkre a szám értéke!

A **4–45. példában** ugyanezt a hatást érjük el, amikor az ismétlés feltételét adjuk meg a *While* kulcsszóval:

```
Do
    Szam = InputBox("Írjon be egy 100-nál nagyobb számot!", _
                    "Szám bekérése", 0)
Loop While Szam <= 100
```

Figyeljünk arra, hogy a nagyobb ellentettje a kisebb vagy egyenlő!

## Elöltesztelő Do ... Loop ciklus

Ha valamennyi pénzt beteszünk a bankba, akkor az adott kamatláb mellett hány év múlva érjük el az általunk kívánt összeget? Bár ez a kamatos kamatszámítás a matematika eszközeivel is megválaszolható, használjuk ki a számítógép gyorsaságát, és próbálkozással adjuk meg a feleletet.

A **4–46. példában** az alaphoz (a betett pénzhez) addig adjuk hozzá az éves kamatot, amíg túl nem lépi a kívánt összeget. Közben számoljuk az éveket, így a végén megkapjuk a választ a kérdésre.

A ciklus előkészítésénél a szövegmezőkbe gépelt alaptőke, százalékos kamat és kívánt összeg értékét átadjuk a *Toke*, *Kamat* illetve *Osszeg* változóknak. Így gyorsítjuk a

program futását és egyszerűsítjük a kódot. Az összehasonlítások miatt a szövegmezők *value* tulajdonságát numerikus értéké kell alakítani.

Az évek számolásához deklaráljuk az *Ev* változót, melynek értékét inicializáljuk:

```
Ev = 0
```

Elöltesztelő ciklust készítünk, hiszen lehetséges, hogy a felhasználó által megadott alaptőke eleve eléri a kívánt összeget. Ekkor nem szabad lefutnia a ciklusnak, mert hibás eredményt adna. Egy évig javasolná a bankban tartani a pénzünket, amire pedig nem is lenne szükség.

A cikluson belül kiszámítjuk az újabb tőkét, és eggyel megnöveljük az évek számát. Az ismétlés addig folyik, amíg a tőke el nem éri a megadott összeget (ismétlési feltétel):

```
Do While Toke < Osszeg
    Toke = Toke * (1 + Kamat / 100)
    Ev = Ev + 1
Loop
```

Ezek után már csak meg kell jelenítenünk az eredményt.

Megoldásunk szépséghibája, hogy az  $(1 + \text{Kamat} / 100)$  kifejezés értékét minden egyes ismétlés során kiszámítja a program. Ezt célszerű a ciklus elé tenni, a cikluson belül pedig az így meghatározott értéket felhasználni. A későbbiekben is ügyeljünk arra, hogy ugyanazt a kifejezést ne számoljuk ki minden egyes ismétlésnél, mert alaposan megnőhet a futási idő!

### Kilépési és ismétlési feltételek

A ciklusok kilépési, illetve ismétlési feltételei tetszőleges logikai kifejezések lehetnek. Feltételként alkalmazhatunk akár egyetlen logikai változót is. Így jól áttekinthető szerkezeteket hozhatunk létre.

Kérjünk be a felhasználótól egy 100 és 200 közé eső, 7-tel osztható számot! Ismétljük meg a bekérést mindaddig, amíg a beírt szám meg nem felel a feltételnek.

Egy szám akkor osztható 7-tel, ha az osztási maradéka 0, így a kilépési feltételt megfogalmazó logikai kifejezés (zárójeleket csak az áttekinthetőség kedvéért alkalmaztunk):

```
(Szam Mod 7 = 0) And (Szam >= 100) And (Szam <= 200)
```

Mivel a vizsgálat előtt be kell kérnünk a számot, hátultesztelő ciklust alkalmazunk:

```
Do
    Szam = InputBox ("Üzenet", "Cím", 0)
Loop Until (Szam Mod 7 = 0) And (Szam >= 100) And (Szam <= 200)
```

Az *Until* vagy a *While* után írt bonyolult logikai kifejezés megnehezíti a kód áttekinthetőségét. A **4–47. példában** egy beszédes nevű logikai változót alkalmazunk helyette, és ezt írjuk az ismétlési feltétel helyére:

```

Do
    Szam = InputBox (Üzenet, Cím, 0)
    JoSzam = (Szam Mod 7 = 0) And (Szam >= 100) And (Szam <= 200)
Loop Until JoSzam

```

A szám beolvasása után az interpreter megvizsgálja a feltételeket. Az „és” műveletek eredménye csak akkor igaz, ha mindegyik feltétel teljesül. Ekkor a *JoSzam* változó értéke *True* (igaz) lesz, ki lehet lépni a ciklusból.

Ismételten felhívjuk a figyelmet az összetett logikai feltételek átgondolására. Ha a fenti kifejezésben az „és” műveletek helyett „vagy” műveleteket használtunk volna:

```
(Szam Mod 7 = 0) Or (Szam >= 100) Or (Szam <= 200)
```

akkor a kifejezés mindig igaz eredményt adna, hiszen bármely számra teljesül, hogy nagyobb, mint 100 vagy kisebb, mint 200. (A 7-tel való oszthatóság ebben az esetben már nem is számít.) Így egyből kilépnénk a ciklusból, a nem megfelelő számokat is elfogadnánk.

Ha rosszul fogalmazzuk meg a relációt, például:

```
(Szam < 100) And (Szam > 200)
```

alakba íránk, akkor az eredmény mindig hamis lenne, hiszen nincs olyan szám, amelyik kisebb, mint 100 és nagyobb, mint 200. Így végtelen ciklushoz jutnánk.

## Példa a ciklusok alkalmazására

Összefoglalásként a **4–48. példa** olyan programot mutat be, amely előállítja egy megadott egész szám prímtényezői felbontását, és azt a szokásos alakban jeleníti meg. Például:

$$70560 = 2^5 \cdot 3^2 \cdot 5 \cdot 7^2$$

Ennél a feladatnál már alaposan át kell gondolni az algoritmust. Egy szám osztóit úgy kereshetjük meg, ha a számot elosztjuk 2-től kezdve a természetes számokkal:

```

Ciklus 2-től kezdve az N természetes számokra
    az osztók keresése
Ciklus vége

```

Ha találunk egy osztót, akkor egy beágyazott ciklusban addig osztjuk vele a számot, amíg az így kapott hányadosok tovább oszthatók, és feljegyezzük, hány osztást végeztünk. Ez lesz a talált osztó kitevője a prímtényezői felbontásban:

```

' belső ciklus:
Kitevő = 0
Ciklus amíg a szám osztható az N természetes számmal
    Szám = Szám / N
    Kitevő = Kitevő + 1
Ciklus vége
a talált prímszám megjelenítése (N^Kitevő)

```

Mivel a belső ciklusban az osztásokkal folyamatosan csökkentjük a számot, a külső ciklust addig kell futtatni, amíg a szám értéke nagyobb, mint 1. Ezért nem használhatunk *For ... Next* utasítást, tehát a ciklusváltozót mi növeljük:

```
N = 2
Ciklus amíg Szam > 1
    ' belső ciklus
    N = N + 1
Ciklus vége
```

Már csak a kiírásra kerülő *Eredmeny* karakterláncot kell összeállítanunk. A kezdeti értékadást követően ezt a belső ciklus után egy többágú szelekcióval tehetjük meg. Ha a kitevő 0 maradt, akkor ez az N természetes szám nem osztó, ezért nem változik meg az *Eredmeny*. Ha a kitevő 1, akkor csak az N kerül be a prímtényezős felbontásba, egyébként pedig a belső ciklusban meghatározott kitevővel kell kiírni. Ennek a szelekciónak igazából csak két ága van:

```
Ha Kitevő = 1 Akkor
    szorzásjel és az N hozzáfűzése az eredményhez
Egyébként Ha Kitevő <> 0 Akkor
    szorzásjel és az N^Kitevő hozzáfűzése az eredményhez
Elágazás vége
```

A teljes kód a CD-n található. A külső ciklusban beírtuk az állapotsorba az N-et, hogy követni lehessen a számolást.

### *A prímtényezős felbontás módosítása*

Ha kipróbáljuk a programot, akkor találunk benne egy kis szépséghibát. A szorzásjelet az első szám elé is kiírja. Amíg nem tudunk karakterláncokat kezelni, addig ezt egy kis trükkel tüntetjük el. A szorzásjelet egy változóba írjuk, és ezt a változót fűzzük hozzá az *Eredmeny*-hez a cikluson belül. Először azonban a változónak üres sztringet adunk értékül:

```
Szorozva = ""
```

majd az *Eredmeny* első értékadása után átírjuk szorzásjelre:

```
Szorozva = " . "
```

Sajnos ezt a feltételes elágazás mindkét ágába bele kell írunk, mert nem tudjuk, hogy melyik fog először bekövetkezni. A program minden egyes prímtényező kiírása után elvégzi ezt az értékadást, de ez elhanyagolható mértékben befolyásolja a futási időt, hiszen általában csak néhány tényezőről van szó.

Ha már úgyis módosítjuk a programot, akkor végezzünk el még néhány változtatást. Vegyük észre, hogy a kétágú szelekció egy *Select ... Case* utasítással is helyettesíthető, melynek *Case 0* ágát üresen hagyjuk, de felhasználjuk a *Case Else* ágat:

```
Select Case Kitevo
Case 0
    ' Ekkor nem kell hozzáfűzni semmit az Eredmeny-hez.
Case 1
    Eredmeny = Eredmeny & Szorozva & N
    Szorozva = " . "
```

```

Case Else
  Eredmeny = Eredmeny & Szorozva & N & " <SUP>" & Kitevo & "</SUP>"
  Szorozva = " . "
End Select

```

Következő módosításunk megfelel a futási időt. Az oszthatóságot az  $N = 2$  kivételével ugyanis elegendő csak páratlan számokra kipróbálni, mert 2-vel már ahányszor csak lehet, elosztottuk a számot. A lépésköz módosítását egy változó segítségével végezzük el, aminek először 1-et adunk értékül, mert a 2 után a 3 következik az osztók sorában:

```
Lepeskoz = 1
```

A külső ciklus végén az  $N$ -et megnöveljük a *Lepeskoz*-zel, majd átírjuk az értékét 2-re, hogy az  $N = 3$  után már csak a páratlan számok következzenek:

```

N = N + Lepeskoz
Lepeskoz = 2

```

Ezt az értékadást sajnos minden ciklusban el fogja végezni az interpreter, de felesleges lenne egy feltételes elágazásba írni, mert az még jobban megnövelné a futási időt. Ez a növekedés azonban elhanyagolható ahhoz képest, hogy felére csökkentettük az ismétlések számát.

Utolsó módosításunk nagyon jelentős mértékben csökkenti a futási időt. Matematikai eszközökkel belátható, hogy a használt algoritmussal elegendő egy szám négyzetgyökéig megvizsgálni az osztókat, így a külső ciklus ismétlési feltételét a következőképpen változtatjuk meg:

```

Do
  ...
  MaxN = Int(Sqr(Szam))
Loop While Szam > 1 And N <= MaxN

```

Egészrészt (*Int* függvény) azért számoltunk, mert az  $N$  is egész, és gyorsabb lesz az összehasonlítás.

Ha a *Szam* az osztások során nem csökkent le 1-re, akkor prímszám maradt az első hatványon. Ezt a ciklusban nem fűztük hozzá az *Eredmeny*-hez, így külön kell megtennünk:

```

If Szam > 1 Then
  Eredmeny = Eredmeny & Szorozva & Szam
End If

```

A módosított programot a fenti változtatásokkal a **4–49. példa** mutatja be. Próbáljuk ki az eredetivel együtt a következő számokra!

```

344157184
235342110
2334565
102400007

```

A számítási idők csökkenése önmagáért beszél.

## Strukturált algoritmusok

Ebben a fejezetben áttekintettük azokat a vezérlőelemeket, melyeket egy program megírásánál alkalmazhatunk. A szelekciók és iterációk vezérlik az utasítások végrehajtását, meghatározzák a végrehajtás sorrendjét. A feltételes elágazás egy-egy ágában, a cikluson belül vagy az egyes vezérlési szerkezetek között az utasítások már a leírt sorrendben hajtódnak végre. Az egymás után írt utasítások sorozatát szekvenciának nevezzük.

Szekvencia: egymás után végrehajtandó utasítások sorozata.

A szekvencia egy-egy utasítása természetesen lehet szelekció vagy iteráció is.

Eseménykezelő eljárásaink szekvenciákból, szelekciókból és iterációkból álltak. A programozásban nagyon fontos szerepet játszik az úgynevezett Böhm-Jacopini tétel, mely szerint minden olyan algoritmus felépíthető ebből a három elemből, melynek egy belépési és egy kilépési pontja van. Az eljárásokban a *Sub* utasítás jelentette a belépést, az *End Sub* a kilépést.

Strukturált algoritmus: olyan algoritmus, amely csak szekvenciákból, szelekciókból és iterációkból áll.

Megemlítjük, hogy az iterációk közül elegendő lenne az előtesztelő, feltételvezérelt ciklus alkalmazása. A hátultesztelő ciklus helyettesítéséhez a ciklus előtt olyan kezdőértéket kell adni a változóknak, hogy először ne teljesüljön az ismétlési feltétel, így a ciklus egyszer biztosan lefut. Léptető ciklust pedig a következő módon hozhatunk létre:

```
' A ciklusváltozó inicializálása:
I = Kezdőérték
Ciklus amíg I <= Végérték
    ' a ciklus utasításai
    I = I + Lépésköz
Ciklus vége
```

A strukturált szerkezet lényegesen megnöveli a programok áttekinthetőségét, megkönnyíti a hibakeresést és a módosítást. A továbbiakban is törekedni fogunk arra, hogy eljárásaink és függvényeink strukturált módon épüljenek fel.

A strukturált algoritmust ne keverjük össze a forráskód strukturált elrendezésével. Ez utóbbi a formára, az előző kifejezés pedig a belső szerkezetre utal, bár szoros kapcsolatban vannak egymással.

## Moduláris programozás

A klasszikus, algoritmusvezérelt programok szigorúan ragaszkodtak az algoritmusok strukturált megvalósításához. Az eseményvezérlés szétfeszítette a strukturált szerkezet kereteit, hiszen szkriptjeink önálló, egymástól függetlenül működésbe lépő eseménykezelő eljárásokból állnak. Az eseményvezérelt programozási módszereknél előtérbe került a feladat önálló részekre, modulokra bontása.

Moduláris program: olyan program, amely önálló feladatokat végző modulokból áll. A modulok meghatározott módon kapcsolódnak a környezetükhöz, és általában események hatására lépnek működésbe.

Eseményeket nem csak a felhasználó, hanem maguk a modulok is létrehozhatnak, így más modulok működését indítják el.

A moduláris és a strukturált programozás nincsen ellentétben egymással. Egy hosszadalmas eseménykezelő eljárást célszerű felbontani különálló részekre (szubrutinokra), egy modulon belül pedig strukturáltan valósítjuk meg az algoritmust. Mivel a modul viszonylag rövid, az algoritmusokból való kiugrás (az *Exit* utasítás használata) nem rontja az áttekinthetőséget. Ezért – általában a hibakezelésnél – megengedjük az alkalmazását.





## 5. TÖMBÖK ÉS TÁBLÁZATOK

Eddigi programjainkban csak néhány változót használtunk. Nagy mennyiségű adat tárolására kényelmetlen módszer lenne mindegyiknek külön nevet adni. A feldolgozásnál az utasításokat minden egyes változóval le kellene írni. Ezeknek az ismétléseknek az elkerülésére a programokban tömböket alkalmazunk.

**Tömb:** azonos névvel ellátott, sorszámmal megkülönböztetett változók csoportja.

**Index:** a változó sorszáma a tömbben.

A tömb által tartalmazott változókat a tömb elemeinek nevezzük.

Indexekkel ellátott jelölésekkel nem csak a programozásban találkozunk. Egy egyenlet megoldásait  $x_1$ ,  $x_2$ -vel, egy sorozat elemeit  $a_1$ ,  $a_2$ , ...,  $a_n$ -nel jelöljük.

A későbbiekben megismerkedünk olyan tömbökkel is, melyek elemeinek több indexe lesz. Az elemek indexeinek a számát a tömb dimenziójának nevezzük.

**Dimenzió:** a tömbelemek indexeinek a száma.

Egyelőre csak egydimenziós tömbökkel foglalkozunk.

### 5.1. Egydimenziós tömbök

#### Tömbök a VBScriptben

A VBScriptben a tömbök elnevezésére ugyanazok a szabályok vonatkoznak, mint az egyszerű változókra. Egy elemet a tömb nevével, és a kerek zárójelek közé írt indexszel adunk meg:

```
Megoldas(1), Megoldas(2), Vasarlo(27)
```

Az elemek sorszáma mindig 0-tól kezdődik. A tömböt deklarálni kell, deklarációjában megadjuk az index maximális értékét:

```
Dim Tomb(10), LottoSzam(4)
```

A fenti deklarációban a *Tomb* nevű tömb 11 elemet tartalmaz, melyek sorszáma 0-tól 10-ig változhat. A *LottoSzam* nevű tömb 5 elemet tartalmaz 0-tól 4-ig indexelve. A deklarációban a maximális index csak egy konkrét szám lehet. Nem írhatunk kifejezést (változót és konstans sem) a helyére.

Néha nem használjuk fel a nulla indexű elemet. Ezzel némileg pazaroljuk a memóriát, de szemléletesebbé tesszük a kódot. A hétköznapi életben is a sorszám kezdőértéke általában 1.

A hivatkozásokban a tömb elemeinek indexeként tetszőleges, numerikus értéket eredményező kifejezést megadhatunk:

```
Megoldas(I), SorozatEleme(2 * N + 1)
```

Ha az index törtszám, akkor a VBScript a legközelebbi egészre kerekíti. Ha az index nem a deklarációban megadott határok közé esik, akkor hibaüzenetet kapunk.

A tömbök elemei értékadó utasítással kapnak értéket:

```
Megoldas(1) = (-B + Sqr(Diszkriminans) / (2 * A))
```

Az **5–1. példában** a tömb elemeit az *InputBox* segítségével olvassuk be:

```
Dim Szam(4)
For I = 0 To 4
    Szam(I) = InputBox(I + 1 & ". szám:", "Beolvasás", 0)
Next
```

Az *InputBox* üzenetében a sorszámot is megadjuk, hogy a felhasználó követni tudja a bevittet. Mivel az indexelés 0-tól kezdődik, a beolvasott adat sorszáma éppen 1-gyel nagyobb, mint a tömb indexe.

Az adatokat egy másik ciklussal jelenítjük meg. Itt is sorszámozzuk az elemeket:

```
For I = 0 To 4
    document.write(I + 1 & ". szám: " & Szam(I) & "<BR>")
Next
```

Bár a fenti példában számokról beszéltünk, a VBScriptben egy tömb tetszőleges típusú értékeket tartalmazhat. Megtehetjük, hogy munkatársunk nevét, életkorát és házasság voltát egyetlen tömb elemeiben tároljuk:

```
Munkatars(0) = "Kovács István"
Munkatars(1) = 32
Munkatars(2) = True
```

A legtöbb programozási nyelv nem ennyire rugalmas, és megköveteli, hogy egy tömb elemei azonos típusúak legyenek.

### Tömbök használata

A tömböket – az egyszerű változókhoz hasonlóan – értékadó utasításokban, kifejezésekben egyaránt felhasználhatjuk. Írjunk programot, amely beolvassa 5 kör sugarát, majd táblázatos elrendezésben kiírja a képernyőre a kerülettel és a területtel együtt.

Az adatokat egy *Kor* nevű tömbben tároljuk, melynek indexe 0-tól 4-ig terjed:

```
Dim Kor(4)
```

A beolvasáshoz most is az *InputBox* függvényt használjuk:

```
For I = 0 To 4
    Kor(I) = InputBox("Adja meg a kör sugarát!", _
        "Kerület és terület", 0)
Next
```

A beolvasás után az adatok formázásával összeállítjuk a táblázatot:

```
Tablázat = ""
For I = 0 To 4
    ' A sorszám hozzáfűzése:
    Tablázat = Tablázat & I + 1 & ". kör"
```

```
' A sugár hozzáfűzése:
Tablázat = Tablázat & Formaz(Kor(I), 6, 2)
' A kerület hozzáfűzése:
Tablázat = Tablázat & Formaz(Kor(I) * 2 * Pi, 8, 2)
' A terület hozzáfűzése:
Tablázat = Tablázat & Formaz(Kor(I)^2 * Pi, 9, 2)
Tablázat = Tablázat & "<BR>"
Next
```

A  $\pi$  értékét konstansként tároljuk. Minden sorban először kiírjuk a kör sorszámát. Ez éppen 1-gyel nagyobb, mint a ciklusváltozó I értéke. Ezután megjelenítjük a kör sugarát, kerületét és területét. A végén ne feledkezzünk el a soremelésről!

A CD-n szereplő **5–2. példa** az *InputBox* üzenetének szövegében közli a felhasználóval a kör sorszámát is, a táblázat kiírása előtt pedig fejléctet készít. A formázott megjelenéshez az adatokat egy PRE-objektumba foglalja. A *Formaz* függvény használata miatt csatolnunk kell a kódhoz a *Formaz.vbs* fájlt. Az egész programot a *window*-objektum *onload* eseménykezelő eljárásába tettük.

### Iterációk a tömbök elemeire

A tömbök elemeit az előző példához hasonlóan iterációkkal kezeljük. Az elemek megkétszerezését például a következő ciklussal érhetjük el:

```
For I = 1 To MaximalisIndex
    Tomb(I) = Tomb(I) * 2
Next
```

Az elemeket át is rendezhetjük. Az **5–3. példa** olyan programot mutat be, amely egy tömb minden egyes elemét átírja az eggyel nagyobb indexű helyre, az utolsó elemet pedig a tömb elejére helyezi.

Mivel a beírás törli az eredetileg ott lévő elemet, az utolsó értéket először félretesszük egy ideiglenes (Temp: temporary) változóba:

```
Temp = Tomb(MaximalisIndex)
```

Az egyes elemeket egy ciklus segítségével az eggyel nagyobb indexű helyre tesszük. Az utolsó előtti helyről indulunk, és visszafelé haladunk, hogy az áthelyezés során ne töröljük ki az elemeket:

```
For I = MaximalisIndex - 1 To 0 Step -1
    Tomb(I + 1) = Tomb(I)
Next
```

A végén a félretett utolsó elemet betesszük az első helyre:

```
Tomb(0) = Temp
```

A CD-n lévő példában a léptetést és a listázást egy-egy szubrutin segítségével végezzük. A *window onload* eseménykezelőjében először értéket adunk a tömb elemeinek (mindegyiknek a saját indexét), utána kilistázzuk az eredeti sorrendet. A parancsgomb eseménykezelőjében meghívjuk a léptetést, majd a listázást végző eljárást.

### A legkisebb és legnagyobb tömbelem kiválasztása

A tömbök kezelésénél gyakori feladat a legkisebb vagy a legnagyobb elem megkeresése. A két módszer megegyezik egymással, csak a használt relációt kell megfordítani.

A legkisebb elem keresésénél először a tömb elején álló elemet tekintjük a legkisebbnek, aztán sorra összehasonlítjuk a többi elemmel. Ha találunk nála kisebbet, akkor annak az indexét jegyezzük fel:

```
LegkisebbElemIndexe = 0
Ciklus I = 1-től a maximális értékig
    Ha Tömb(I) < Tömb(LegkisebbElemIndexe) Akkor
        LegkisebbElemIndexe = I
    Elágazás vége
Ciklus vége
```

A pszeudokóddal leírt algoritmus alapján könnyű elkészíteni a VBScript programot. A forráskódot a CD-melléklet **5–4. példájában** találjuk. Először véletlenszámokkal töltünk fel egy 10-elemű tömböt, majd megkeressük a legkisebb, s ugyanilyen módon a legnagyobb értéket. A két szám mellett kilistázzuk a tömb minden elemét.

Az előző példákban zavaróan hat, hogy a lista végén is szerepel az a pontosvessző, amit az elemek elválasztására használunk. Fölöslegesen bonyolítaná a programot, ha a listát összeállító cikluson belül mindig megvizsgálnánk, hogy elérkeztünk-e már az utolsó elemhez. Helyette futtassuk a ciklust az utolsó előtti elemig. Az utolsó elemet a ciklus után, elválasztó jel nélkül fűzzük hozzá a listához (**5–5. példa**):

```
Lista = ""
For I = 0 To 8
    Lista = Lista & Tomb(I) & "; "
Next
Lista = Lista & Tomb(9)
```

Sztringfüggvények segítségével egyszerűen megoldhatjuk az elválasztójel törlését.

### Két tömbelem cseréje

Hamarosan szükségünk lesz arra, hogy két adott indexű tömbelemet felcseréljünk. Ehhez az egyik elemet egy ideiglenes változóba tesszük. Utána a másikat már áttehetjük az egyikbe (ezzel természetesen az eredeti tartalmát töröltük, éppen ezért tettük félre). Végül a félretett értéket írjuk a másik tömbelem helyére.

Az I és J indexű tömbelemeket tehát a következő kóddal lehet felcserélni:

```
Temp = Tomb(I)
Tomb(I) = Tomb(J)
Tomb(J) = Temp
```

Figyeljünk arra, hogy az utasítások sorrendje nem változtatható meg, mert akkor a félre nem tett elemet fogjuk felülírni!

A CD-melléklet **5–6. példája** véletlenszámokkal feltölt, majd kilistáz egy 10-elemű tömböt. Két szövegmező segítségével meg lehet adni a felcserélendő elemek indexét. A csere után ismét kilistázza a tömböt.

Ezt a módszert természetesen nem csak két tömbelem, hanem bármely két változó értékének felcserélésénél alkalmazhatjuk.

### A tömbelemek rendezése

Gyakran előfordul, hogy egy tömb elemeit nagyság szerint növekvő vagy csökkenő sorrendbe kell rendeznünk. Emlékezzünk vissza arra, hogy sztringeket is össze tudunk hasonlítani egymással, így rendezni nem csak numerikus változókat lehet.

A rendezési algoritmusok kidolgozása és hatékonysága fontos kérdés a programozásban. A legegyszerűbben úgy hajtánánk végre a feladatot, hogy megkeressük egy tömb legkisebb elemét, majd betesszük egy másik, egyelőre üres tömb elejére. Aztán megkeressük a maradék legkisebb elemét, betesszük a második helyre, és így tovább. Ez az algoritmus azonban nem túl hatékony. A másik tömb miatt kétszer akkora memóriát foglal el, és ha már nincs rá szükség, akkor – egyelőre – nem tudjuk felszabadítani a helyét. A továbbiakban a rendezést úgy értjük, hogy nem egy másik tömbbe helyezzük át az elemeket, hanem a tömbön belül változtatjuk a sorrendet.

Az előbb vázolt algoritmus új tömb felhasználása nélkül úgy hajtható végre, hogy megkeressük a legkisebb elemet, és felcseréljük a tömb elején álló elemmel. Aztán a maradékból keressük meg a legkisebbet, felcseréljük a második helyen álló elemmel, és így tovább:

```
Ciklus a tömb elemeire
  a legkisebb elem megkeresése
  cseréje az adott elemmel
Ciklus vége
```

A keresést végző utasításokat egy újabb ciklusba tesszük. A ciklusváltozó kezdőértékét minden végrehajtás után 1-gyel megnöveljük, mert az addigi elemek már a helyükre kerültek. Ezért a belső ciklust a külső ciklus változójának értékétől kezdve indítjuk. A keresést a már ismert algoritmus segítségével végezzük:

```
Ciklus I = 0-tól MaximálisIndex-1-ig
  LegKisebbElemIndexe = I
  Ciklus J = I+1-től MaximálisIndex-ig
    Ha Tömb(J) < Tömb(LegKisebbElemIndexe) Akkor
      LegKisebbElemIndexe = J
    Elágazás vége
  Ciklus vége
  a Tömb(I) és a Tömb(LegKisebbElemIndexe) cseréje
Ciklus vége
```

A két elem cseréjét a már ismert módon hajtjuk végre.

Az eljárás forráskódját a CD-melléklet **5–7. példájában** találjuk. Itt egy tízelemű tömbnek véletlenszámokat adunk értékül, majd a fenti algoritmussal rendezzük. A weblapon megjelenítjük mind az eredeti, mind pedig a rendezett elemek listáját. Az **5–8. példában** a sorrendet a belső ciklus minden egyes lefutása után kiírjuk. Figyeljük meg, hogy a legkisebb elemek sorra a tömb elején gyűlnek össze!

Az alkalmazott módszer miatt ezt az eljárást minimumkiválasztásos rendezésnek hívják. A sok összehasonlítás és cserélgetés miatt nem túl hatékony algoritmus. A rendezési módszerekre még visszatérünk.

### Az indexhatár ellenőrzése

Már említettük, hogy ha túllépjük a tömb deklarációjában megadott legnagyobb indexet, akkor hibajelzést kapunk, és leáll a program futása. Ezért ha tömböket használunk az adatok tárolására, mindig ellenőriznünk kell, hogy nem értük-e el az indexek maximális értékét.

Az **5–9. példában** beolvassuk, majd megjelenítjük egy osztály tanulóinak névsorát. Ha általános célú programot akarunk írni, akkor meg kell becsülnünk, legfeljebb hány eleme lehet a tömbnek. A deklarációt ennek a becslésnek megfelelően kell elvégezni. Ha úgy gondoljuk, hogy az osztálylétszám nem lépi túl az 50-et, akkor a

```
Dim TanuloNev(49)
```

deklarációt használhatjuk. Példánkban nem akarunk sok adatot begépelni, ezért a maximális indexet 4-nek vesszük:

```
Dim TanuloNev(4)
```

Mivel 0-tól indul a sorszámozás, ez 5 tanuló nevének a tárolását teszi lehetővé.

A beolvasást egy *NevBe* azonosítójú szövegmező és egy Beolvas feliratú parancsgomb segítségével végezzük. A tömbelemek indexelésére a *Darab* változót használjuk, melyet minden név beolvasása után 1-gyel növelünk. Az első név beolvasása előtt értékét -1-re állítjuk. A későbbiekben ezzel azt is jelezzük, hogy még egyetlen tömbelem sem kapott értéket:

```
Darab = -1
```

Figyeljünk arra, hogy az ilyen célra szolgáló változót globálisként kell deklarálni!

A parancsgomb *onclick* eseménykezelője először megvizsgálja, hogy a *Darab* értéke elérte-e a deklarációban szereplő legnagyobb értéket. Ha már egyenlő 4-gyel, akkor hibaüzenetet ad, és ráállítja a fókuszt a listázás gombra:

```
If Darab = 4 Then  
    window.alert("Nem írhat be több nevet.")  
    Listaz.focus()
```

Ha még nem írtuk tele a tömböt, először növeljük a *Darab* értékét, majd az általa meghatározott elembe tároljuk a szövegmezőbe írt nevet. Az adatbevitel megkönnyítése céljából visszaállítjuk a fókuszt a szövegmezőre:

```
Else  
    Darab = Darab + 1  
    TanuloNev(Darab) = NevBe.value  
    NevBe.focus()  
    NevBe.select()  
End If
```

A kiírást a szokásos módon, egy újabb parancsgomb segítségével végezzük. A listát a kiírás előtt egy sztringben gyűjtjük össze, amit először törölünk:

```
Lista = ""
```

A beolvasások után a *Darab* változó értéke éppen megegyezik az utoljára megadott név indexével, így a kiírásnál a ciklusváltozót 0-tól *Darab*-ig futtatjuk. A listában a tanuló neve elé sorszámot is írunk. Ez eggyel nagyobb, mint a ciklusváltozó értéke:

```
For J = 0 To Darab  
    Lista = Lista & J + 1 & ". " & TanuloNev(J) & "<BR>"  
Next
```

Ezek után már csak a *Lista* sztringjét kell átadni az eredményt megjelenítő bekezdés *innerHTML* tulajdonságának. A teljes forráskód a CD-n található.

Barátságosabb felhasználói felületet kapunk, ha a szövegmező elé kiírjuk a sorszámot. Így tájékoztatjuk a felhasználót arról, hogy hol tart az adatbevitelben. A módosított programot az **5–10. példa** tartalmazza.

## Tömböket kezelő függvények

A tömbök elemeinek értékét általában a felhasználótól kapjuk, vagy valamelyik háttértárról olvassuk be. Esetenként előfordulhat, hogy a programunkban kell megadni, minden egyes elemre külön-külön. Ez a felsorolás kissé hosszadalmas kód begépelését igényli.

Ezt a munkát egyszerűsíthetjük az *Array* függvény használatával. Az *Array* a paramétereként felsorolt értékekből tömböt készít. A tömb nevét zárójelek és a maximális index megadása nélkül kell deklarálni. A következő utasítások végrehajtása után:

```
Dim Tomb  
Tomb = Array(2, 4, 6)
```

a *Tomb* elemeinek értéke:

```
Tomb(0) = 2  
Tomb(1) = 4  
Tomb(2) = 6
```

Megjegyezzük, hogy ilyen módon igazából egy tömböt rendeltünk egy egyszerű változóhoz, de a különbség egyelőre nem lényeges a számunkra.

Hamarosan szükségünk lesz arra, hogy a program futása során meghatározzuk egy előzőleg deklarált tömb maximális indexének értékét. Ezt az *UBound* (upper bound, felső korlát) függvénnyel tehetjük meg. A

```
Dim Tomb(20)
```

deklaráció esetén az *UBound(Tomb)* értéke 20.

## Tömbkezelő függvények használata

Az **5–11. példa** programja betűkkel ír ki egy 1000-nél kisebb pozitív egész számot. A számot a *SzamBe* azonosítójú szövegmezőbe lehet beírni, az eredményt a *SzamnevKi* azonosítójú SPAN-objektumban jelenítjük meg. A kiírás a *Gomb* parancsgombra való kattintással történik meg.

A betűkkel kiírt számneveket az *EgyesTomb*, *TizesTomb*, *SzazasTomb* tömbökben tároljuk, amelyek indexeként a megfelelő helyiértéken álló számjegyet használjuk. A tömbök nulladik elemeként üres sztringet adunk meg. Így ha az adott helyiértéken 0 áll, akkor a számnévbe nem kerül semmi:

```
EgyesTomb = Array("", "egy", "kettő", ...)  
TizesTomb = Array("", "tíz", "húsz", ...)  
SzazasTomb = Array("", "száz", "kétszáz", ...)
```

A parancsgomb eseménykezelőjében a szövegmező értékét szokás szerint átadjuk egy változónak, mert többször is felhasználjuk:

```
Szam = SzamBe.value
```

A szám felbontása számjegyekre a maradékos osztás segítségével történhet. A százasként álló számjegyet megkapjuk, ha a számot maradékosan elosztjuk 100-zal:

```
SzazasJegy = Szam \ 100
```

Az egyesek helyén álló számjegyet megkapjuk, ha a számot elosztjuk 10-zel, és az osztás maradékát vesszük:

```
EgyesJegy = Szam Mod 10
```

A középső számjegyhez két művelettel juthatunk. A számot maradékosan elosztjuk 10-zel, majd vesszük a hányados 10-zel való osztásának a maradékát:

```
TizesJegy = (Szam \ 10) Mod 10
```

Javasoljuk az Olvasónak, hogy néhány háromjegyű számmal próbálja ki a felbontást!

Az így kapott *SzazasJegy* és *EgyesJegy* változók értéke már közvetlenül felhasználható a számneveket tartalmazó tömbök indexelésére. Sajnos a tízeseknél figyelniünk kell arra, hogy ha az egyesek helyén nem 0 áll, akkor a számnévbe „tizen” illetve „huszon” kerül. Ezért bevezetünk egy *Tiz* nevű segédváltozót, melynek először a *TizesTomb* megfelelő elemét adjuk értékül. Ha az egyesek helyén nem nulla áll, akkor egy *Select Case* utasítással módosítjuk a segédváltozó értékét:

```
Tiz = TizesTomb(TizesJegy)  
If EgyesJegy > 0 Then  
    Select Case TizesJegy  
        Case 1  
            Tiz = "tizen"  
        Case 2  
            Tiz = "huszon"  
    End Select  
End If
```



Ezután már csak az eredmény megjelenítése következik:

```
SzamnevKi.innerText = SzazasTomb(SzazasJegy) & Tiz & _  
EgyesTomb(EgyesJegy)
```

A *SzazasJegy* változó igazából felesleges, értékét közvetlenül beírhatnánk az output utasításba. Egy kis változtatással elhagyhatnánk a százask megnevezésének tömbjét is, helyette szükség esetén az egyesek tömbjeinek elemeihez kellene hozzáfűzni a „száz” karaktersorozatot.

## A tömbök memóriaigénye

Tömbök használatával könnyen elérhetjük az interpreter memóriakezelésének a korlátait. Bár a Windows operációs rendszer elég rugalmasan biztosítja az alkalmazások számára a memóriát, a VBScript nem nyújt lehetőséget ennek szabályozására.

Számítógépünkön az **5–12. példa** tömbjének létrehozásával az interpreter már nem tudott megbirkózni:

```
Dim A(1000000000)  
window.alert("Elkészült a tömb!")
```

és „Kevés a memória” hibaüzenettel leállt. Ha tízmilliót írtunk a maximális index helyére, akkor néhány másodperc alatt készült a tömb (**5–13. példa**). A határ a két eset között attól is függ, hogy milyen felhasználói és rendszerprogramok futnak a háttérben.

A tömb deklarálásánál még nem kapnak értéket az elemek. Az értékadás során tovább növekedhet a szükséges memória mérete. Többdimenziós tömbök használata esetén is rohamosan nő a memóriaigény.

A memóriával takarékosan kell bánni. Sajnos a tömböt deklaráló utasításban egyelőre nem használhatunk változót maximális indexként, így értékét nem tudjuk hozzáigazítani a futtatás során felmerülő igényekhez. A dinamikus tömbök segítségével azonban lehetőségünk nyílik arra, hogy futás közben módosítsuk a tömb méretét.

## 5.2. Kollekción

A HTML-kód objektumaira eddig azonosítóik alapján hivatkoztunk. A dokumentum objektummodell azonban speciális tömböket biztosít a számunkra, melyek elemei a weblap objektumai. Ilyen módon indexük alapján is elérjük az objektumokat, illetve tulajdonságaikat. Ezeket a tömböket kollekciónak (gyűjteményeknek) nevezzük.

Kollekción: a HTML-kód objektumaiból álló, speciális tömb.

### A document.all kollekción

A HTML-kód összes objektumát a *document*-objektum *all* (minden) kollekciónja tartalmazza. Ezt a gyűjteményt a böngésző hozza létre a weblap megjelenítésekor. Az *all* kollekciónra a *document*-objektum tulajdonságaihoz hasonlóan hivatkozhatunk:

```
document.all
```

A *length* tulajdonság megadja a dokumentumot alkotó objektumok számát:

```
document.all.length
```

Az *all* kollekcióban a dokumentum objektumai a HTML-kódban való előfordulás sorrendjében egy 0-val kezdődő indexet kapnak (az utolsó objektum indexe tehát *length* – 1). Ezzel az indexszel úgy hivatkozhatunk rájuk, mintha egy tömb elemei lennének:

```
document.all(I)
```

Ez a hivatkozás egyenértékű a dokumentum azonosítójának kiírásával. Írhatjuk, olvashatjuk bármely tulajdonságát, meghívhatjuk metódusait. A következő utasítás például megadja az *I* indexű objektum azonosítóját:

```
document.all(I).id
```

Az objektumok rendelkeznek egy *tagName* (osztálynév) tulajdonsággal is, amely megadja az objektum osztályát. Ezt a tulajdonságot nem tudjuk megváltoztatni, csak olvasni. Az **5–14. példa** ciklusa kilistázza a weblap összes objektumának indexét, osztályát és zárójelbe téve az azonosítóját (*id*):

```
Lista = ""
For I = 0 To document.all.length - 1
    Lista = Lista & I & ". " & document.all(I).tagName & _
        " (" & document.all(I).id & ")" & vbCrLf
Next
window.alert(Lista)
```

A CD-mellékleten található dokumentumba elhelyeztünk néhány objektumot, hogy szemléltessük az eredményt. Figyeljük meg, hogy nem minden objektumnak adtunk azonosítót!

### Iterátor alkalmazása

Az előző példa ciklusát a kollekció minden elemére végrehajtottuk. Ilyenkor használhatjuk a *For Each ... In* (a ... minden elemére) utasítást, amely az *In* kulcsszó után álló kollekció minden elemére végrehajtja az iterációt. Ciklusváltozót ilyenkor is ki kell jelölnünk, amivel a ciklusmagban a kollekció objektumaira hivatkozunk:

```
For Each Elem In Kollekcio
    ...
Next
```

Az *Each* kulcsszót gyakran iterátornak nevezik, mert iterálja a kollekció összes elemét.

A *For Each* utasítás alkalmazása nem teszi szükségessé az elemek számának a meghatározását az **5–15. példa** ciklusában (a ciklusmagot itt kissé leegyszerűsítettük):

```
Dim Elem
For Each Elem In Document.all
    Lista = Lista & Elem.tagName & "; "
Next
```

Az *Elem* nevű változó (amit természetesen bárhogyan nevezhetnénk) a kollekció objektumaira hivatkozik. Segítségével elérhetjük az objektumok tulajdonságait és metódusait.

Figyeljünk arra, hogy ilyenkor nem kell indexet használni! Meghagyhattuk volna az *I* jelölést, de az *Elem* elnevezés jobban mutatja a ciklusváltozó szerepét.

Iterátort a szokásos egydimenziós tömbökre is alkalmazhatunk, ha az összes elemet fel kell dolgoznunk a ciklusban. Ekkor a ciklusváltozó minden egyes ismétlésnél felveszi a tömb egy-egy elemének értékét. Így csak az értékadó utasítások jobb oldalán használhatjuk (**5–16. példa**):

```
For Each Elem In Tomb
    Lista = Lista & Elem & "; "
Next
```

Az *Elem = Kifejezés* utasítás hatására az *Elem* nevű változó ugyan megkapná a kifejezés értékét, de ez nem adódik át a tömbnek!

A *For Each* előtesztelő ciklus, ha tehát egyetlen eleme sincs a tömbnek vagy a kollekciónak, akkor egyszer sem hajtódik végre.

### Azonos osztályú objektumok kollekciói

Bár a dokumentum objektum modell az *all* kollekción kívül biztosít a számunkra még néhány fontos gyűjteményt, mi magunk is létrehozhatunk ilyen csoportot az azonos osztályba tartozó objektumokból. Erre az *all* kollekció *tags* metódusa ad lehetőséget, amely kollekciót képez a paramétereként megadott osztály objektumaiból:

```
document.all.tags("OsztályNév"), például: document.all.tags("SPAN")
```

A példában szereplő kollekciónak már csak a SPAN-objektumok a tagjai. Ez a kollekció ugyanúgy használható az elemek kezelésére, mint az *all*.

Mint látjuk, az objektumosztály megnevezését sztringként kell megadni, és tetszőleges, sztringet eredményező kifejezés lehet. Ha a megjelölt osztályból egyetlen objektum sincs a dokumentumban, akkor a metódus egy speciális értéket, a *Null*-t adja vissza, amivel majd később találkozunk.

Az **5–17. példában** a sorok első szavának a kiemeléséhez nyolc SPAN-objektumot helyeztünk el a HTML-kódban. A parancsgombok segítségével meghívunk egy eljárást, amely kiszínezi az objektumok szövegét. Az eljárásnak átadjuk a kívánt színt, majd egy ciklussal elvégezzük a színezést:

```
Sub Kiszinez(Szin)
    Dim Elem
    For Each Elem In document.all.tags("SPAN")
        Elem.style.color = Sin
    Next
End Sub
```

Ha ugyanezt az eredményt úgy akartuk volna elérni, hogy mindegyik SPAN-objektumnak egyedi azonosítója lenne, akkor nyolcszor kellett volna megismételni a betűszínt beállító utasítást!

Az első parancsgombra az INPUT-objektumok kollekciónak a segítségével állítottuk rá a fókuszt (a sorszámozás 0-val kezdődik):

```
document.all.tags("INPUT")(0).focus()
```

A *tags* metódussal létrehozott kollekciónak is van *length* tulajdonsága. Így meghatározhatjuk az elemek számát. Fenti példánkban a

```
document.all.tags("SPAN").length
```

értéke 8.

Az *Each* iterátor a kollekción összes objektumára végrehajtja a ciklust. Ha válogatni akarunk közöttük, akkor indexet használunk. Az **5–18. példában** a páros indexű SPAN-objektumok szövegét pirosra, a páratlanokét kékre színezzük (az indexelés 0-val kezdődik):

```
For I = 0 To document.all.tags("SPAN").length - 1 Step 2
    document.all.tags("SPAN")(I).style.color = "red"
    document.all.tags("SPAN")(I + 1).style.color = "blue"
Next
```

## Objektum-hivatkozások

Az előző példában kissé hosszú volt az utasítás, amellyel megadtuk egy SPAN-objektum színét. A

```
document.all.tags("SPAN")(I).style.color
```

kifejezésben az egyes részek jelentése:

document	maga a dokumentumobjektum
.all	a dokumentum összes objektumát tartalmazó kollekción
.tags("SPAN")	az összes objektum közül csak a SPAN-objektumokat tartalmazó kollekción
(I)	a SPAN kollekción I-edik objektuma
.style	az I-edik objektum stílus tulajdonsága
.color	a stílus tulajdonság betűszín eleme

Bár minden egyes rész jelentése világos és egyértelmű, mégis hosszadalmas leírni, nehéz áttekinteni az ilyen hivatkozásokat. A kifejezést objektum-hivatkozások alkalmazásával rövidíthetjük.

Egy objektum-hivatkozást először változóként kell deklarálni, majd a *Set* utasítással kapcsolhatjuk hozzá egy objektumhoz:

```
Dim Változónév
...
Set Változónév = ObjektumAzonosító
```

Az objektum-hivatkozásokat szokás objektumváltozóknak is nevezni.

Az **5–19. példában** bemutatjuk, hogyan rövidíthetjük az 5–18. példa ciklusát egy objektum-hivatkozás segítségével. Deklarálunk egy *SpanElem* nevű objektumváltozót:

```
Dim I, SpanElem
```

majd hozzárendeljük a *document.all.tags("SPAN")* kollekciót:

```
Set SpanElem = document.all.tags("SPAN")
```

Így a *SpanElem* a SPAN-objektumokat tartalmazó kollekcióra hivatkozik. A ciklusban a kollekció hosszú hivatkozása helyett a *SpanElem* megnevezést használjuk:

```
For I = 0 To SpanElem.length - 1 Step 2
    SpanElem(I).style.color = "red"
    SpanElem(I + 1).style.color = "blue"
Next
```

Az objektum-hivatkozások nem azonosak az objektumokkal, csak a memóriabeli címüket (helyüket) kapják meg. Így ugyanarra az objektumra több objektumváltozó is hivatkozhat (**5–20. példa**):

```
Set ElsoSzo = document.all.tags("SPAN")(0)
Set Hull = document.all.tags("SPAN")(0)
```

Ha megváltoztatjuk az egyiknek valamely tulajdonságát, akkor változni fog a másiké is, hiszen ugyanarról az objektumról van szó:

```
ElsoSzo.style.color = "green"
window.alert(Hull.style.color)
```

Egy hozzárendelés után magát a változót is használhatjuk a további hozzárendelésekhez:

```
Set ElsoSzo = document.all.tags("SPAN")(0)
Set Hull = ElsoSzo
```

Így mindkét változó ugyanazt az objektumot jelöli ki.

A *Set* utasítással egyelőre csak DHTML-objektumokat rendelünk a változókhoz. A későbbiekben a segítségével saját objektumainkra is hivatkozni fogunk.

## HTML-kollekciók létrehozása

A *tags* metódussal csak egy adott osztályhoz tartozó elemekből készíthetünk kollekciókat, és az osztály összes objektuma bekerül a gyűjteménybe. Szükség esetén a HTML-kód tetszőleges objektumaiból alkothatunk kollekciót. Ehhez egyforma azonosítóval kell őket ellátni. Ilyenkor a böngésző indexeli az azonos nevű objektumokat, amiket így a tömbelemekhez hasonlóan, vagy az *Each* iterátorral érhetünk el:

```
KözösAzonosító(Index)
```

Ez a hivatkozás egyenértékű az objektum egyedi nevével, bármilyen tulajdonságát vagy metódusát utána írhatjuk:

```
KözösAzonosító(Index).tulajdonságnév
```

Így a kiválasztott objektumokat ciklusokkal tudjuk kezelni, amit egyedi azonosítók esetén csak külön utasításokkal tehetnénk meg.

Az **5–21. példában** az 5–17. példa kódját úgy alakítjuk át, hogy a parancsgombra való kattintás után a címsor, az első versszak két szava és a teljes második versszak le-

gyen kiszínezve. A megfelelő objektumokat a *Szines* azonosítóval látjuk el. Az eseménykezelő *For Each* ciklusában a közös azonosítót adjuk meg kollekcióként:

```
For Each Elem In Szines
    Elem.style.color = Szin
Next
```

Mint minden kollekció, a közös azonosítóval ellátott objektumok kollekciói szintén rendelkeznek *length* tulajdonsággal. Elemeiket a fentiekhez hasonlóan indexekkel is elérhetjük, például:

```
Szines(3).style.color = "green"
```

### 5.3. Többdimenziós tömbök

#### Többdimenziós tömbök

A programozás során gyakran előfordul, hogy táblázatokat kell kezelnünk. A táblázatokban egy elem helyét a sor és az oszlop számával adhatjuk meg. Ehhez két indexet használunk, az adatokat kétdimenziós tömbben tároljuk. Az indexek értékét (a sor és az oszlop sorszámát) a tömb neve után zárójelben, vesszővel elválasztva adjuk meg. A VBScriptben a sorszámozás itt is 0-val kezdődik! A

```
Tomb(2, 3)
```

elem például a kettes indexű sorban és a hármas indexű oszlopban található. A sor és oszlop megnevezést csak a szemléltetés miatt használjuk, a programban nincs jelentősége.

A dimenziószámot tovább is növelhetjük, használhatunk három, négy stb. dimenziós tömböket is:

```
Tomb(2, 5, 1), Tomb(4, 11, 2, 2)
```

VBScriptben a dimenziók száma legfeljebb 60 lehet, de ritkán lépi túl a 3-at vagy a 4-et. Az index legkisebb értéke minden egyes dimenzió esetén 0. Az indexek legnagyobb értékét a tömb deklarációjában kell megadni. Egy háromdimenziós tömb esetén például a

```
Dim Tomb(2, 5, 4)
```

utasítás azt jelenti, hogy az első index legfeljebb 2, a második legfeljebb 5, a harmadik pedig legfeljebb 4 lehet. Mivel a sorszámozás 0-tól indul, a tömb elemeinek száma:  $(2 + 1) \cdot (5 + 1) \cdot (4 + 1) = 90$ .

A tömbök dimenziószámának növelésével rohamosan nő a méretük és a helyfoglalásuk a memóriában.

Többdimenziós tömbök esetén az *UBound* függvény használatánál a tömb neve mellett meg kell adni, hogy melyik dimenzió maximális indexértékét kérdezzük le. A fent deklarált háromdimenziós tömbnél például az

```
UBound(Tomb, 2)
```

értéke 5, mert a második index legnagyobb értékét határoztuk meg.

## Értékadás a tömbelemeknek

A többdimenziós tömbök elemeit egymásba ágyazott ciklusokkal kezelhetjük, ahol minden egyes ciklusváltozó egy-egy indexet képvisel.

Az **5–22. példában** olyan 10x10-es tömböt készítünk, amelynél az azonos indexű elemek értéke 1, a többi 0.<sup>18</sup> Mivel az indexelés 0-tól indul, a tömb deklarálása:

```
Dim Tomb(9, 9)
```

Az értékadást két egymásba ágyazott ciklussal végezzük el. Minden elem 0 értéket kap, de ha a két index egymással egyenlő, akkor átírjuk 1-re:

```
For I = 0 To 9
  For J = 0 To 9
    Tomb(I, J) = 0
    If I = J Then
      Tomb(I, J) = 1
    End If
  Next
Next
```

A CD-n lévő példa a *Formaz* függvény segítségével ki is listázza a tömbelemeket.

Ez a példa nagyon hasznos egyfajta programozói gondolkodásmód elsajátításának a szempontjából. Eddig is ügyeltünk arra, hogy lehetőleg ne alkalmazzunk kétágú szelekciót ott, ahol ezt egy előzetes értékadással elkerülhetjük. Most sem *If ... Then ... Else* utasítást írtunk a belső ciklusba:

```
If I = J Then
  Tomb(I, J) = 1
Else
  Tomb(I, J) = 0
End If
```

hanem szükség esetén átírtuk egyre az előzőleg adott zérus értéket.

De még tovább léphetünk. Az egész feltételes elágazást kikerülhetjük, ha először nullákkal töltjük fel a tömböt, majd egy másik ciklussal a megfelelő elemek helyére 1-et írunk. A szelekció elhagyásával gyorsabb és áttekinthetőbb programot kapunk (**5–23. példa**):

<pre>' 1. lépés: For I = 0 To 9   For J = 0 To 9     Tomb(I, J) = 0   Next Next</pre>	<pre>' 2. lépés: For I = 0 To 9   Tomb(I, I) = 1 Next</pre>
---	---

Ezzel a módszerrel 10 értékadást fölöslegesen végzünk el, de 100 vizsgálatot elhagytunk, ami jelentősen csökkenti a futási időt. A program áttekintését sem nehezíti feltételes elágazás. További egyszerűsítésként a `Tomb(I, I) = 1` utasítást beírhattuk volna az első lépés külső ciklusának végére is.

<sup>18</sup> Ez a matematikai egységmátrix modellje.

A feladat megoldására gyakran egy harmadik módszert ajánlanak, amely kihasználja, hogy a *True* -1-et, a *False* pedig 0-t jelent. Így egyetlen lépésben, szelekció nélkül tudunk értéket adni a tömb elemeinek:

```
Tomb(I, J) = Abs(I = J)
```

Ha az Olvasó rájött, hogy ez a megoldás miért jó, akkor az **5–24. példa** segítségével próbálja ki mindhárom algoritmust! Az érzékelhető időtartamhoz kissé megnöveltük a tömbméretet. A reális összehasonlításhoz nem alkalmaztunk folyamatjelzőt az állapotsorban, egy párbeszédablak megjelenése jelzi, ha kész az értékadás. Számítsunk arra, hogy a böngésző az indulás után hamarosan megszakítja a folyamatot, és rákérdez a folytatásra.

A példából egyértelműen kiderül, hogy az utolsó, „trükkös” megoldás a leghosszabb (és a legnehezebben érthető).<sup>19</sup> Ennél még a szelekciót tartalmazó program is gyorsabban végez. A legegyszerűbb megoldás a leggyorsabb: az előzetes értékadás, majd módosítás. Néha az egyenes út a legrövidebb!

### Adattárolás kétdimenziós tömbökkel

Az **5–25. példa** programja tárolja és megjeleníti a következő adatokat:

Név:	Születési év:	Születési hely:
Dékány Endre	1951	Debrecen
Kiss István	1963	Budapest
László Zsolt	1971	Szeged
Szabó László	1972	Győr

Az egyszerűség kedvéért az adatokat most nem egy szövegmező segítségével olvassuk be, hanem beleírtuk a forráskódba. Helykímélés céljából egy sorba több utasítást helyeztünk el. Ilyenkor kettősponttal kell őket egymástól elválasztani:

```
A(0, 0) = "Dékány Endre" : A(0, 1) = 1951 : A(0, 2) = "Debrecen"
```

A megjelenítést két egymásba ágyazott ciklus segítségével végezzük. A ciklusváltozók az indexek legkisebb értékétől a legnagyobb értékig futnak:

```
For I = 0 To 3
  For J = 0 To 2
    document.write(A(I, J)) & ", "
  Next
  document.write("<BR>")
Next
```

Emlékeztetünk arra, hogy a VBScript igen rugalmasan kezeli a tömbök elemeinek típusát. Sok programnyelvben az összes elem csak a tömb deklarációjakor megadott típusnak megfelelő, egyfajta értéket tartalmazhat.

---

<sup>19</sup> Az utolsó módszer több könyvben úgy szerepel, ahogy idéztük, bár abszolút érték helyett gyorsabb lenne az  $(I=J)$  ellentettjét meghatározni. De még ellentettképzéssel is ez a leglassabb megoldás.



Bár weblapunkon az egyes személyek adatait külön sorokba írtuk, a megjelenítés eléggé áttekinthetetlen. Készíthetnénk sztringekre is a *Formaz* függvényhez hasonló eszközt az esztétikus kivitelhez, egyszerűbb azonban, ha a DHTML táblázatobjektumát használjuk fel erre a célra.

## A táblázatobjektum

A táblázatobjektum a `TABLE` (táblázat) osztály egyede. A táblázat sorokat tartalmaz. Minden egyes sor a `TR` (table row, táblázat sor) osztály objektuma. A sorok cellákat tartalmaznak, melyek a `TD` (table data, táblázat adat) osztályhoz tartoznak. Mindegyik objektumnak van nyitó és záró tagja. A két tag között helyezkedik el a tartalmának a kódja.

Egy táblázat szerkezetét az **5–26. példa** mutatja:

```
<TABLE>
  <TR> <!--! 1. sor -->
    <TD> ... </TD> <!--! 1. sor, 1. cella -->
    ...
    <TD> ... </TD> <!--! 1. sor, utolsó cella -->
  </TR>
  <TR> <!--! 2. sor -->
    <TD> ... </TD> <!--! 2. sor, 1. cella -->
    ...
    <TD> ... </TD> <!--! 2. sor, utolsó cella -->
  </TR>
  ...
</TABLE>
```

Táblázat készítésekor az elején egy `<TABLE>`, a végén egy `</TABLE>` tagot kell beírunk a kódba. A sorok elejére egy `<TR>`, végére egy `</TR>` tag kerül. A sorokon belül a cellák kezdetét egy `<TD>`, végét pedig egy `</TD>` tag jelzi.

A táblázat celláiban szöveget, képet és egyéb objektumokat helyezhetünk el. Egy cella akár további táblázatokat is tartalmazhat.

A táblázat egyes soraiban különbözhet a cellák (azaz az oszlopok) száma. A továbbiakban csak soronként azonos számú cellával rendelkező táblázatokat használunk.

## A táblázatobjektum tulajdonságai

A már ismert *align* tulajdonság a táblázatokra is alkalmazható. Meghatározzuk vele a táblázat igazítását az ablakban (*left*: balra, *center*: középre, *right*: jobbra). Ha nem szerepel a nyitó tagban, akkor a táblázat balra igazítva jelenik meg.

A táblázatot a *border* (keret) tulajdonság segítségével keretezhetjük be. Értéke megadja a keret vastagságát pixelben:

```
border = 2
```

Ha nem szerepel a nyitó tagban, vagy 0-ra állítjuk, akkor nem jelenik meg a keret a táblázat körül. Megjegyezzük, hogy az üresen maradó cellákba nem törhető szóközt (*&nbsp;*) kell tennünk, különben az Internet Explorer nem jeleníti meg körülöttük a keretet.

A keretező vonal színét a *borderColor* (keretszín) tulajdonsággal adhatjuk meg az angol elnevezéseket vagy a hexadecimális kódot használva:

```
borderColor = "red"
```

A cellák közötti távolságot a *cellSpacing* tulajdonság határozza meg pixelben:

```
cellSpacing = 3
```

A cellákon belül a *cellPadding* tulajdonság szabja meg a margó méretét pixelben:

```
cellPadding = 5
```

Az egész táblázat szélességét célszerű meghatározni a *width* tulajdonság megadásával. Értéke lehet pixel vagy az ablak aktuális szélességének adott százaléka. A pixel használatánál nem kell jelezni a mértékegységet:

```
width = 400 vagy width = "50%"
```

Ha megadjuk a táblázat, illetve az egyes cellák szélességét, akkor a böngésző sokkal gyorsabban jeleníti meg a weblapot. Figyeljünk arra, hogy a táblázat szélességét nem csak a cellák, hanem a köztük lévő távolság (*cellSpacing*) és a belső margó (*cellPadding*) is befolyásolja!

A cellák szélességére vonatkozó előírások ellentmondhatnak a táblázat *width* tulajdonságának. Célszerű csak a cellaszélességet meghatározni. Ezt elegendő a táblázat első sorában megtenni, a *width* értéke a többi sorra is vonatkozik.

A *height* tulajdonsággal a táblázat magasságát szabjuk meg. A *width*-nél felsorolt mértékegységeket használjuk:

```
height = 300 vagy height = "30%"
```

Az **5–27. példa** kissé látványosabban jeleníti meg az 5–26. példa táblázatát. Az **5–28. példa** a táblázat említett tulajdonságainak változtatását mutatja be. Ebben már a szövegmezőket és a parancsgombokat is egy táblázatban helyeztük el, csak nem használtunk keretet. Ezt a módszert gyakran alkalmazzák az objektumok elrendezéséhez a weblapon.

A táblázatoknak még számos tulajdonságát megadhatjuk. A CD-melléklet Dokumentumok mappájában található Táblázatok.htm fájl összefoglalja a további lehetőségeket. A weblap bemutatja a táblázatot alkotó sorok és cellák tulajdonságait is. Itt csak az *align* tulajdonságot emeljük ki, amely a soron vagy a cellán belül meghatározza a tartalom igazítását. Figyeljük meg, hogy a cellákat össze is lehet vonni!

### Tömb megjelenítése táblázattal

Táblázatokkal már sokkal áttekinthetőbb megjelenítést tudunk biztosítani a tömb-elemeknek. Az egyes cellák tartalmát egyedi azonosítók segítségével is beírhatnánk, de ez nagyon kényelmetlen módszer lenne. Helyette az egész táblázatot a programból hozzuk létre.

Az 5–25. példában szereplő adatok táblázatos megjelenítését a következő utasítások végzik:

```

document.write("<TABLE>")      ' a táblázat nyitó tagja
For I = 0 To 3                  ' a sorokat elkészítő ciklus
    document.write("<TR>")      ' egy sor nyitó tagja
    For J = 0 To 2              ' egy soron belül a cellák létrehozása
        document.write("<TD>")  ' a cella nyitó tagja
        document.write(A(I, J)) ' a cella tartalma
        document.write("</TD>") ' a cella záró tagja
    Next J                      ' elkészült egy cella
    document.write("</TR>")     ' a sor záró tagja
Next I                          ' elkészült egy sor
document.write("</TABLE>")     ' a táblázat záró tagja

```

Figyeljük meg, hogy a külső ciklus helyezi el a TR-objektumok nyitó és záró tagját! A belső ciklus írja a kódba a cellák nyitó tagját, tartalmát, majd záró tagját.

A weblapon a táblázatnak fejléce (címsora) is van. Ezt nagyon hosszadalmas lett volna *document.write* metódusokkal megírni, ezért a fenti táblázatot előállító **5–29. példában** szkriptet a BODY-ba tettük. A TABLE nyitó tagját, fejlécét és záró tagját a HTML-kód tartalmazza, csak a többi sort és cellát készítettük el ciklusokkal.

Ha a táblázatot már a weblap megjelenítése után kell létrehozni, akkor egy DIV-objektumba tehetjük. Előtte azonban célszerű egy sztringben összeállítani a HTML-kódot. Ezt mutatja be az **5–30. példa**, ahol a teljes táblázatot a szkript utasításaival hoztuk létre.

## Tömbelemek keresése

A táblázatok alkalmazásánál gyakran előforduló feladat, hogy egy adott értékű tömbelemet kell megkeresni. Ha a tömb nem rendezett, akkor az elejétől kezdve össze kell hasonlítani az elemeket a megadott értékkel. Az összehasonlítást addig végezzük, amíg meg nem találjuk a keresett elemet, vagy végig nem érünk a tömbön. Ezért feltételezérelt ciklust használunk, az indexet tehát nekünk kell növelni. A vizsgálatot a ciklus ismétlési feltételében végezzük, a ciklusmag csak az indexet változtatja:

```

I = 0
Ciklus amíg I-edik tömbelem <> keresett érték És
    I < a tömb maximális indexe
    I = I + 1
Ciklus vége

```

Ez a megoldás azonban nem árulja el, hogy miért lett vége a ciklusnak. Megtaláltuk-e a keresett elemet, vagy nincs már több összehasonlításra váró elem. Erre nagyon egyszerűen válaszolhatunk:

```

Ha I-edik tömbelem = keresett érték Akkor
    megtaláltuk az elemet, indexe éppen I-vel egyenlő
Egyébként
    nem találtuk meg az elemet
Elágazás vége

```

Az **5–31. példában** az 5–29. példa táblázatában kereshetünk egy nevet, amit weblapon lévő szövegmezőbe írunk be. Ha megtaláltuk, megjeleníti a születési évet, ha nem, akkor hibaüzenetet ad.

Rendezett tömb esetén némileg gyorsabban célhoz juthatunk, mert a keresést csak addig kell folytatni, amíg a keresettnél kisebb elemeknél tartunk. A nagyobb elemek között biztosan nem fogjuk megtalálni. Így a ciklusfej a következőképpen módosul:

```
Ciklus amíg I-edik tömbelem < keresett érték És  
I < a tömb maximális indexe
```

A többi utasítás és a folytatás is változatlan marad.

Az eljárás alkalmazását az **5–32. példában** láthatjuk. Mivel az 5–29. példában a nevek szerint rendezett tömböt használtunk, csak az 5–31. példa ciklusfejét módosítottuk. Ne felejtjük el, hogy ha a magyar ábécének megfelelő sorrend esetén a sztringek összehasonlítását az *StrComp* függvényvel kell elvégezni! A példában erre nem volt szükség.

Az algoritmusokról szóló fejezetben gyorsabb keresési módszert is bemutatunk.

## A sorok és cellák kollektói

Bár a táblázat minden egyes cellájához rendelhetünk azonosítót, és objektumként kezelhetjük, ez meglehetősen kényelmetlen módszer a tulajdonságok, köztük az *innerText* megváltoztatására. Helyette használhatjuk a táblázatobjektum kollektióit.

Minden egyes táblázat (TABLE) objektum létrehozásakor automatikusan létrejön a sorait tartalmazó *rows* (sorok) kollektió. Az indexelés itt is 0-val kezdődik. Az *Adatok* azonosítójú táblázatobjektum 3. sorára (amelynek az indexe tehát 2) a következőképpen hivatkozhatunk:

```
Adatok.rows(2)
```

Minden egyes sor (TR) objektum rendelkezik a *cells* (cellák) kollektióval, amely az adott sor celláit tartalmazza. Az indexelés a sorokhoz hasonlóan 0-tól indul. Az előbb említett sor 4. cellájára (melynek indexe 3) való hivatkozás:

```
Adatok.rows(2).cells(3)
```

Egyszerűbb esetben (ha nem vontunk össze cellákat) a cellák indexe egyben a táblázat oszlopait is indexeli, szintén 0-tól kezdve.

Ha azonosítóval láttuk el a sorobjektumot, akkor közvetlenül hivatkozhatunk a *cells* kollektióra:

```
CimSor.cells(3)
```

Az *all* kollektióhoz hasonlóan a *rows* és a *cells* szintén rendelkezik a *length* tulajdonsággal, amely megadja az elemek (a sorok illetve az oszlopok) számát. Mivel az indexelés 0-val kezdődik, a legnagyobb index mindkét kollektiónál  $length - 1$ .

## A sorok és cellák elérése

A kollektiók segítségével elérhetjük akár a HTML-kóddal, akár a saját szkriptjeink által létrehozott táblázatok sorait és celláit. A hivatkozáshoz célszerű azonosítóval ellátni a táblázatobjektumot.

A következő utasítással a fenti példákban említett cella tartalmát olvassuk be a *Temp* nevű változóba:

```
Temp = Adatok.rows(2).cells(3).innerText
```

Az alábbi utasítás a cellába 25-öt ír:

```
Adatok.rows(2).cells(3).innerText = 25
```

Jegyezzük meg, hogy a cellák tartalmát a böngésző mindig sztringként jeleníti meg, hiszen az *innerText* tulajdonsághoz rendeljük hozzá.

Az *innerHTML* tulajdonsággal formázott megjelenítést végezhetünk:

```
Adatok.rows(2).cells(3).innerHTML = "<B>Fontos adat</B>"
```

A *rows* és *cells* kollekciók segítségével az objektumok többi tulajdonságát is elérhetjük. A következő utasítás:

```
rows(2).align = "right"
```

jobbra igazítja a megadott sor tartalmát. Egy cella hátterét pedig így színezzük be:

```
Adatok.rows(2).cells(3).bgColor = "red"
```

## Függvényhívás a HTML-kódból

A fent említett tulajdonságokat az **5–33. példa** segítségével tanulmányozhatjuk. A *rows* és *cells* kollekciók segítségével közvetlenül módosítjuk egy cella tartalmát és egyéb tulajdonságait. A teljes hivatkozás helyett használhattunk volna a globális szkriptben deklarált objektumváltozót is, melynek az *onload* eseménykezelőben adunk értéket (a táblázat a betöltés során jön létre):

```
Dim Cella
Sub window_onload
    Set Cella = Adatok.rows(2).cells(3)
End Sub
```

A példában egy ritka, de ügyes fogást alkalmaztunk a változtatások végrehajtására. A weblapra hivatkozás-objektumokat (A) illesztettünk, amelyek egy-egy VBScript függvényhívást végeznek. (A hivatkozás-objektumokat a 2.2. fejezet végén ismertettük.) Ekkor a *href* tulajdonságban meg kell adni, hogy VBScript kódra mutat, nehogy a böngésző elkezdje keresni a fájlt:

```
<A href = "VBScript: FüggvényNév()">
```

Ebben az esetben az interpreter a megadott függvény utasításait fogja végrehajtani. Ez helyettesíti tehát az *<A>* objektum *onclick* eseménykezelőjét. Így a weblapon mintegy „menü” hozunk létre, a szokásos kiemeléssel. Ha a felhasználó az aláhúzott szövegre viszi az egérkurzort, akkor egy kéz jelenik meg fölötte, mutatva, hogy rá lehet kattintani a felíratra.

Ilyen módon csak függvényeket hívhatunk meg, eljárásokat nem. Ha a függvény rendelkezik visszatérési értékkel, akkor a böngésző megjeleníti azt egy új weblapon.

## A táblázat kollekcióinak a használata

Az **5–34. példa** programja kiírja a képernyőre az 5–29. példa táblázatát, és lehetővé teszi a cellák tartalmának a megváltoztatását. A módosításra kerülő sor és oszlop indexére, illetve az új tartalomra szövegmezőkkel kérdez rá.

A hivatkozáshoz a táblázatobjektumnak azonosítót adunk. Az új adat beolvasásához elhelyezzük a weblapon a *SorBe*, *OszlopBe*, *AdatBe* azonosítójú szövegmezőket, és a módosítást végző parancsgombot.

A parancsgomb eseménykezelőjében beolvassuk, és numerikus értékke alakítjuk a *SorBe* és *OszlopBe* szövegmezők tartalmát, mivel a kollekciók indexénél ez nem történik meg automatikusan:

```
Sor = CSng(SorBe.value)
Oszlop = CSng(OszlopBe.value)
```

Ezután átadjuk a kijelölt cellának az *AdatBe* szövegmező tartalmát:

```
Tabla.rows(Sor).cells(Oszlop).innerHTML = AdatBe.value
```

A weblap használatánál figyeljünk arra, hogy a 0. indexű sort a címsor foglalja el!

Mivel az *innerHTML* tulajdonságot módosítottuk, a szövegmezőbe HTML-tagokat is gépelhetünk:

```
<B>Vastag betűs név</B>
```

A HTML-kód összeállításánál nem adtuk meg a cellák szélességét, ezért az adatok változtatásánál a böngésző átméretezi a táblázatot. Ennek elkerülése céljából ajánlott a táblázatoknál mindig előírni a *width* tulajdonság értékét!

## Többdimenziós tömbök

Mint említettük általában egy-, két- esetleg háromdimenziós tömböket használunk. Következő példánk egy négydimenziós tömb alkalmazását mutatja be, bár valójában adatbázis-kezeléssel lenne célszerű megoldani ezt a feladatot.

Az **5–35. példa** elkészít és megjelenít egy tömböt, amely egy iskola diákjainak az egyes tantárgyakból kapott jegyeit tárolja, tantárgyanként maximum 5-öt. Az egyszerűség kedvéért csak 4 osztályt, osztályonként 5 diákot és minden diáknál ugyanazt a 7 tantárgyat kezeljük. Így is  $4 \cdot 5 \cdot 7 \cdot 5 = 700$  eleme lesz a tömbünknek!

Az adatok tárolásához négydimenziós tömböt használunk. Az első index jelentse az osztály sorszámát (0-tól 3-ig), a második az osztályban a diák sorszámát (0-tól 4-ig), a harmadik a tantárgy sorszámát (0-tól 6-ig), a negyedik az adott tantárgyból kapott jegyek sorszámát (0-tól 4-ig). A tömböt deklaráló utasítás:

```
Dim Adatok(3, 4, 6, 4)
```

Az Olvasót megkíméljük az adatok fáradtságos begépelésétől, az *onload* eseménykezelőben véletlenszerűen választott osztályzatokkal töltjük fel a tömböt. Ehhez négy egymásba ágyazott ciklusra van szükségünk:

```
For Osztaly = 0 To 3
  For Diak = 0 To 4
    For Tantargy = 0 To 6
      For JegySorszam = 0 To 4
        Adatok(Osztaly, Diak, Tantargy, JegySorszam) = _
          Int(5 * Rnd() + 1)
      Next
    Next
  Next
Next
```

A listázást csak több táblázat segítségével tudjuk elvégezni. Minden egyes táblázat egy-egy osztály adatait jeleníti meg. Az egyes sorok felelnek meg a diákoknak, az oszlopok a tantárgyaknak. Egy tantárgy jegyeit egy-egy cellába zsúfoljuk be (különben diákonként lenne szükség egy-egy táblázatra).

Az egyszerűség kedvéért az osztályokat, a diákokat és a tantárgyakat is indexükkel jelöljük a weblapon.

A feladatot felbontjuk a táblázatok, egy táblázaton belül a sorok, egy soron belül a cellák összeállítására. Minden egyes részfeladatot egy-egy eljárással oldunk meg. Az *onload* eseménykezelőbe kerül a táblázatok létrehozó ciklus, melynek pszeudokódja:

```
Ciklus az összes osztályra
  A táblázatot összeállító eljárás meghívása
Ciklus vége
```

Egy táblázat létrehozásának a folyamata a *Tablázat* eljárásban:

```
Eljárás ' Egy táblázat létrehozása
  A táblázat felirata (az osztály indexe)
  A táblázat nyitó tagja
  ' A címsor elkészítése (a cellákban a tantárgyak indexei):
  A címsor nyitó tagja és az első (üres) cella
  Ciklus az összes tantárgyra
    A tantárgy indexét tartalmazó cella beillesztése
  Ciklus vége
  Az első sor záró tagja
  Ciklus az összes diákra
    A sort összeállító eljárás meghívása
  Ciklus vége
  A táblázat záró tagja
Eljárás vége
```

Egy sor összeállítása egy diák jegyeinek megjelenítésével a *Sor* eljárásban:

```
Eljárás ' Egy sor összeállítása
  A sor nyitó tagja
  Az első cella a diák indexével
  Ciklus az összes tantárgyra (cellára)
    Egy cellát megjelenítő eljárás meghívása
  Ciklus vége
  A sor záró tagja
Eljárás vége
```

Végül a *Cella* eljárásban egy-egy cellát kitöltünk a diák jegyeivel az adott tantárgyból:

```
Eljárás ' Egy cella megjelenítése
  A cella nyitó tagja
  Ciklus az összes jegyre
    Egy jegy kiírása
  Ciklus vége
  A cella záró tagja
Eljárás vége
```

A pszeudokód alapján már könnyű megírni a VBScript programot. Javasoljuk az Olvasónak, hogy alaposan nézze át a CD-n lévő példát! Az osztályok, diákok, tantárgyak, jegyek aktuális indexeit globális változóként deklaráltuk, hogy ne kelljen ugyanazokat az értékeket átadni az egyes eljárások hívásánál.

## 5.4. Dinamikus tömbök

A memóriaigény rugalmas kezelését dinamikus tömbök használatával érhetjük el. A dinamikus tömb méretét (és dimenziószámát) megváltoztathatjuk a program futása során, sőt, ha már nincs rá szükségünk, fel is szabadíthatjuk a helyét. A dinamikus tömbökkel szemben az eddig tárgyalt típust statikus tömbnek nevezzük.

Statikus tömb: mérete és dimenziószáma a deklarálás után nem változtatható.  
Dinamikus tömb: mérete és dimenziószáma a program futása során megváltoztatható. Helye fel is szabadítható a memóriában.

### Dinamikus tömbök létrehozása

Dinamikus tömböt úgy hozunk létre, hogy a deklarációban nem adjuk meg a maximális index értékét, csak üres zárójeleket írunk:

```
Dim Tomb()
```

A tömb elemeinek ekkor még nem adhatunk értéket, mert hibaüzenetet kapunk. Az értékadás előtt meg kell határozni a tömb méretét úgy, hogy a *ReDim* utasítással megadjuk a maximális index értékét (az indexelés itt is 0-val kezdődik):

```
ReDim Tomb(MaximálisIndex)
```

A statikus tömbök deklarációjával szemben a *ReDim* utasításban a zárójelek között tetszőleges kifejezés állhat, amely nem negatív számot eredményez (**5–36. példa**):

```
ReDim Tomb(InputBox("Írja be a maximális index értékét!", Cím, 0))
```

A fenti utasítás a felhasználótól kéri be a maximális index értékét, majd méretezi a tömböt. Ha a kifejezés törtszám, akkor az interpreter egészsre kerekít. Nem értelmezhető értékek esetén hibaüzenetet kapunk. A CD-n lévő példa a dimenzionálás után az *UBound* függvény segítségével kiírja a maximális index értékét, véletlenszámokkal feltölti a tömböt, és kilistázza az elemeket.



Dinamikus tömbök esetén a dimenziószámot is változtathatjuk (**5–37. példa**):

```
Dim Tomb()  
ReDim Tomb(2, 4)  
window.alert("Két dimenzió" & vbNewLine & _  
             "1. index maximum: " & UBound(Tomb, 1) & _  
             vbNewLine & "2. index maximum: " & UBound(Tomb, 2))  
ReDim Tomb(5)  
window.alert("Egy dimenzió, max. index: " & UBound(Tomb))
```

Ha fel akarjuk szabadítani a tömb elemei által elfoglalt memóriát, akkor -1-et adjunk meg paraméternek:

```
ReDim Tomb(-1)
```

Az 5–37. példa ezt az utasítást is tartalmazza. Figyeljük meg, hogy ebben az esetben az *UBound* függvény értéke szintén -1 lesz! Ez az érték egyben azt is jelzi, hogy a tömböt csak újradimenzionálás után tudjuk felhasználni adatok tárolására.

Mint láttuk, programjainkban tetszőleges számú *ReDim* utasítást alkalmazhatunk.

A tömböt teljes mértékben az *Erase* utasítással törölhetjük a memóriából, de utána az *UBound* hibát fog jelezni:

```
Erase Tomb
```

Az *Erase* utasításban a tömb neve után nem szabad kitenni a zárójelet.

### Az adatok megőrzése

A *ReDim* utasítás használatakor a tömbelemek elvesztik értéküket!

Ezt a következményt *Preserve* (megőrzés) kulcsszó használatával kerülhetjük el:

```
ReDim Preserve Tömbnév(új_maximális_index)
```

Az **5–38. példában** egy dinamikus tömböt feltöltünk egész véletlenszámokkal, majd különböző újradimenzionálások után kiíratjuk a képernyőre a tartalmát.

Először létrehozuk és dimenzionáljuk a tömböt:

```
Dim Tomb()  
ReDim Tomb(3)
```

Majd egy ciklus segítségével feltöltjük 1 és 10 közé eső véletlenszámokkal:

```
For I = 0 To 3  
    Tomb(I) = Int(10 * Rnd() + 1)  
Next
```

A megjelenítést külön szubrutinnal végezzük.

Az újradimenzionálásnál bővítjük a tömb méretét, de megőrizzük az elemek eddigi értékét:

```
ReDim Preserve Tomb(6)
```

Ha most újra kiíratjuk az elemeket, akkor láthatjuk, hogy az eredeti értékek megmaradtak, az új elemek pedig nem kaptak értéket.

Ha az újradimenzionálást a *Preserve* kulcsszó nélkül végezzük el:

```
ReDim Tomb(6)
```

akkor – bár itt a méret nem is változott – az eredeti értékek törlődtek, üres tömböt kaptunk.

A tömb méretének csökkentése természetesen mindig értékvesztéssel jár. Ha egy 20 elemet tartalmazó tömb méretét 15-re csökkentjük, akkor az utolsó 5 elem értéke még a *Preserve* használata esetén is elvész.

Többdimenziós tömböknél az értékek megőrzésekor csak az utolsó dimenzió maximális indexe változtatható:

```
Dim Tomb
ReDim Tomb(3, 4, 5)
...
ReDim Preserve Tomb(3, 4, 10)
```

Ha másik indexhatárt akarunk módosítani, például:

```
ReDim Preserve Tomb(3, 10, 5)
```

akkor hibaüzenetet kapunk. A *Preserve* alkalmazása esetén nem változtathatjuk meg a dimenziószámot sem.

### Dinamikus tömbök használata

A dinamikus tömbök méretét az adatbevitel során az igényeknek megfelelően folyamatosan módosíthatjuk. Az **5–39. példában** pontosan akkora tömböt használunk, amekkora a beolvasott adatok tárolásához szükséges.

Az adatokat egy *AdatBe* azonosítójú szövegmezőbe lehet beírni. A tárolást a *Gomb* azonosítójú parancsgomb *onclick* eseménykezelője végzi.

A deklarációval dinamikus tömböt hozunk létre. Indexének maximális értékét a *MaxIndex* változóban tároljuk, melynek kezdetben -1-et adunk értékül:

```
Dim Tomb()
MaxIndex = -1
```

Az *onclick* eseménykezelőben minden egyes adat beolvasásánál megnöveljük a *MaxIndex* értékét, átméretezzük a tömböt, és tároljuk benne a szövegmező tartalmát. Ne feledkezzünk meg a *Preserve* használatáról:

```
MaxIndex = MaxIndex + 1
ReDim Preserve Tomb(MaxIndex)
Tomb(MaxIndex) = AdatBe.value
```

Ezzel a módszerrel elértük, hogy a tömb mindig pontosan akkora méretű, hogy tárolni tudja az adatokat. A CD-n lévő példa minden egyes beolvasás után kiírja a tömb aktuális méretét, és listázza az elemeit.

A dinamikus tömbök használatának előnyei mellett vannak hátrányai is. Az újradimenzionálás nagy munkát ró az interpreterre. A bővítéssel járó újabb helyfoglalás az elemek megőrzése mellett rontja a memória-kezelés hatékonyságát, ami lassítja a program futását. Ennek részleteit nem tárgyaljuk.

## Rendezett adatbevitel

Adatok megjelenítése esetén gyakran előfordul, hogy rendezve szeretnénk látni a listát. Nem túl sok adat esetén azt is megtehetjük, hogy a bevitelnél keressük meg az új elem helyét. Ha innen kezdve továbbléptetjük a tömbelemeket, akkor felszabadul egy hely az új elem számára. Így eleve rendezett tömböt kapunk.

A fent vázolt rendezési módszer nem a leghatékonyabb. Sok összehasonlítást és adatmozgatást igényel, nagy mennyiségű adat esetén az újabb elem helyének megkeresése és felszabadítása jelentős várakozási idővel jár. A 9. fejezetben másfajta adattárolási lehetőséggel is megismerkedünk, amely jóval egyszerűbbé teszi a rendezett adatbevitelt.

A rendezett bevitel bemutatásához az **5–40. példában** személyek nevét és születési évét tároljuk az *Adatok* kétdimenziós dinamikus tömbben, a nevek szerint rendezve. Emlékeztetünk arra, hogy a sztringek összehasonlításánál az ábécében előbb álló karakterek sorozat a kisebb. A tömb méretét minden egyes beolvasásnál megnöveljük. Így mindig éppen akkora lesz, amekkora az adatok tárolásához szükséges.

Sajnos a dinamikus tömböknél csak az utolsó index maximális értéke változtatható, a többi nem. Ezért egy személy adatait egy oszlopban tároljuk:

<b>Név:</b>	Andor Péter	Kovács István	...
<b>Születési év:</b>	1963	1971	...

Ez az elrendezés a megjelenítést nem befolyásolja, ott felcserélhetjük a sorokat és oszlopokat.

A programot az előző példához hasonló módon kezdjük, deklarálunk egy dinamikus tömböt és egy *MaxIndex* nevű változót, melynek -1 kezdőértéket adunk:

```
Dim MaxIndex, Adatok()  
MaxIndex = -1
```

A tömb első indexe a sorokat, második indexe az oszlopokat jelöli. Az *I* indexű személy esetén tehát az *Adatok(0, I)* tartalmazza a nevet, az *Adatok(1, I)* pedig a születési évet.

Ha a felhasználó beírta a *NevBe* és a *SzuletesiEvBe* szövegmezőkbe a személy adatait, akkor rákattinthat a Beolvas feliratú parancsgombra. A parancsgomb *onclick* eseménykezelőjében megkeressük a személy helyét a tömbben, majd végrehajtjuk az adatok tárolását és megjelenítését.

Először átírjuk a nevet egy változóba, megnöveljük a *MaxIndex* értékét, és újradimenzionáljuk a tömböt. Csak az oszlopok számát kell megváltoztatni, a sorok száma mindig 2 marad (az első index maximális értéke tehát 1):

```
Nev = NevBe.value  
MaxIndex = MaxIndex + 1  
ReDim Preserve Adatok(1, MaxIndex)
```

Az újradimenzionálás miatt a táblázat végén keletkezik egy üres oszlop.

Most a név alapján meg kell keresnünk az ábécérendnek megfelelő helyet, amit a már ismert algoritmussal végzünk. Az összehasonlításhoz az *StrComp* függvényt hasz-

náljuk, hogy a magyar ábécének megfelelő sorrendet kapjuk meg. A keresett indexet a *Hely* nevű változóban tároljuk:

```
Hely = 0
Do While StrComp(Adatok(0, Hely), Nev, 1) = -1 And _
    Hely < MaxIndex
    Hely = Hely + 1
Loop
```

Ha a név nagyobb, mint a vele összehasonlított név, vagy elértük a tömb végét, akkor a *Hely* változó éppen az új személy indexét jelzi a táblázatban.

A tároláshoz helyet kell készítenünk a személy adatainak, azaz a *Hely* által mutatott indexű elemtől kezdve minden elemet egygel feljebb kell léptetni a tömbben. Emelkeztetünk arra, hogy a tömb utolsó helye az újradimenzionálás miatt üres:

```
For I = MaxIndex - 1 To Hely Step -1
    Adatok(0, I + 1) = Adatok(0, I) ' a név áthelyezése
    Adatok(1, I + 1) = Adatok(1, I) ' a születési év áthelyezése
Next
```

Már csak a felszabaduló helyre kell beírni az új személy adatait, és készen is vagyunk:

```
Adatok(0, Hely) = Nev
Adatok(1, Hely) = SzuletesiEvBe.value
NevBe.focus()
NevBe.select()
```

A CD-n lévő példa a tárolás után a már ismert módon táblázatos formában megjeleníti az adatokat. Figyeljünk arra, hogy a táblázat első oszlopában a tömb első (sor-) indexének értéke 0, a második oszlopban pedig 1!

### Ütköző alkalmazása

Az előző példában szereplő *Do ... Loop* ciklus ismétlési feltételében megvizsgáltuk, hogy a *Hely* változó kisebb-e, mint a tömb maximális indexének értéke. Ezt meg kellett tennünk, mert a növelés után az ismétlésnél túlléptük volna az indexhatárt.

A feltétel azonban csak akkor nem teljesül, ha elértük a tömb végét. Egy nagyméretű tömb esetén ez nagyon sok felesleges vizsgálatot jelent. A feltételek és a vizsgálatok számának csökkentését egy gyakori fogással, az úgynevezett ütköző alkalmazásával érhetjük el.

Amikor a tömbben létrehozunk egy új oszlopot, akkor elemei még üresek. Így a beolvasott név a nagyobb, tehát a ciklusban az interpreter megnövelné a *Hely* változó értékét. A tömbnek azonban nincs több oszlopa, az ismétlési feltétel következő vizsgálatánál hibaüzenetet kapnánk. Ezért kellett az ismétlési feltételben megnéznünk azt is, hogy elértük-e már a maximális indexet. Ha a tömb legutolsó elemének helyére be tesszük az éppen beolvasott nevet, akkor az utolsó összehasonlítás le fogja állítani az ismétléseket anélkül, hogy figyelniénk a maximális index elérésére (a *Nev* nem nagyobb saját magánál). Ezzel nem csak a végrehajtási időt csökkentjük, hanem a kódot is egyszerűbbé tesszük.

A tömb végére elhelyezett speciális értéket azért nevezzük ütközőnek, mert külön vizsgálat nélkül is leállítja a ciklus futását.

Ütköző használatához az 5–40. példa kódját csak nagyon kis mértékben kell módosítani. A tömb újradimenzionálása után beírjuk a beolvasott nevet a legnagyobb indexű elembe, és töröljük az indexhatár ellenőrzését a *Do* ciklus ismétlési feltételéből (**5–41. példa**):

```
ReDim Preserve Adatok(1, MaxIndex)
' az ütköző elhelyezése:
Adatok(0, MaxIndex) = Nev
Hely = 0
' egyszerűbb ismétlési feltétel:
Do While StrComp(Adatok(0, Hely), Nev, 1) = -1
    Hely = Hely + 1
Loop
```

### Sorok és cellák beillesztése a táblázatba

Az előző példában minden egyes adatbevitel után újrameztük a táblázat HTML-kódjának összeállítását. Ezzel fölöslegesen dolgoztattuk az interpretert, ugyanis a böngésző képes arra, hogy egy már elkészített táblázatba illesszen be újabb sorokat és cellákat. Ehhez a *TABLE*-objektum *insertRow* (sor beillesztése) és a *TR*-objektum *insertCell* (cella beillesztése) metódusait használhatjuk fel. Először be kell illesztenünk egy sort, majd abba a cellákat.

Az *insertRow* metódus paramétere kijelöli az új sor helyét:

```
TáblaNév.insertRow(index)
```

A sorok indexe (számozása) 0-tól kezdődik. A következő utasítás az *Eredmenyek* nevű táblázat elejére illeszt be egy új sort:

```
Eredmenyek.insertRow(0)
```

Ha paraméterek nélkül hívjuk meg a metódust, akkor az új sor a táblázat végére kerül. Érvénytelen paraméter esetén természetesen hibaüzenetet kapunk.

Az *insertCell* metódus paramétere hasonló szerepet játszik, csak a cella helyének indexét jelöli ki (a számozás itt is 0-tól kezdődik):

```
SorAzonosító.insertCell(index)
```

Ha paraméterek nélkül hívjuk meg a metódust, akkor az új cella a sor végére kerül. Érvénytelen paraméter esetén itt is hibaüzenetet kapunk.

Mivel a sorokat ritkán látjuk el külön azonosítóval, a cellák beillesztésénél gyakran a *rows* kollekciót használjuk:

```
TáblaNév.rows(index).insertCell(index)
```

Emlékezzünk vissza arra, hogy az üres cella keretét az Internet Explorer nem rajzolja ki. Csak akkor jelenik meg a keret, ha valamit írunk a cellába (például nem törhető szóközt).

Az **5–42. példában** a metódusok használatának bemutatására az 5–27. példa web-lapjára beillesztünk két szövegmezőt és egy parancsgombot. A két szövegmezőbe le-

het beírni az új sor indexét és a cellák számát, a parancsgomb eseménykezelője pedig elvégzi a beillesztéseket. A beillesztés után az új cellákba véletlenszámokat írunk.

A hivatkozáshoz a táblázatot a *Tablázat* azonosítóval láttuk el, és kissé egyszerűsítettük a formáját.

Az eseménykezelőben először tároljuk a szövegmezők tartalmát. A sorindexet aktuális paraméterként használjuk, ezért át kell alakítanunk numerikus értéké:

```
Sor = CSng(SorIndex.value)
Cella = CellaSzam.value
```

Beillesztjük a táblázatba az új sort:

```
Tablázat.InsertRow(Sor)
```

Végül egy ciklussal beillesztjük az új cellákat, és beírjuk a véletlenszámokat. Mivel a cellák indexelése is 0-tól kezdődik, ezért a ciklusváltozó értékének felső határa egygyel kisebb, mint a beillesztendő cellák száma:

```
For I = 0 To Cella - 1
    Tablázat.Rows(Sor).InsertCell(I)
    Tablázat.Rows(Sor).Cells(I).InnerText = _
        Int(100 * Rnd() + 1)
Next
```

Figyeljük meg, hogy egy sorba tetszőleges számú cellát beilleszthetünk, legfeljebb növekedni fog az oszlopok száma! A programot csak a metódusok bemutatására készítettük, ezért nem foglalkozunk a bevitt adatok ellenőrzésével.

### Példa a cellák beillesztésére

Az **5–43. példában** úgy alakítjuk át az 5–41. példa kódját, hogy a már elkészült táblázatba illesztjük be az új adatokat. Ehhez a megjelenítést szolgáló bekezdésobjektum helyett egy *Tablázat* azonosítójú TABLE-objektumot helyezünk a weblapra, és elkészítjük a címsort is.

A megjelenítéshez a táblázat HTML-kódját tartalmazó *Lista* sztring összeállítása helyett a *Tablázat*-objektumba illesztjük be az új sorokat és cellákat. A sor indexe a táblázat címsora miatt egygyel nagyobb lesz, mint a tárolásnál meghatározott *Hely* változó értéke. Ennek felhasználásával illesztjük be az új sort:

```
SorIndex = Hely + 1
Tablázat.InsertRow(SorIndex)
```

Az új sor elkészítése után egy ciklussal elkészítjük a két cellát, és beírjuk az adatokat:

```
For I = 0 To 1
    Tablázat.Rows(SorIndex).InsertCell(I)
    Tablázat.Rows(SorIndex).Cells(I).InnerText = _
        Adatok(I, Hely)
Next
```

Zavaró lehet, hogy a táblázat címsora akkor is megjelenik, amikor még nem tartalmaz adatokat. Ennek kiküszöböléséhez a **5–44. példában** a táblázat *visibility* tulajdon-

ságát *hidden* értékre állítjuk. A *visibility*-t csak a *style* tulajdonságon keresztül érhetjük el:

```
<TABLE id = "Tablazat" ... style = "visibility: hidden">
```

A parancsgomb eseménykezelőjében pedig visszaállítjuk a *visible* értékre:

```
Tablazat.style.visibility = "visible"
```

Igaz, hogy az interpreter ezt az utasítást minden egyes alkalommal végrehajtja, amikor adatokat írunk be, de ez teljesen elhanyagolható mértékben növeli meg a program futási idejét. Felesleges lenne egy *If* utasítással megvizsgálni, hogy már látható-e a táblázat (vagy vannak-e adatok az *Adatok* tömbben), mert ugyanerre az eredményre jutnánk, ráadásul a szelekció miatt bonyolultabb lenne a kód, és hosszabb lenne a végrehajtási idő.

### Sorok és cellák törlése a táblázatból

A sorokat és a cellákat törölhetjük is a táblázatból. Egy cella törlése nem csak a tartalom törlését jelenti, végrehajtása esetén a sorban utána elhelyezkedő cellák eggyel balra lépnek.

A törlést a *deleteRow* és *deleteCell* metódusok végzik, a beillesztéshez hasonló paraméterezéssel:

```
TáblaAzonosító.deleteRow(index), illetve  
SorAzonosító.deleteCell(index)
```

Az **5–45. példa** kiegészíti az 5–42. kódját a két metódus alkalmazásával. Ehhez megjelenítünk két újabb szövegmezőt, melyekbe a törlendő sor, illetve cella indexét írhatjuk. A hozzájuk tartozó parancsgomb eseménykezelője végzi el a törlést.

Ha megadjuk a cella indexét, akkor az így meghatározott cellát töröljük, egyébként a sort fogjuk törölni:

```
If Cella <> "" Then  
    Tablazat.rows(Sor).deleteCell(CSng(Cella))  
Else  
    Tablazat.deleteRow(Sor)  
End If
```

### A sorok és cellák száma

A HTML-kód táblázatainak módosításakor szükségünk lehet a sorok, egy soron belül pedig a cellák számának ismeretére. Az adatokat a *rows* és *cells* kollekciók *length* (hossz, méret) tulajdonsága adja meg:

```
TáblázatAzonosító.rows.length, SorAzonosító.cells.length
```

Az **5–46. példa** kiegészíti az 5–45. példa kódját a *length* tulajdonság megjelenítésével. Ehhez két SPAN-objektumot helyezünk el a weblapon, és az elsőben közvetlenül megjelenítjük a sorok számát:

```
SorokSzama.innerText = Tablazat.rows.length
```

A másodikba egy ciklus segítségével írjuk be soronként a cellák számát. A listát először egy sztringben állítjuk össze:

```
Lista = ""
For I = 0 To Tablázat.rows.length - 1
    Lista = Lista & I & ". sor: " & _
        Tablázat.rows(I).cells.length & " cella<BR>"
Next
```

A fenti utasításokat az összes eseménykezelőben el kellene helyezni, ezért eljárást készítettünk belőlük, amit a betöltésnél és a módosítások után mindig meghívunk.

### Példa a sorok törlésére

Az **5–47. példában** kiegészítjük az 5–44. példa kódját a már bevitt adatok törlésének lehetőségével. Ehhez a weblapon elhelyezünk egy Töröl gombot.

A parancsgomb eseménykezelőjében először megvizsgáljuk, hogy van-e egyáltalán adat az *Adatok* tömbben. Ha nincsen, akkor figyelmeztetjük a felhasználót, ráállítjuk a fókuszot a név szövegmezőre, és kilépünk az eljárásból. Az ellenőrzéshez a *MaxIndex* változót használjuk:

```
If MaxIndex = -1 Then
    window.alert("Üres a lista!")
    NevBe.focus()
    NevBe.select()
    Exit Sub
End If
```

Ha már vittek be előzőleg adatokat, akkor a bevitelnél használt algoritmussal megkeressük a név helyét. A hely indexét most is a *Hely* változóban tároljuk. Mivel itt nem alkalmazhatunk ütközöt (nincs hely a tömb végén), az ismétlési feltételben meg kell vizsgálnunk, hogy nem értünk-e a tömb végére:

```
Hely = 0
Do While StrComp(Adatok(0, Hely), Nev, 1) = -1 And _
    Hely < MaxIndex
    Hely = Hely + 1
Loop
```

A keresés után meg kell vizsgálnunk, hogy megtaláltuk-e a szövegmezőbe írt nevet a listában. Ha nem, hibaüzenetet küldünk a felhasználónak, kijelöljük a beírt nevet, és kilépünk az eseménykezelőből:

```
If Nev <> Adatok(0, Hely) Then
    window.alert("Nincs ilyen név a listában.")
    NevBe.focus()
    NevBe.select()
    Exit Sub
End If
```

Ha megtaláltuk a nevet, akkor töröljük a megjelenített táblázatból a sort. A táblázat címsora miatt a *Hely* változó eggyel kisebb értéket mutat, mint a sor indexe:

```
Tablázat.deleteRow(Hely + 1)
```



### *Törlés a tömbből*

Az előzőekben csak a megjelenített táblázatból töröltük a sort, az adatokat tartalmazó tömbből még nem. Ehhez a tömb elemeit a `Hely + 1` indextől kezdve a tömb végéig eggyel kisebb indexű helyre léptetjük. Így a *Hely* változó által mutatott indexű elem törlődik:

```
For I = Hely + 1 To MaxIndex
    Adatok(0, I - 1) = Adatok(0, I)
    Adatok(1, I - 1) = Adatok(1, I)
Next
```

Eggyel kevesebb adatunk van, ezért csökkentjük a *MaxIndex* értékét:

```
MaxIndex = MaxIndex - 1
```

A tömb utolsó elemére már nincs szükségünk, ezért csökkentjük a méretet:

```
ReDim Preserve Adatok(1, MaxIndex)
```

Ha a *MaxIndex* értéke -1 lett (kiürült a tömb), akkor a táblázat *visibility* tulajdonságát *hidden* értékűre állítjuk, végül a következő név begépeléséhez kijelöljük a szövegmező tartalmát.

Mivel megengedtük egyforma nevek bevitelét, – a keresési algoritmusnak megfelelően – ezek közül az elsőt fogjuk kitörölni.

Üres tömb esetén hibaüzenetet küldtünk a felhasználónak, ha a Törlés gombra kattint. Elegánsabb megoldást kapunk, ha az adatok beolvasása előtt letiltjuk a parancsgomb működését (**5–48. példa**):

```
Torol.disabled = True
```

A tiltást el kell helyezni az objektum nyitó tagjában (betöltéskor még üres a lista), és a törlés eseménykezelőjének a végén, ha a *MaxIndex* értéke -1 lett:

```
If MaxIndex = -1 Then
    Torol.diasbled = True
End If
```

A működést a beolvasás eseménykezelőjében engedélyezzük, a táblázat láthatóvá tételével együtt:

```
Torol.disabled = False
```

Így a törlés eseménykezelőjét csak létező adatok esetén hívhatjuk meg, tehát nincs szükségünk az elején a *MaxIndex* vizsgálatára. Ezzel egyszerűsítettük a program szerkezetét.

Vigyázzunk arra, hogy az ilyen jellegű megjelenítésnél elváljak egymástól az adatokat tároló tömb és a weblapra kikerülő táblázat! Így valamilyen programhiba következtében könnyen megtörténhet, hogy azok nincsenek egymással összhangban. Ha a táblázatból töröljük az egyik sort, a tömbből pedig egy másikat, vagy egyiket sem, akkor már egészen mást látunk, mint amit tárolunk. Ezért a tárolás és megjelenítés elkülönítését csak kellő gondossággal szabad alkalmazni!

Megjegyezzük, hogy a gyakori bővítés és törlés alaposan leterheli az interpretert. Nagyméretű tömbök esetén le is lassítja az adatbevitelt, és érzékelhető várakozási időt okoz. Ezt a problémát a későbbiekben ismertetésre kerülő lista adatszerkezettel lehet kiküszöbölni.

## 5.5. Tömbök alkalmazása

Ha az Olvasó idáig eljutott a könyv tanulmányozásában, akkor megismerkedett a programok legfontosabb összetevőivel: a változókkal, a kifejezésekkel, az értékadó utasításokkal, a relációkkal, a szelekcióval és iterációval, a függvényekkel és eljárásokkal, végül pedig a tömbökkel.

Az eddigi ismeretek alkalmazását egy nagyobb példán keresztül mutatjuk be. Javasoljuk az Olvasónak, hogy figyelmesen tanulmányozza a forráskódot, próbálja megérteni minden egyes sorát, minden egyes utasítását! Ha túl bonyolultnak és hosszadalmasnak találja, akkor ki is hagyhatja ezt a részt.

### A feladat megfogalmazása

Programunk diákok év végi eredményeit tárolja, illetve értékeli. Először bekéri a diákok nevét, továbbá néhány tantárgyból az év végi jegyeket. Kiszámítja a jegyek átlagát diákonként és tantárgyanként, meghatározza az osztályátlagot, a kitűnő és a bukott tanulók számát. A listában a neveket ábécé szerint rendezve írja ki:

Név:	Magyar:	Történelem:	Angol:	Matematika:	Ének:	Átlag:
András Ilona						
Kovács Gábor						
Vass Géza						
Átlag:						

A táblázat után felsorolja a kitűnő és a bukott tanulók nevét.

Eddigi ismereteinknek megfelelően megvizsgáljuk a bevitelt, az osztályzatok helyére csak 1-től 5-ig fogadunk el számjegyeket. Osztályzatot egyetlen tantárgynál sem kötelező megadni (lehet, hogy a neveket viszik fel előbb, még jegyek nélkül), de nem engedjük meg a név üresen hagyását.

Nem engedjük meg azt sem, hogy több tanuló is ugyanazon a néven szerepeljen a listában. Természetesen eltekinthetnénk ettől a feltételtől, de ez megnehezítené a későbbi keresést, módosítást, törlést. Az adatok tárolásánál általános elvárás, hogy az egyes személyeknek egyedi azonosítójuk legyen. Erre a célra nem a legmegfelelőbb a név használata, de mi most más adatot nem tárolunk. Az azonos nevű diákokat ezért külön jelöléssel kell ellátni (például egy számmal a név után).

Az egyszerűség kedvéért csak 5 tantárgyat és legfeljebb 10 diákot kezelünk. A programot azonban úgy készítjük el, hogy mindkét érték könnyen módosítható legyen.

A feladat pontos megfogalmazása nagy mértékben segíti a program írását. Az utólagos változtatások végrehajtása, újabb funkciók beépítése aránytalanul nagy munká-

val jár. Általában sok helyen kell módosítani a programot, az egyes változtatások egymásra is hatással vannak. Gyakran jobb, ha előről kezdjük az egész tervezést.

Ha például mégis úgy döntünk, hogy minden tantárgyból kötelező osztályzatot megadni, akkor sokkal egyszerűbb lesz az átlagolás, főlegesen számoljuk a jegyeket. A begépelt osztályzatok érvényességét vizsgáló eljárást is módosítani kell, hogy ne engedje meg a rovat üresen hagyását. Így egyre több helyen változtatunk a már megírt programon, ami további hibákhoz vezethet.

A programot több lépésben fogjuk elkészíteni. Minden egyes lépés forráskódja szerepel a CD-n. A teljes megoldást az 5–54. példa mutatja be.

Megjegyezzük, hogy programunk nem tudja elmenteni az adatokat, kilépéskor tehát törlődnek. A háttértárak kezelésével a későbbiekben ismerkedünk meg, akkor majd gondoskodni tudunk az adatok megőrzéséről. A példához hasonló adatbázis-kezelési feladatokra egyébként nem célszerű egyedi megoldásokat adni, itt csak az eddigi anyag gyakorlása a cél.

## A program terve

A program részletes megtervezése legalább olyan fontos része a sikeres megírásnak, mint a feladat precíz, minden részletre kiterjedő megfogalmazása. A programtervezés a programozási ismeretek külön fejezetét képezi, itt csak az alapjait tekinthetjük át ennek az összetett folyamatnak.

A lépésenkénti finomítás elvét fogjuk követni, először nagy vonalakban fogalmazzuk meg a tennivalókat, majd minden egyes pontot egyre részletesebben megtervezünk, amíg el nem jutunk egészen a forráskódig.

Készülő dokumentumunk alapvetően két részből áll, a deklarációkat, függvényeket eljárásokat tartalmazó szkriptből és a HTML-kódból. A HTML-kódban található mindazon elemek, amelyek állandónak tekinthetők a weblapon (magyarázó szöveg, parancsgombok). A szkript tartalmazza:

- a globális változók deklarációit és inicializálását,
- az eseménykezelő eljárásokat és az általuk felhasznált függvényeket.

A kitűzött feladat alapján el kell készítenünk a következő eljárásokat:

- a beolvasást végző parancsgomb *onclick* eseménykezelőjét,
- a megjelenítést végző parancsgomb *onclick* eseménykezelőjét,
- a *window onload* eseménykezelőjét a betöltéskor végrehajtandó utasításokkal.

## Az adatok tárolása

Az algoritmus önmagában nem sokat ér, ha nem támogatja egy hatékonyan megtervezett adatszerkezet. Úgy is fogalmazhatnánk, hogy:

$$\text{program} = \text{algoritmus} + \text{adatszerkezet}$$

Vannak olyan programtervezési módszerek, amelyek eleve az adatszerkezetből vezetik le az algoritmust.

Mivel adataink táblázatos formába írhatók, tárolásukra egy kétdimenziós dinamikus tömböt fogunk használni. Már a rendezett tárolás bemutatásánál is láttuk, hogy a dinamikus tömbnek csak az oszlopszáma növelhető. Ezért a tárolásnál az oszlopok tartalmazzák a diákokat, a sorok pedig az egyes tantárgyak eredményeit:

	Oszlopindex:	0	1	...
<b>Név:</b>	Sorindex: 0	András Ilona	Kovács Gábor	...
<b>Magyar:</b>	1			
<b>Történelem:</b>	2			
<b>Angol:</b>	3			
<b>Matematika:</b>	4			
<b>Ének:</b>	5			

A táblázatot mindvégig így fogjuk kezelni, a kiírásnál viszont egyszerűen felcseréljük a sorokat és oszlopokat.

A fenti ábrán feltüntettük a sor- és oszlopindexeket is. Láthatjuk, hogy a tanuló nevét a 0 indexű sorba írtuk, így a tantárgyak száma éppen a maximális sorindexszel egyezik meg. Az egyszerű bővíthetőség miatt ezt a program elején konstansként deklaráljuk, módosítás esetén csak át kell írni az értékét:

```
Const TantargySzam = 5
```

A tanulókat az oszlopindex jelöli 0-tól kezdve, melynek legnagyobb értéke tehát eggyel kisebb, mint a tanulók száma. Ezt is konstansként adjuk meg, melynek deklarációja legfeljebb 10 tanuló esetén:

```
Const TanuloMaxIndex = 9
```

Az adatok beolvasásánál figyelni fogjuk, hogy az oszlopindex ne lépje túl ezt az értéket. Ennek csak memória-takarékossági okai vannak. A konstans átírásával bármikor megnövelhetjük a tanulók lehetséges számát.

Deklarálunk egy tömböt, amely a tantárgyak nevét tartalmazza. Komolyabb alkalmazásoknál ezt célszerű valamilyen háttértárra elmenteni, majd onnan betölteni. Ennek módját a későbbiekben ismerjük meg. Egyelőre a program elején töltjük fel a tantárgyneveket. Megkönnyítjük a táblázat megjelenítését, ha utolsó elemébe beírjuk az „Átlag” sztringet. Így maximális indexének értéke megegyezik a tantárgyak számával:

```
Dim Tantargy(5)
Tantargy(0) = "Magyar" : ... : Tantargy(5) = "Átlag:"
```

A feltöltéshez használhattuk volna az *Array* függvényt is (leírását lásd az 5.1. fejezetben).

Ha a programot bővíteni akarjuk, akkor csupán az eddig deklarált konstansokat és a *Tantargy* tömb inicializálását kell megváltoztatnunk. A későbbiekben sehol nem alkalmazunk olyan kódot, amely felhasználná, hogy legfeljebb hány tantárgy és hány diák kezelését engedélyeztük!

Hátra van még természetesen az adatokat tartalmazó dinamikus tömb deklarálása:

```
Dim Adatok()
```

Globális változókból csak keveset használunk. Nyilván kell tartanunk, hogy hány tanuló adatait olvastuk be. Az oszlopok száma helyett kényelmesebb lesz a maximális index értékét tárolni, melyet először -1-re állítunk. Ez az érték jelöli, hogy még egyetlen beolvasás sem történt:

```
Dim MaxIndex  
MaxIndex = -1
```

A *MaxIndex* értékét minden egyes diák adatainak a tárolása előtt eggyel meg fogjuk növelni.

A sorok és oszlopok kezelését végző ciklusok ciklusváltozóit szintén globálisként deklaráljuk:

```
Dim Sor, Oszlop
```

Az *Oszlop* változóra szükségünk lesz a HTML-kódba illesztett szkriptekben, és felhasználjuk az eseménykezelő eljárásokban is.

Feladatunk első pontjával, a globális változók deklarációjával és inicializálásával végeztünk.

## A dokumentum HTML-kódja

A HTML-kódba írjuk a magyarázó szöveget, a bevitelhez szükséges táblázatot, a parancsgombokat és az adatok megjelenítéséhez használt DIV-objektumot.

A szöveget, a parancsgombokat, illetve a táblázatobjektum nyitó és záró tagját az eddig megszokott módon helyezzük el a kódban. Az inputobjektumokat (feliratok, szövegmezők) tartalmazó táblázat celláinak elkészítését azonban egy-egy beillesztett szkriptre bízuk, hiszen a *Tantargyak* tömb már tartalmazza a tantárgyak nevét, így fölösleges lenne ezeket külön-külön leírni. Olyan kódot szeretnénk készíteni, amely könnyen bővíthető. Ha egy ciklusra bízuk a HTML-kód összeállítását, akkor a tantárgyak számának növelésével is megfelelő dokumentumhoz jutunk. A szkriptek utasításait tehetjük volna a *window onload* eseménykezelőjébe, de így egyszerűbb lesz a programunk, hiszen az állandó részeket elhelyezhetjük a HTML-kódban, nem kell a *document.write* metódust használnunk.

A TABLE-elemen belül a két sor nyitó és záró tagját a HTML-kódba írjuk. Az első sorba kerülnek a tantárgynevek, a másodikba pedig a bevitelhez szükséges szövegmezők. Mindkét sor első celláját külön írjuk be a HTML-kódba, mert más formátumot (szélesség, szövegmező jellemzők) szeretnénk beállítani:

```
<TD>Név:</TD>
```

A tantárgyak nevét tartalmazó cellákat azonban már egy egyszerű ciklussal illesztjük be (a tantárgyak indexe 0-tól indul és *Tantargyszam-1*-ig tart):

```
<SCRIPT>
  For Oszlop = 0 To TantargySzam - 1
    document.write("<TD width = 80>")
    document.write(Tantargy(Oszlop))
    document.write("</TD>")
  Next
</SCRIPT>
```

A második sor szövegmezőinek elkészítését is hasonló módon végezzük. A jegyeket tartalmazó cellák szélességét már az előző ciklus 80 pixelre állította. A szövegmezők méretére és maximális hosszára 1 karaktert írunk elő. Az adatok beolvasásához a szövegmező-objektumoknak a *Bevitel* azonosítót adjuk, majd indexeléssel különböztetjük meg őket egymástól. A ciklusmagot annyiszor kell megismételni, ahány tantárgy van:

```
<TD><INPUT id = "Bevitel" type = "text"></TD>
<SCRIPT>
  For Oszlop = 1 To TantargySzam
    document.write("<TD>")
    document.write("<INPUT id = 'Bevitel' type = 'text'")
    document.write(" size = 1 maxLength = 1>")
    document.write("</TD>")
  Next
</SCRIPT>
```

A sztringek összefűzésével egyetlen *document.write* is elegendő lett volna.

Az eddig elmondottaknak megfelelő kódot az **5–49. példa** tartalmazza, amelyben az adatok tárolását a Tárolás, megjelenítését az Értékelés parancsgomb *onclick* eseménykezelője végzi. Ezek természetesen még nem működnek, és a kivitt szolgáló üres DIV-objektumot sem látjuk, ha a böngészővel betöltjük a fájlt.

Ezzel a dokumentum HTML-kódját elkészítettük.

### Az onload eseménykezelő

Az eseménykezelőtől azt várjuk, hogy állítsa a fókuszt az első (tehát 0 indexű) szövegmezőre:

```
Bevitel(0).focus()
```

Az 5–49. példa kódja tartalmazza az *onload* eseménykezelőt is.

Megemlíjtük, hogy a Windows-ban a tabulátor billentyűvel is lehet lépkedni a szövegmezők között (visszafelé a Shift+Tab-bal haladhatunk). Az utolsó tantárgy szövegmezőjénél lenyomott tabulátor pedig éppen a Tárolás gombra állítja a fókuszt, így elegendő lenyomni az Enter billentyűt. Nagy mennyiségű adat bevitelénél az egér és a billentyűzet felváltva történő használata lassítja a munkát.

### Az adatbevitel ellenőrzése

Ha a felhasználó a „Tárol” gombra kattint, akkor két feladatot kell elvégeznünk:

- ellenőrizzük a begépelt adatok érvényességét,
- tároljuk az érvényes adatokat.

Az adatok érvényességének ellenőrzésekor minden előforduló hibalehetőségre végrehajtjuk az alábbi szelekciót:

```
Ha hibás_az_adat akkor
    hibaüzenet a felhasználónak
    kilépés az eseménykezelőből
Elágazás vége
```

Mivel hibás adat esetén nem tároljuk az adatokat, fölösleges az interpretert (és a program készítőjét) *ElseIf* utasításokkal terhelni, melyek feldolgozása több adminisztrációval jár. Egyszerűbb, ha kilépünk az eseménykezelőből.

A bevitelnél a következő hibák fordulhatnak elő:

- elértük a diákok megengedett maximális létszámát,
- nem töltötték ki a „Név” rovatot,
- olyan nevet írtak be, amely már szerepel a listában,
- nem 1 és 5 közé eső értéket írtak be valamelyik osztályzat helyére.

A begépelte adatok ellenőrzésének sorrendje a program szempontjából lényegtelen, mégis célszerű átgondolni. Főlegesen zaklatjuk a felhasználót, ha például minden rosszul begépelte jegyet kijavítottunk vele, utána közöljük, hogy úgysem írhat be több tanulót, mert betelt a hely.

A diákok létszámát pontosan a fenti pszeudokódnak megfelelően ellenőrizzük. Az aktuális indexet a *MaxIndex*, az engedélyezett legnagyobb értéket a *TanuloMaxIndex* értéke jelzi, így a feltétel:

```
If MaxIndex = TanuloMaxIndex Then
```

Ekkor hibaüzenetet adunk és kilépünk az eljárásból. A felhasználó munkájának megkönnyítésére az „Értékelés” gombra állítjuk a fókuszot. Ezzel ezt az ellenőrzést elvégeztük:

```
window.alert("Nem tárolható több tanuló adata!" & vbNewLine & _
    "Kattintson az Értékelés gombra!")
Ertekel.focus()
Exit Sub
End If
```

A begépelte nevet a *Bevitel(0)* szövegmező tartalmazza. Mivel többször is szükségünk lesz rá, gyorsítjuk és egyszerűsítjük a munkát, ha először egy változóban tároljuk:

```
Nev = Bevitel(0).value
```

Ha a *Nev* = "", akkor az előző feltételes elágazás mintájára hibaüzenetet adunk, és kilépünk az eljárásból. A kilépés előtt most a szövegmezőre célszerű állítani a fókuszot, mert feltehetően ennek kitöltésével folytatódik a munka.

Az eddig elkészült kódot az **5–50. példa** tartalmazza. Ha név beírása nélkül kattintunk a bevitel gombra, akkor hibaüzenetet kapunk.

Most következik annak ellenőrzése, hogy van-e már ilyen név a listában. A keresés külön algoritmus végrehajtását igényli, ezért egy függvénnyel végezzük el a feladatot. Ha az adott név még nem szerepel a listában, akkor *Keres* függvényünk -1-et ad visz-

sza, egyébként pedig a már szereplő név indexét. Erre a program bővítésénél lesz szükségünk, amikor a törlést és a bevitt adatok módosítását is lehetővé tesszük. Most egyszerűen beírjuk a függvényhívást az ellenőrzést végző kódba. A függvénynek aktuális paraméterként megadjuk a keresett nevet:

```
If Keres(Nev) <> -1 Then
```

A folytatás ugyanaz, mint eddig: hibaüzenet és kilépés az eljárásból. Mivel most szerepel név a szövegmezőben, a kilépés előtt kiválasztjuk, hogy könnyebb legyen a javítás.

Utolsóként a begépelte jegyeket ellenőrizzük. Mivel a vizsgálat több lépésből áll, ismét rábízzuk egy függvényre. A *Hiba* függvény -1-et ad vissza, ha minden jegy elfogadható (tehát 1 és 5 közé esik), egyébként pedig megadja a hibás értéket tartalmazó szövegmező indexét. A függvénynek nincs szüksége paraméterekre. Mivel értékét többször is fel fogjuk használni, a hívást külön utasítás tartalmazza:

```
HibaHely = Hiba()
```

Ha a függvény nem -1-et adott vissza, akkor valamelyik jegy hibás. Ekkor hibaüzenetet adunk, és a kilépés előtt kijelöljük a hibás jegyet. Ezzel nagymértékben segítjük a felhasználó munkáját. A hibás jegyet tartalmazó szövegmező indexét éppen a *Hiba* függvény adta meg, tehát a *HibaHely* nevű változó tartalmazza:

```
Bevitel(HibaHely).select()
```

Ezzel az összes általunk számításba vett hibalehetőség ellenőrzését elvégeztük.

A hibaellenőrzések során felhasznált változókat természetesen az alprogramunk elején deklarálni kell.

### A Keres függvény kódja

Mielőtt az ellenőrzések kódolására és az adatok tárolására rátérnénk, készítsük el a két hiányzó függvényt.

A *Keres* függvény paramétere a keresett karaktersorozat:

```
Keres(KeresettNev)
```

A függvény végignézi a tömb azon elemeit, melyek első indexe 0 (itt tároljuk a neveket). Ha megtalálja benne a paraméterként megadott karaktersorozatot, akkor visszaadja annak indexét. Ha nem, akkor a visszatérési érték -1 lesz.

Gondolnunk kell arra, hogy a dinamikus tömb esetleg még nem is tartalmaz adatokat. Ekkor természetesen nem találhatja meg a keresett nevet, így a visszatérési érték szintén -1 lesz. Ezért először ezt adjuk visszatérési értéknek:

```
Keres = -1
```

Ezután a *MaxIndex* segítségével megvizsgáljuk, hogy van-e eleme a tömbnek. Ha igen, akkor a rendezett tömbre vonatkozó, már ismert algoritmussal elvégezzük a keresést:



```
If MaxIndex <> -1 Then
    Hely = 0
    Do While StrComp(Adatok(0, Hely), KeresettNev, 1) = -1 And _
        Hely < MaxIndex
        Hely = Hely + 1
    Loop
```

A függvényérték meghatározásához tudnunk kell, hogy a ciklus megtalálta-e a keresett nevet. Ha igen, átírjuk a visszatérési értéket a hely indexére:

```
If Adatok(0, Hely) = KeresettNev Then
    Keres = Hely
End If
End If
```

Vegyük észre, hogy ha még nincs eleme a tömbnek (a *MaxIndex* értéke -1), akkor a keresést nem hajtjuk végre, így a függvényérték marad -1!

Ezzel a függvényt elkészítettük, teljes kódját az **5–51. példa** tartalmazza.

### A Hiba függvény kódja

A bevitt adatok ellenőrzésénél szükségünk lesz még egy olyan függvényre, amely megvizsgálja a begépelt jegyeket. Ha értékük 1 és 5 közé esik, akkor -1-et ad vissza, hibás jegy esetén pedig a szövegmező indexét a *Bevitel* kollekcióban. Mivel eléggé a programunkhoz kötődő feladatot oldunk meg, közvetlenül hivatkozunk objektumainkra és változóinkra, ezért a függvény nem használ paramétereket.

A rossz karaktert tartalmazó szövegmező indexét a *Rossz* változóban tároljuk, melynek először -1-et adunk értékül:

```
Rossz = -1
```

A vizsgálatok után ez a változó a függvény visszatérési értékét fogja tartalmazni.

Függvényünk egy ciklussal végignézi a jegyeket tartalmazó szövegmezők értékét, melyek indexe 1-től *TantargySzam*-ig tart. Ha hibát talál, akkor nem is folytatja tovább a vizsgálatot. Mivel *Do ... Loop* ciklust alkalmazunk, ezért nekünk kell gondoskodnunk a léptetésről, amit a *Tantargy* nevű változóval oldunk meg:

```
Tantargy = 0
Do
    Tantargy = Tantargy + 1
```

A ciklusban beolvasunk egy jegyet, és megvizsgáljuk, hogy megfelel-e a feltételeknek. Ne felejtsük el, hogy a szövegmezők értéke sztring, és megengedtük a rovat üresen hagyását is. Ha hibás értéket találunk, akkor a *Rossz* változóban feljegyezzük a szövegmező indexét:

```
Jegy = Bevitel(Tantargy).value
If Jegy <> "" And (Jegy < "1" Or Jegy > "5") Then
    Rossz = Tantargy
End If
```

A ciklust addig kell futtatni, amíg nem találtunk hibát, vagy nem néztük végig az összes jegyet:

```
Loop While Rossz = -1 And Tantargy < TantargySzam
```

Ezek után már csak a függvény visszatérési értékét kell megadni:

```
Hiba = Rossz
```

Ezzel a bevitt ellenőrző mindkét függvényt elkészítettük. A teljes kódot az 5–51. példa tartalmazza. Próbáljuk ki a hibás adatbevittet, és figyeljük meg a program működését! Van-e olyan hiba, amire nem reagál? Mivel az adatokat még nem tároltuk, ezért „megengedi” hogy többször is megadjuk ugyanazt a nevet. Ennek a hibának a kiszűrése csak az adatok tárolása után fog működni.

### Az adatok tárolása

Elérkeztünk az eseménykezelő eljárás legfontosabb részéhez, ahol az ellenőrzött adatokat beírjuk a tömbbe. A beillesztés módszerével már az ütköző használatának bemutatásánál megismerkedtünk. Teljes egészében az ott szereplő algoritmust követjük:

- megnöveljük a *MaxIndex* értékét,
- újradimenzionáljuk az *Adatok* tömböt a már beírt értékek megőrzésével,
- ütközőként beírjuk az új nevet a tömb végére,
- megkeressük az új név alapján a diák helyét, hogy rendezett maradjon a tömb,
- a tömb eddigi elemeinek léptetésével helyet szabadítunk fel az új adatok számára,
- egy ciklussal beírjuk a felszabadított helyre a szövegmezők tartalmát,
- az újabb diák nevének begépeléséhez kiválasztjuk a nevet tartalmazó szövegmezőt.

Mivel a rendezett tárolásnál és az ütköző használatánál bemutatott lépéseket alkalmazuk, reméljük, az Olvasó maga is el tudja készíteni a kódot. Az egyetlen változás, hogy itt több adatot kezelünk, ezért a mozgathoz és a beíráshoz ciklust használunk.

Az eddig megírt forráskódot az **5–52. példa** mutatja be. Mivel a keresés algoritmus egyszerű, és ütköző alkalmazásával fel is gyorsítható, nem hívtuk meg a *Keres* függvényt. Megjegyezzük, hogy a szövegmezőkre a *Bevitel* kollekció segítségével hivatkozunk.

A weblap betöltése után kipróbálhatjuk a diákok nevének ellenőrzését. Nem fogja megengedni, hogy ugyanazon a néven még egyszer adatokat tároljunk.

### Az adatok megjelenítése

Szeretnénk látni munkánk eredményét, vagyis táblázatosan megjeleníteni a begépet adatokat. Ezt a feladatot az *Ertekel* parancsgomb *onclick* eseménykezelője végzi el. Ne felejtsük el, hogy a megjelenítéshez fel kell cserélnünk az *Adatok* tömb sorait és oszlopait! A továbbiakban a *Sor* és *Oszlop* indexek a tömbre fognak vonatkozni.

A táblázatot először a *Tabla* sztringben állítjuk össze, és csak a végén adjuk át a megjelenítést végző DIV-objektumnak. Erre már láttunk példát az előzőekben.

A táblázat összeállítása közben diákonként elvégezzük a jegyek átlagolását, valamint a kitűnő és bukott diákok listájának az elkészítését. Ehhez deklaráljuk, és inicializáljuk a *Kituno* illetve a *Bukott* sztringeket:

Kituno = ""  
Bukott = ""

A táblázat elkészítésekor a szükséges HTML-kódot sorra hozzá kell fűzni a *Tabla* sztringhez, kiegészítve az *Adatok* tömb megfelelő elemeivel. Az összeállítás menetét az **5–53. példa** kódjába illesztett megjegyzések segítségével követhetjük.

1. Elhelyezzük a TABLE-objektum nyitó tagját.
2. Beillesztjük az első sor TR-objektumának nyitó tagját, és a tartalmát középre igazítjuk.
3. Külön helyezzük el az első cellát a „Név” felirattal, mert szélesebb lesz, mint a többi.
4. A többi cellát egy ciklussal illesztjük be, amely 0-tól *TantargySzam*-ig fut. Az utolsó lépés az „Átlag” feliratot írja be a cellába.
5. Lezárjuk az első sort.

A táblázat első sorát a törzsbe illesztett szkriptben is elkészíthetnénk. Ekkor az adatok listázásáig a megjelenítést célszerű letiltani.

6. A táblázat többi sorát egy nagy ciklus segítségével illesztjük be. Minden egyes sor egy-egy diák adatait tartalmazza.

A diákok adatait megjelenítő ciklus a következő lépésekből épül fel:

7. Beillesztjük a sor nyitó tagját.
8. Balra igazítva beillesztjük az `Adatok(0, Oszlop)` elemet, azaz a diák nevét tartalmazó cellát (az *align* tulajdonság megadása miatt a sor nyitó tagjában előírt középre igazítás ebben a cellában nem érvényesül).
9. A sor többi celláját a jegyekkel egy ciklus illeszti be. Ezekre a sorra előírt, középre igazítás lesz érvényes.
10. A ciklusban először beillesztjük a TD-objektum nyitó tagját.
11. Beírjuk a tantárgyi jegyet. Ha az adott tantárgyból nincsen jegy, akkor egy nem törhető szóközt kell írunk a cellába, mert különben az Internet Explorer nem rajzolja ki a cella határvonalát.
12. Beillesztjük a TD-objektum záró tagját. Ezzel egy diák jegyeinek a megjelenítését végző ciklust befejeztük.
13. A sor végén az utolsó cellába a diák átlaga kerül. Ezt egy függvény számítja ki, amelyet külön meg kell írunk. A függvény az átlagot formázott sztringként adja vissza.
14. Ha a diák átlaga „1,00”, akkor a nevét a bukott tanulók listájához, ha pedig „5,00” akkor a kitűnőkhöz fűzzük hozzá, elválasztójellel kiegészítve.
15. Beírjuk az átlagot az utolsó cellába.
16. A TR-objektum záró tagjának beillesztésével befejeztük a sor elkészítését.

A fenti ciklust az összes diákra lefuttatjuk. Hátra van még a táblázat utolsó sorának, a tantárgyi átlagoknak a megjelenítése.

17. Beillesztjük az első cellát a középre igazított „Átlag” felirattal.

18. A tantárgyi átlagokat egy ciklus segítségével jelenítjük meg. A számítást a *TantargyAtlag* függvény végzi.
19. A táblázat utolsó cellájába az összes jegy átlagát írjuk, amit az *OsszesAtlag* függvény számol ki.
20. A TR-objektum záró tagjának elhelyezésével befejezzük az utolsó sor megjelenítését.
21. A TABLE-objektum záró tagjának beillesztésével elkészült a táblázat.

A táblázat után megjelenítjük a kitűnő és a bukott tanulók névsorát, amit már a diákok átlagának meghatározásánál összeállítottunk.

22. Üres lista esetén kötőjelet írunk ki.
23. A kitűnő és a bukott tanulók névsorát hozzáfűzzük a *Tabla* sztringhez.
24. A DIV-objektum *innerHTML* tulajdonságának segítségével megjelenítjük a sztringet.
25. Tartalmának kijelölésével a fókusz a név szövegmezőre állítjuk, hogy lehetővé tegyük a következő diák adatainak a bevitelét.

Ezzel a megjelenítést beprogramoztuk. A kódot az 5–53. példa tartalmazza. Mivel az átlagszámítást még nem kódoltuk, a szükséges függvényeket ugyan elhelyeztük benne, de egyszerűen csak egy kérdőjelet adnak vissza. Ezzel programírás közben a függvények és eljárások kódjának megírása nem vonja el a figyelmünket, de megelőzzük, hogy hívásuk futási hibát okozzon.

Bár a kód egy kissé hosszadalmas, ennek csak az esztétikus megjelenítésre való törekvés az oka. A sorok és oszlopok kiírását végző cikluson és egy-két egyszerű feltételes elágazáson kívül nem tartalmazott semmilyen bonyolult szerkezetet.

Ezt az algoritmust kell alkalmaznunk minden olyan esetben, amikor egy kétdimenziós táblázat weblapon történő megjelenítése a cél.

### A tantárgyi átlagok számítása

A *TantargyAtlag* függvény aktuális paraméterként megkapja annak a tantárgynak a sorszámát (egyben indexét), melynek átlagát számolni akarjuk:

```
TantargyAtlag(Sorszam)
```

A függvény visszatérési értéke a két tizedesre megformázott tantárgyi átlag lesz. Ha az adott tantárgyból egyáltalán nincsen jegy, akkor egy "-" jelet fog visszaadni:

```
TantargyAtlag = "-"
```

Mivel a jegyek száma kevesebb is lehet, mint a diákok száma, ezért azt egy külön változóban tartjuk nyilván, 0 kezdőértékkel:

```
JegyekSzama = 0
```

Az összegezést a már ismert módon végezzük, de meg kell vizsgálnunk, hogy az adott tantárgyból van-e jegy. Ha igen, akkor hozzáadjuk az összeghez, és növeljük a jegyek számát:

```
Osszeg = 0
For I = 0 To MaxIndex
    If Adatok(Sorszam, I) <> "" Then
        Osszeg = Osszeg + Adatok(Sorszam, I)
        JegyekSzama = JegyekSzama + 1
    End If
Next
```

A felhasználó akkor is rákattinthat az Értékelés gombra, amikor még egyáltalán nem vitt be adatokat. Ekkor azonban az előlesztelő *For* ciklus nem fut le, tehát nem kapunk hibaüzenetet.

A visszatérési érték az összeg és a jegyek számának hányadosa:

```
If JegyekSzama > 0 Then
    TantargyAtlag = FormatNumber(Osszeg / JegyekSzama, 2)
End If
```

A függvény kódját az **5–54. példa** tartalmazza.

### Az összes jegy átlaga

A táblázat utolsó cellájába a diákok összesített eredménye kerül. Itt nem egyszerűen az átlagokat átlagoljuk, hanem az összes jegy átlagát számítjuk ki. A függvény nem rendelkezik paraméterrel:

```
OsszesAtlag()
```

A visszatérési értékre a tantárgyi átlagokat számító függvénynél elmondottak érvényesek. Az algoritmus is ugyanaz, csak itt két, egymásba ágyazott ciklust használunk. A külső lépteti a tantárgyakat, a belső pedig a diákokat:

```
JegyekSzama = 0
Osszeg = 0
For I = 1 To TantargySzam
    For J = 0 To MaxIndex
        If Adatok(I, J) <> "" Then
            Osszeg = Osszeg + Adatok(I, J)
            JegyekSzama = JegyekSzama + 1
        End If
    Next
Next
```

Végül a visszatérési értéket is a tantárgyi átlagokhoz hasonlóan határozzuk meg, figyelembe véve, hogy egyáltalán volt-e beírva jegy.

A függvény kódját az **5–54. példa** tartalmazza.

### A diákok átlagának meghatározása

Utoljára hagytuk a diákok jegyeinek átlagát számító *DiakAtlag* függvényt, amely a diák indexét kapja paraméterként:

```
DiakAtlag(DiakIndex)
```

Az eddigiektől abban térünk el, hogy figyelniünk kell a jegyeket. Amint előfordul közöttük 1-es, a diák megbukott, átlaga is 1-es lesz. Ezért az összegezést nem kell tovább folytatnunk. Ehhez bevezetünk egy *Bukott* logikai változót, melynek *False* kezdőértéket adunk. A *For ... Next* ciklus helyett *Do ... Loop* ciklust használunk. Amint találunk 1-est a jegyek között, a *Bukott* értékét átírjuk *True*-ra, egyébként pedig a szokásos módon végezzük az összegezést, csak most az indexet is nekünk kell növelnünk:

```
DiakAtlag = "-"
JegyekSzama = 0
Osszeg = 0
I = 0
Bukott = False
Do
    I = I + 1
    Jegy = Adatok(I, DiakIndex)
    If Jegy = "1" Then
        Bukott = True
    ElseIf Jegy <> "" Then
        Osszeg = Osszeg + Jegy
        JegyekSzama = JegyekSzama + 1
    End If
```

A ciklust addig ismétljük, amíg nem találtunk 1-es jegyet vagy össze nem adtuk az összes tantárgy jegyeit:

```
Loop While Not Bukott And I < TantargySzam
```

A függvény visszatérési értékét az előzőekhez hasonlóan adjuk meg, figyelembe véve, hogy bukott tanuló esetén "1,00" legyen. Először a *Bukott* változó értékét kell megvizsgálni, mert ha igaz, akkor jegye is van a diáknak:

```
If Bukott Then
    DiakAtlag = "1,00"
ElseIf JegyekSzama > 0 Then
    DiakAtlag = FormatNumber(Osszeg / JegyekSzama, 2)
End If
```

Ha egyik feltétel sem teljesül, akkor marad az eljárás elején beállított "-"

Ezzel kitűzött célunkat elértük, megírtuk programot. A teljesen kész forráskódot az 5–54. példa tartalmazza. A függelékben található feladatok néhány további funkcióval (a már bevitt adatok módosítása, törlése) bővítik a választékot.

Javasoljuk az Olvasónak, hogy különböző adatok bevitelével futtassa a programot! Figyelje az eredményeket, próbáljon meg különböző hibákat elkövetni. Ismételten tanácsoljuk, hogy minden részletében gondolja át a forráskódot. A könyv eddigi anyagát csak akkor sikerült maradéktalanul elsajátítania, ha nem talál érthetetlen részt benne.

## **II. PROGRAMOZÁS HALADÓKNAK**





## 6. VÁLTOZÓK ÉS KIFEJEZÉSEK

A programozás alapelemeinek elsajátítása után részletesen áttekintjük a VBScript eszközeit, lehetőségeit. Először a változókkal kapcsolatos ismereteinket bővítjük.

### 6.1. A változók altípusai

A programozási nyelvek kétféle változót különböztetnek meg. Az egyszerű típus-hoz azok tartoznak, amelyek csak egyetlen értéket tudnak tárolni. Ilyenek például a numerikus változók, a karaktersorozatok és a logikai változók. Az összetett típusú változók egyszerre több adat tárolására alkalmasak. Közülük eddig a tömbökkel és a kollekciókkal ismerkedtünk meg.

#### A Variant típus

Az egyszerű típusok felosztása főleg a típusos programnyelvek jellemzője, melyeknél egy változó deklarálásakor kötelező megadni a típusát is. A VBScriptben erre nincs lehetőség, az interpreter egy általános, úgynevezett Variant típust rendel a változónévhez.

A Variant típusú változó különféle adatok tárolására használható. A program futása során felváltva tartalmazhat numerikus értéket, sztringet vagy más típusú adatot. Ez nagy rugalmasságot biztosít, de veszélyekkel is jár. Könnyebben követhetünk el hibát a program írásánál, az interpreter nem figyelmeztet, hogy véletlenül összekevertünk két különböző típusú értéket.

A típusok keverésének másik hátránya abból adódik, hogy a különböző (például egész és tört) értékeket az interpreter eltérő módszerrel kódolja. A kifejezések kiértékelésénél viszont azonos típusra kell az operandusokat átalakítani. Ez a konvertálás lassítja a program futását. Konverziós függvények alkalmazásával kikerülhetjük az automatikus átalakítást. A *CSng* konverziós függvényt már használtuk az előzőekben, hamarosan a többivel is megismerkedünk.

#### A Variant altípusai

Bár a változók deklarációjánál nem lehet megadni, a VBScript megkülönböztet egymástól néhány típust, amit a Variant altípusainak nevezünk. A különböző altípusok eltérő kódolással kerülnek a memóriába. A kódolás miatt eltérőek azok a tartományok, amelyekbe az egyes értékek eshetnek.

Numerikus adatok esetén figyelembe kell venni, hogy csak egész számokat, vagy törtet is tárolnunk kell. Ez utóbbi esetben valós (lebegőpontos) típusról beszélünk. Valós típusú változó természetesen tartalmazhat egész értéket, de az egész típusúnak nem lehet törtrésze. Egész típusú például a darabszám. Valós típusú például valamilyen távolság, amely 5 méter, 25,7 méter stb.

A következő táblázatban összefoglaltuk a VBScript altípusait és a hozzájuk tartozó tartományokat.

Altípus:	Megnevezés:	Tartomány:
Boolean	logikai	True (-1) vagy False (0)
String	karakterlánc	legfeljebb kétmilliárd karakter
Date	dátum	100. január 1. és 9999. december 31. között
Time	idő	0:00:00-tól 23:59:59-ig
Object	objektum-hivatkozás	objektumra mutat
Error	hiba	hibakódot tartalmaz
<b>Egész típusú változók (csak egész számokat tudnak tárolni):</b>		
Byte	bájt	0-tól 255-ig
Integer	egész	-32 768-tól +32 767-ig
Long	hosszú egész	-2 147 483 648-tól +2 147 483 647-ig
Currency	pénznem (a 10000-szeresét tárolja)	-922 337 203 685 477,5808-tól -922 337 203 685 477,5807-ig
<b>Valós típusú változók (egész és törtszámokat is tárolhatnak):</b>		
Single	egyszeres pontosságú valós (lebegőpontos)	-3,402823E38-tól -1,401298E-45-ig a negatív és +1,401298E-45-től +3,402823E38-ig a pozitív számokra, illetve 0
Double	kétszeres pontosságú valós (lebegőpontos)	-1,79769313486232E308-tól -4,94065645841247E-324-ig a negatív és +4,94065645841247E-324-től +1,79769313486232E308-ig a pozitív számokra, illetve 0

6–1. táblázat. A VBScript változók altípusai

A bájt típust főleg bitek kezelésére használják.

A dátum típusú változók elég nagy tartományt átfognak, de a Gergely-naptár 1582-es (egyres országokban jóval később történt) bevezetése előtti dátumokra nem veszik figyelembe a naptárreform miatt fellépő eltérést.<sup>20</sup>

Az egész típusú változók esetén a memóriában elfoglalt bájtok száma határozza meg a lehetséges tartományt. Egyszerű egészeknél 2, hosszú egészeknél 4 bájtot használnak a tárolásra.

A pénznem típust az egészekhez soroltuk, mivel a memóriában a 10000-szeresük tárolódik. A számításoknál azonban ezt nem kell figyelembe venni, négy tizedesjegy használható az értékadásnál.

A valós típusú változóknál a lebegőpontos elnevezés a kódolás módjára utal. Külön tárolódik a kitevő, és külön a hatvány szorzótényezője, amit mantisszának nevezünk.

<sup>20</sup> A pápa rendeletére 1582. október 4. után 15-ét írtak, és a szökőévek gyakoriságát is módosították.

A számtartományt a felhasznált bájtok száma korlátozza. A tárolás hasonlít a zsebszámológép kijelzőjéhez, ahol a rendelkezésre álló helyre beírhatunk nagyon kicsi és nagyon nagy számokat is, de csak meghatározott számú számjegyet használhatunk fel. A számítógépnél természetesen bináris kódot alkalmaznak.

A Double altípus több bájton tárolja a kitevőt és a mantisszát, mint a Single, ezért nagyobb a maximális érték és a pontosság.

A táblázatban feltüntetett, eddig nem tárgyalt típusokhoz hamarosan visszatérünk.

### Az altípusok jelölése

Mivel a VBScriptben nem adhatjuk meg a deklarációban a változók típusát, a programozók gyakran egy hárombetűs előtaggal, úgynevezett prefixummal utalnak erre a változónév előtt. Ez megkönnyíti a programok elemzését és a hibakeresést.

Altípus:	Prefixum:	Példa:
Boolean	bln	blnTalalat
Byte	byt	bytBitAdat
Date, Time	dtm	dtmStart
Double	dbl	dblNagyonPontos
Error	err	errHibaKod
Integer	int	intSzamlal
Long	lng	lngNagySzam
Object	obj	objMunkatars
Single	sng	sngKicsitPontos
String	str	strNev

6–2. táblázat. A változók altípusát jelölő prefixumok

A fenti rövidítések alkalmazása nem kötelező, akár egyéni jelölésmódot is használhatunk. A szakkönyvekben azonban gyakran találkozunk ezekkel az előtagokkal. Mi a továbbiakban csak akkor alkalmazzuk őket, ha kifejezetten hangsúlyozni akarjuk a változók altípusát.

### Konverzió az altípusok között

Az interpreter egy változó értékadásánál maga dönti el, hogy melyik altípust választja. Az érték pontos tárolására alkalmas legtömörebb ábrázolási módot használja. Kis egész számok Integer, törtrésszel is rendelkező értékek pedig Double típusúak lesznek. Ezt a típuskonverziós függvények használatával változtathatjuk meg.

Az interpreter a kifejezések kiértékelésénél automatikusan elvégzi a típuskonverziót. A kifejezésben szereplő operátorok döntenek el, hogy milyen altípust választ. A típuskonverziós függvényekkel megváltoztathatjuk az eredmény altípusát.

A típuskonverziós függvények neve C-vel kezdődik (conversion), és annak az altípusnak a rövidítésével folytatódik, amire átalakítunk. A függvények egyetlen paramétere az a kifejezés, melynek értékét átalakítjuk.

Függvény:	Visszatérési érték:
CBool	logikai
CByte	bájt
CCur	pénznem
CDate	dátum és idő
CDBl	kétszeres pontosságú valós
CInt	egész
CLng	hosszú egész
CSng	egyszeres pontosságú valós
CStr	sztring

6–3. táblázat. A VBScript konverziós függvényei

A *CBool* esetén ha a kifejezés értéke nulla, akkor a függvényérték *False*, minden egyéb numerikus értéknél pedig *True*.

Ha a logikai és numerikus értékre történő átalakításnál a kifejezés értéke nem értelmezhető valamilyen számként, akkor hibaüzenetet kapunk. Nem jön létre hibaüzenet, ha egy a területi beállításoknak megfelelő sztringet konvertálunk, amely esetleg ezres elválasztójeleket is tartalmaz.

Ha törtszámokat alakítunk át egész típusra, akkor az interpreter kerekíti az értéket.

Feltétlenül el kell végeznünk a típuskonverziót, ha egy szövegmezőből beolvasott számot hasonlítunk össze numerikus változóval. Erre eddig a *CSng* függvényt használtuk. A továbbiakban ha csak egész számok fordulhatnak elő, akkor célszerűbb a *CInt* vagy a *CLng* függvényt alkalmazni.

### Túlcsordulás

A konverziónál megtörténhet, hogy a szám nincs benne az új típusnak megfelelő tartományban, ezt a hibát túlcsordulásnak nevezzük.

Túlcsordulás: a numerikus érték kívül esik az értéktartományon.

Ha például a változó értéke 70000, akkor ezt az interpreter automatikusan a Long altípushoz sorolja. A **6–1. példában** a `CInt(70000)` hatására hibaüzenetet kapunk:

```
Hosszu = 70000
Egesz = CInt(Hosszu)
```

Akkor is túlcsordulás következhet be, ha az átalakítandó értéket egy kifejezés hozza létre:

```
CInt(30000 + 30000)
```

Túlcsordulással nem csak típuskonverziónál találkozhatunk. Ha túllépjük a Double altípus számábrázolási tartományát, akkor szintén hibaüzenetet kapunk:

```
NagyonNagySzam = 1E308 + 1E308
```

Túlcsordulás akkor is létrejön, ha negatív irányban lépünk ki a számtartományból:

```
NagyonNegativSzam = -1E308 - 1E308
```

Az interpreter a kifejezések kiértékelésénél automatikusan azt az altípust választja, amelyben az eredmény tárolható. Ha két nagy egész számot adunk össze, ami már nem férne be a Long altípus tartományába, akkor a Double altípusra tér át. Így a következő értékadó utasítás végrehajtásánál nem jön létre túlcsordulás:

```
Dupla = 2000000000 + 2000000000
```

A túlcsordulásra feltétlenül gondolnunk kell a programok írásánál, mert keletkezésekor hibaüzenetet kapunk. Esetenként az operandusok ügyes csoportosításával elkerülhetjük a túlcsordulást. A

```
9E307 + 9E307 - 2E307
```

kifejezés értéke belefér a kétszeres pontosságú számtartományba. Kiértékelése közben azonban az összeadásnál túlcsordulás jön létre. Ha megváltoztatjuk a sorrendet:

```
9E307 - 2E307 + 9E307
```

akkor nem kapunk hibaüzenetet, mert egyetlen részeredmény sem lépi túl az értéktartományt.

## A TypeName függvény

A VBScript rendelkezésünkre bocsát egy függvényt, amellyel lekérdezhetjük egy változó altípusának a nevét. A *TypeName* (típusnév) paramétere az a változónév vagy kifejezés, amelynek az altípusára kíváncsiak vagyunk:

```
TypeName (Kifejezés)
```

A függvény visszatérési értéke a 6–1. táblázatban megadott altípus neve. Az idő típusú változóknál a Date, tömbök esetén pedig a Variant(), megjelölést adja vissza.

A *TypeName* segítségével megtudhatjuk, hogy a VBScript milyen altípust választ egy változó értékének a tárolásánál:

```
TypeName (True), TypeName (4567) vagy TypeName (6.2)
```

A CD-melléklet **6–2. példájában** különböző értékeket konvertáltunk, és alkalmaztuk a *TypeName* függvény is. Javasoljuk az Olvasónak, hogy alaposan gondolja át a forráskódot az eredménnyel együtt!

## A változók alapértelmezett értéke

Ha egy változó a deklaráció után még nem kapott értéket, akkor egy kifejezésben az interpreter automatikusan a következő értékeket használja:

<i>Az operátor típusa:</i>	<i>A változó értéke:</i>
aritmetikai	0
sztringek összefűzése	"" (üres sztring)
logikai	False (= 0)

Így ha az *Ures* változónak a deklarálás után nem adunk értéket, akkor a következő eredményeket kapjuk:

<code>Ures * 1 = 0</code>	<code>Ures + 1 = 1</code>	
<code>Ures &amp; "" = ""</code>	<code>"a" &amp; Ures &amp; "b" = "ab"</code>	
<code>Not Ures = -1</code>	<code>True And Ures = 0</code>	<code>False Or Ures = 0</code>

A logikai műveletekkel kapcsolatban emlékeztetünk arra, hogy a *True* -1-nek, a *False* pedig 0-nak felel meg.

Az interpreter a relációs operátorok esetén is a fenti alapértelmezett értékeket használja. Az alábbi relációk mind igazak:

```
Ures = Ures
Ures < 5, Ures > -3, Ures = 0, mert ekkor Ures = 0
Ures = "", Ures < "a", mert ekkor Ures = ""
Ures = False, Ures <> True, mert ekkor Ures = 0
```

Az alapértelmezett értékek kényelmessé teszik a programírást, ám sok hibalehetőség rejtenez magukban. Megtörténhet, hogy elfeledkezünk az inicializálásról, de ezt az interpreter nem jelzi. A legtöbb programozási nyelv megköveteli az értékadást a változó felhasználása előtt.

## A változók Empty értéke

Az alapértelmezett érték megnehezíti az adatbevitel során fellépő hibák felfedését. A VBScript azonban fenntart egy speciális értéket annak megjelölésére, hogy a változó még nem kapott értéket.

A deklaráció és az első értékadás között a változó értéke *Empty* (üres). Az *Empty*-t felhasználhatjuk a relációkban is. Az *Empty* a VBScript kulcsszava, így nem szabad idézőjelbe tenni!

Az alapértelmezett érték miatt az *Empty*-t nem vizsgálhatjuk relációs operátorokkal, erre a célra az *IsEmpty* (üres-e) függvényt kell alkalmaznunk, mely hívásának szintaxisa:

```
IsEmpty(VáltozóNév)
```

Az *IsEmpty* visszatérési értéke *True*, ha a paramétereként megadott változó még nem kapott értéket, egyébként pedig *False*.

A következő eljárás kiírja a felhasználónak, hogy a hívás a változó első értékadása előtt vagy után történt:

```
Sub Kiir()
    If IsEmpty(Valtozo) Then
        window.alert("A változó még nem kapott értéket.")
    Else
        window.alert("A változó értéke: " & Valtozo)
    End If
End Sub
```

A CD-melléklet **6–3. példájában** deklarálunk egy változót, és a szubrutint meghívjuk az első értékadás előtt, illetve után. Figyeljük meg az üzeneteket!

Az *InputBox* kezelésénél is hasznos lehet az *Empty* vizsgálata. Ha a felhasználó meghagyta (vagy törölte) a kezdőértéket, de a Mégse vagy a Bezárás gombra kattintott, akkor a visszatérési érték *Empty* lesz. Ha törölte a beviteli mezőt, és az OK gombra kattintott, akkor már üres sztringet kapunk. Az *IsEmpty* tehát meg tudja egymástól különböztetni ezt a két esetet (**6–4. példa**):

```
Adat = InputBox("Írjon be egy adatot!", "Beolvasás", 0)
If IsEmpty(Adat) Then
    Eredmeny.innerText = "A Mégse/Bezárás gombra kattintott."
ElseIf Adat = "" Then
    Eredmeny.innerText = "Az OK gombra kattintott, de " & _
                        "üresen hagyta a szövegmezőt."
Else
    Eredmeny.innerText = "Adat = " & Adat
End If
```

Szükség esetén értékadó utasítással is *Empty*-re állíthatunk egy változót:

```
VáltozóNév = Empty
```

A *TypeName* visszatérési értéke egy értékkel nem rendelkező változó esetén szintén *Empty*.

## A változók Null értéke

A változók egy másik speciális értéket, a *Null*-t is felvehetik. A *Null*-al azt jelezzük, hogy a változó nem tartalmaz érvényes adatot. Az *Empty*-vel ellentétben egy változó csak értékadó utasítással kaphat *Null* értéket, például:

```
Változónév = Null
```

Ilyenkor a *TypeName* függvény visszatérési értéke is *Null*.

Ha egy aritmetikai kifejezésben valamelyik változó értéke *Null*, akkor az egész kifejezés *Null*. Ezt úgy szoktuk megfogalmazni, hogy a *Null* terjed a kifejezésekben:

```
Null + 3 = Null
Szam = Null esetén Szam * 2 = Null
```

A *Null*-t néhány függvény is használja az eredménnyel kapcsolatos problémák jelzésére:

```
Abs(-3 + Null) = Null
```

Sztringek összefűzésénél az eredmény csak akkor lesz *Null*, ha mindkét operandus értéke *Null*. Egyébként a *Null* értékű operandust úgy kezeli az interpreter, mintha üres sztring lenne:

```
Null & Null = Null, Null & "ab" = "ab"
```

A logikai műveleteknél ha valamelyik operandus *Null*, akkor az eredmény is *Null*, kivéve a következő eseteket:

```
False And Null = False           True Or Null = True
Null And False = False           Null Or True = True
```

Indoklásként annyit jegyzünk meg, hogy az „és” művelet akkor hamis, ha legalább az egyik operandus hamis. A „vagy” művelet pedig akkor igaz, ha legalább az egyik operandus igaz.

Összehasonlítások esetén ha bármelyik kifejezés *Null*, az eredmény is *Null* lesz (tehát sem hamis, sem igaz):

```
(3 < Null) = Null, (Null <> 5) = Null, (Null = Null) = Null
```

Egy változó *Null* értékét az *IsNull* (Null-e) függvénnyel kell megvizsgálni:

```
IsNull(VáltozóNév)
```

A következő eljárás a változó értékének megfelelő üzenetet írja ki:

```
Sub Kiir()  
    If IsNull(Valtozo) Then  
        window.alert("A változó értéke: Null")  
    Else  
        window.alert("A változó értéke: " & Valtozo)  
    End If  
End Sub
```

A CD-melléklet **6–5. példájában** először egy numerikus értéket, majd a *Null*-t rendeljük hozzá egy változóhoz. Minden egyes értékadás után meghívjuk a fenti *Kiir* eljárást. Magyarozatként a weblapon megjelenítjük az értékadó utasításokat.

## A Null használata

A *Null* értéket főleg adatbázisok kezelésénél, hiányzó adatok esetén használják, de hibajelzést is végezhetünk a segítségével. Nagy hasznát vesszük a listák készítésénél. Mint említettük, néhány függvény vagy metódus visszatérési értéke szintén lehet *Null*.

A DHTML-objektumok *getAttribute* metódusa például megadja a paramétereként szereplő tulajdonság értékét:

```
ObjektumAzonosító.getAttribute(tulajdonságnév)
```

Ha az objektum nem rendelkezik a megjelölt tulajdonsággal, akkor a metódus *Null* értéket ad vissza – ezt a későbbiekben ki is használjuk.

A **6–6. példa** következő utasításával egy táblázatnak a *Nev* szövegmezőbe gépelt tulajdonságát jelenítjük meg az *Ertek*-objektum (például SPAN) segítségével:

```
Ertek.innerText = Tablázat.getAttribute(Nev.value)
```

Ha olyan tulajdonságot gépelünk be, amely nem létezik (például gépelési hibát véstünk), akkor *Null*-t kapunk eredményül.

A 2.7. fejezetben megmutattuk, hogy a DHTML-objektumokhoz saját tulajdonságokat rendelhetünk:

```
<H3 id = "Cica" Nev = "Cirmi">
```

Ilyen módon ezeket az objektumokat akár adatok tárolására is használhatjuk. Ha egy olyan tulajdonságot kérdezünk le, ami nem létezik, akkor *Null* lesz a *getAttribute* visszatérési értéke. Ha közvetlenül hivatkoznánk egy ilyen nem létező tulajdonságra, például:



```
Cica.fajta = "angóra"
```

akkor hibaüzenetet kapnánk.

Újabb tulajdonságokat a *setAttribute* módszerrel rendelhetünk az objektumhoz:

```
ObjektumNév.setAttribute("Tulajdonságnév", kifejezés)
```

ahol az új tulajdonság megkapja a kifejezés értékét, például:

```
Call Cica.setAttribute("fajta", "angóra")
```

A sztringként megadott tulajdonságnév helyett írhatunk karaktersorozatot eredményező kifejezést is.

A *setAttribute* alkalmazása után már az objektum azonosítójával minősítve hivatkozhatunk a tulajdonságra:

```
window.alert(Cica.fajta)
```

A *getAttribute* és *setAttribute* használatát a **6–7. példa** mutatja be. Próbáljunk meg olyan tulajdonságot lekérdezni, amelynek még nem adtunk értéket! Figyeljük meg, hogy a forráskódban a *getAttribute* metódust alkalmaztuk!

### Az objektum-hivatkozások *Nothing* értéke

Az 5.2. fejezetben már használtunk objektum-hivatkozásokat a hosszú kifejezések rövidítésére. Tudjuk, hogy egy objektumot a *Set* utasítással lehet egy változóhoz rendelni, amely így megkapja az objektum címét (helyét) a memóriában. Ugyanarra az objektumra több változó is hivatkozhat.

Ha már nincs szükségünk rá, a hivatkozást a *Nothing* (semmi) értékadással törölhetjük:

```
Set VáltozóNév = Nothing
```

Ez megszünteti a kapcsolatot, de nem törli magát a DHTML-objektumot! A változó értéke *Nothing* lesz.

Két objektum-hivatkozás egyezőségét az *Is* (ugyanaz-e) relációs operátorral vizsgálhatjuk meg:

```
ObjektumHivatkozás_1 Is ObjektumHivatkozás_2
```

A reláció eredménye *True*, ha ugyanarra, illetve *False*, ha nem ugyanarra az objektumra hivatkoznak. Az *Is* operátort kell használni akkor is, ha a *Nothing* értéket akarjuk megvizsgálni:

```
Set Objektumhivatkozás = Nothing  
If Objektumhivatkozás Is Nothing Then ' Nothing az értéke
```

Az *IsObject* függvénnyel azt tudjuk lekérdezni, hogy a paramétereként megadott kifejezés objektumra hivatkozik-e – ha igen, akkor *True* lesz a függvény visszatérési értéke. Ha az objektum-hivatkozás *Nothing*, a függvény akkor is *True*-t ad vissza:

```
If IsObject(Változónév) Then ' objektumra hivatkozik
```

Az *Is* és az *IsObject* használatát a **6–8. példa** mutatja be.

## 6.2. Numerikus változók

A továbbiakban részletesebben kitérünk a numerikus változók használatára, az altípusok megválasztásának, a számítások pontosságának a kérdésére.

### Egész és valós altípusok

Ha számokkal kell dolgoznunk a programban, akkor többféle altípus közül választhatunk. A választás befolyásolja a számítások gyorsaságát és pontosságát is.

Ha csak egész számok fordulnak elő, akkor a számtartománytól függően az egész (Integer) vagy a hosszú egész (Long) altípust választhatjuk. Az egész altípusok az úgynevezett fixpontos értékekhez tartoznak. A fixpontos elnevezés a belső számábrázolásra utal.

Fixpontos mennyiség: rögzített hosszúságú törtrésszel tárolt szám.

Az egész altípusok nem rendelkeznek törtrésszel (hossza nulla).

Ha egy változóban törteket is kell tárolnunk, akkor az egyszeres (Single) vagy a kétszeres (Double) pontosságú valós altípus áll a rendelkezésünkre. Ezeket a kódolás miatt lebegőpontos értékeknek nevezzük, mert a tizedesvessző (tizedespont) helye változhat.

Lebegőpontos (valós) mennyiség: egész és tört értékeket is felvehet.

Az egész altípusokkal végzett számítások (összeadás, kivonás, szorzás, maradékos osztás) nagyon gyorsak, mert a mikroprocesszor közvetlenül végre tudja hajtani ezeket a műveleteket. A törteket is tároló lebegőpontos típusoknál (Single, Double) a számítások lelassulnak. A VBScripthez hasonló interpreterek esetén azonban az utasítások értelmezése sokkal több időt vesz igénybe, így a különbség nem jelentős.

Általában is elmondhatjuk, hogy ha nem túl nagy mennyiségű műveletet kell elvégeznünk, akkor rábízhatjuk magunkat az interpreter automatikus típusválasztására.

### Típuskonverzió a számításoknál

Mint említettük, típuskonverzió előírása nélkül az interpreter automatikusan megválasztja a változók altípusát. Egész számok esetén a nagyságrendtől függően az Integer vagy a Long altípust, törtek esetén pedig a Double altípust használja.

Ha egy kifejezésen belül többféle altípus szerepel, akkor a kiértékelésnél az interpreternek át kell alakítania a mennyiségeket egy közös altípusra, amellyel a számítások elvégezhetők. A konverzió további időt vesz igénybe. Ezt azzal csökkenthetjük, ha a típuskonverziós függvények segítségével a későbbi számításoknak megfelelő formában tároljuk az adatokat. Egy egész típusú értéket például célszerű kétszeres pontosságra alakítani, ha lebegőpontos műveletekben fog részt venni:

```
dblEgesz = CDBl(intEgesz)
```

Konkrét számok esetén egyszerűen adjuk meg a 0 törtrészt:

```
dblEgesz = 1.0
```

A tizedespont hatására az interpreter Double formában tárolja a számot.

A CD-n lévő **6–9. példa** segítségével összehasonlíthatjuk a futási időket. A program 50 milliószor elvégzi az  $1 + 0.1$  (az első szám egész, a második valós), illetve az  $1.0 + 0.1$  (mindkét szám valós) összeadást. A futási idő a számítógéptől függően néhányszor tíz másodperc lehet. Közben többször is figyelmeztetést kapunk a böngésző működésének lassulása miatt. Ilyenkor sajnos nekünk kell engedélyeznünk a ciklus további végrehajtását. Mint tapasztalhatjuk, az azonos típusú értékek összeadása rövidebb időt vesz igénybe.

A példaprogramban nem alkalmaztunk folyamatjelzőt az állapotsorban, mert nagyon meghamisítja (és megnöveli) a futási időt. A számítások befejezését párbeszédablak jelzi. Ha mérjük a ciklusok időtartamát, gondoljunk arra, hogy a Windows operációs rendszer egyszerre több programot is futtat, és csak bizonyos időközönként tér rá a szkriptünk utasításainak a végrehajtására.

### Az értékes jegyek száma

A változók értékadásánál figyelemmel kell lennünk az altípushoz tartozó értékészletre, különben túlcordulás jöhet létre. A lebegőpontos mennyiségeknél azonban a tárolás pontossága sem mindig megfelelő.

Alábbi példáinkban Single altípust használunk, hogy kevesebb jegy áttekintésére legyen szükség, de az elmondottak a nagyobb pontosságú Double altípusra is vonatkoznak.

A **6–10. példa** segítségével vizsgáljuk meg, mi kerül a memóriába az alábbi értékadó utasítások hatására:

<i>Értékadó utasítás:</i>	<i>A memóriába kerülő érték:</i>
Adat = 123456	123456
Adat = 123456.7	123456,7
Adat = 123456.78	123456,8

A CD-mellékleten lévő fájlban még több példát is találunk. Bárhogyan próbálkozunk, a Single típussal csak 6–7 számjegy tárolását tudjuk elérni, a 7. számjegyre esetenként kerekített értéket kapunk. Ennek okára az altípusok tárgyalásánál már utaltunk. A lebegőpontos kódolás normálalakban történik, ahol bizonyos mennyiségű bit áll rendelkezésre külön a kitevő és külön a mantissza tárolására.

A Single altípus kódolási módja 6–7 decimális<sup>21</sup> számjegy megadását teszi lehetővé. Mivel az ábrázolási tartomány abszolút értéke  $10^{38}$  és  $10^{-45}$  közé esik, ez a zsebszámológépekhez hasonlóan eltérő pontosságot jelent a különböző nagyságrendű számoknál. A 3-mal való osztás eredményét a **6–11. példa** mutatja be:

<sup>21</sup> Pontosan 24 bináris jegy (1 egész és a törtrész 23 bitje).

$2/3$	$= 0,6666667$
$2000000/3$	$= 666666,7$
$20000000/3$	$= 6666667$
$2E20/3$	$= 6,666667E+19$
$0,2/3$	$= 6,666667E-02$
$0,000002/3$	$= 6,666667E-07$
$2E-20/3$	$= 6,666667E-21$

A  $2/3$ -ot még 6 tizedesjegyre pontosan, egy továbbira pedig kerekítve megkaptuk. Ha 2 millió osztunk 3-mal, akkor már csak egyetlen kerekített tizedesjegy marad,  $2 \cdot 10^{20}$  esetén pedig az eredmény eltérése a pontos értéktől már  $10^{13}$  nagyságrendű. Ha az osztandót csökkentjük, akkor a pontosság növekedni fog.

Lebegőpontos számok használata esetén a pontosság helyett az értékes jegyek számát tudjuk megadni.

Értékes jegyek: a szám normálalakjában szereplő számjegyek.<sup>22</sup>

A  $0,0000346 = 3,46 \cdot 10^{-5}$ -nek három, a  $20876000 = 2,0876 \cdot 10^7$ -nek öt értékes jegye van. Mint említettük, a Single altípus 6–7, a Double pedig 14–15 értékes jeggyel rendelkezik.

A számítások során figyelemmel kell lennünk az értékes jegyek számára. Ha az 1 millióhoz adunk hozzá 1-et, akkor természetesen 1 millió 1-et kapunk, de ha 10 milliót akarunk növelni 1-gyel, az marad 10 millió (Single altípust használva). A CD-melléklet **6–12. példájában** nagyon kicsi számok esetén is láthatunk hasonló eseteket.

A Single altípus mellett szól a gyorsabb számolás, de a szkriptnyelvek esetén az utasítások értelmezésére fordított idő elfedi az időkülönbséget.

### A relatív hiba

A valós típusú mennyiségeknél a számítások során az értékes jegyek száma változatlan marad. Ez azt jelenti, hogy a hiba legfeljebb a változó értékének a tízmilliomod ( $10^{-7}$ ) része a Single, és  $10^{-15}$  része a Double típusnál. Az eltérést a változó értékéhez viszonyítva tudjuk megadni, amit relatív hibának nevezünk.

Relatív hiba: az eltérésnek és a mennyiség értékének a hányadosa.

A relatív hibát gyakran csak egy nála nagyobb értékkel tudjuk becsülni.

Relatív hibakorlát: a relatív hiba felülről becsült értéke.

A Single altípus relatív hibakorlátja  $10^{-7}$ , Double altípusé pedig  $10^{-15}$ .

A számítások hibája és relatív hibája közti különbségre ügyelni kell a programok írásánál. Erre a gyökvonást mutatjuk be példaként. A matematikai kézikönyvek szerint az  $x$  négyzetgyökét a következő sorozattal lehet meghatározni:

---

<sup>22</sup> Itt nem térünk ki a 3,4 és a 3,40 jelölések közti különbség taglalására.

$$a_1 = 1; \quad a_{n+1} = \frac{a_n + \frac{x}{a_n}}{2}$$

Egy közelítő értéket úgy kapunk meg, hogy az előző közelítést behelyettesítjük a képletbe.

Mivel a hibakorlátot csak matematikai eszközökkel tudnánk megadni, a számítás pontosságának ellenőrzéséhez a kapott gyök négyzetét fogjuk összehasonlítani az eredeti számmal. Az eljárást addig folytatjuk, amíg az eltérés kisebb lesz, mint egy előre megadott érték, a négyzet hibakorlátja. Az eltérést a különbség abszolút értékével határozzuk meg, mert nem tudjuk, hogy melyik szám a kisebb:

```
Abs(Kozelito^2 - X) < NegyzetHiba
```

**A 6–13. példa** ciklusa tetszőleges pozitív szám esetén meghatározza a sorozat tagjait:

```
Az X és a megengedett eltérés (NegyzetHiba) bekérése
Kozelito = 1
Ciklus
  Kozelito = (Kozelito + X / Kozelito) / 2
mígnem Abs(Kozelito^2 - X) < NegyzetHiba
Ciklus vége
```

A CD-mellékleten lévő szkript az állapotsorban megjelenít egy számlálót is, ami mutatja az ismétlések számát. A számítás végén kiírja a közelítő érték négyzetét, hogy összehasonlíthassuk az  $x$ -szel. A *NegyzetHiba* értékét a beolvasásánál az összehasonlítás miatt át kell alakítanunk Double altípusúvá.

Futtassuk a programot különböző értékekkel! Néhány próbálkozás után könnyen juthatunk végtelen ciklushoz. Írjunk be az  $x$  helyére például 7 milliárdot (7E9), megengedett eltérésnek pedig egy milliomodot (1E-6). A ciklusszámláló elkezd növekedni, amíg csak meg nem unjuk, és le nem állítjuk.

A végtelen ciklust az okozta, hogy túl nagy volt a szám és túl kicsi a megengedett eltérés. Az értékes jegyek korlátozott száma nem teszi lehetővé az ilyen pontos számolást. Ha pedig növeljük a megengedett eltérés értékét, akkor a kis számok négyzetgyökét kapjuk meg túlságosan pontatlanul. Próbáljunk meg gyököt vonni a 4 milliomodból (4E-6) az eredeti beállítás szerinti 0,001 eltéréssel. Eredményként 0,031-et kapunk, a pontos 0,002 értéknek több, mint a 15-szörösét!

A megoldást az jelenti, ha relatív értéket használunk a kilépési feltételben. Ügyeljünk arra, hogy ez ne legyen kisebb, mint amennyit az értékes jegyek lehetővé tesznek, mert ismét könnyen végtelen ciklushoz juthatunk.

Fenti példánkban a közelítő érték négyzetének relatív hibáját az eltérés és az  $x$  hányadosa adja meg. Ezért a ciklus kilépési feltételét a **6–14. példában** a következőképpen módosítjuk:

```
Abs(Kozelito^2 - X) / X < Relativ
```

Így már a fenti értékekre is megfelelő pontosságú megoldást kapunk.

## Lebegőpontos mennyiségek megjelenítése

Mint láttuk, ha egy számítás során egész és tört értékek is előfordulnak, akkor valós altípust kell választanunk. Az interpreter automatikusan kétszeres pontossággal tárolja a számot.

Mind a Single, mind a Double altípus esetén a tárolás és a számítás egy kicsit pontosabban történik, mint a megjelenítés, melynek utolsó számjegye általában kerekített érték. Ezért – a zsebszámológépekhez hasonlóan – a **6–15. példában** látható eredmények adódnak:

	<i>Single:</i>	<i>Double:</i>
	$1/3 = 0,3333333$	$0,3333333333333333$
de	$(1/3) * 3 = 1$	1
pedig	$0,3333333 \cdot 3 = 0,9999999$	(lásd a példafájlban)
vagy	$2/3 = 0,6666667$	$0,6666666666666667$
de	$(2/3) * 3 = 1$	1
pedig	$0,6666667 \cdot 3 = 2,0000001$	(lásd a példafájlban)

A fenti eredmények azonban megtévesztők lehetnek, ha összehasonlításokban használjuk őket. Bár a kétszeres pontosságú számítás szerint  $3 \cdot 0,1 = 0,3$ , a következő feltétel mégis teljesülni fog, mint azt a **6–16. példa** üzenetablakának megjelenéséből megtudhatjuk:

```
If 3 * 0.1 > 0.3 Then
    window.alert("3 · 0,1 > 0,3 !!!")
End If
```

A példa forráskódjából látható, hogy a szorzatot egy szkript számolta ki. A megjelenítésnél a kerekítés miatt  $3 \cdot 0,1 = 0,3$ , de az összehasonlításnál már megmutatkozik az eltérés.

## Lebegőpontos számok összegezése

A 4–34. példában bemutattuk, hogy a lebegőpontos számokkal végzett műveletek eredményének kerekítése miatt nem célszerű törtszámokat választani a ciklusfejben szereplő mennyiségek értékeként. A ciklus magjában is érhetnek meglepetések. A **6–17. példa** ciklusának végrehajtása után:

```
Osszeg = 0
For I = 1 To 100
    Osszeg = Osszeg + 0.1
Next
```

azt tapasztaljuk, hogy  $100 \cdot 0,1 \neq 10$ . Ezért a valós számok összegezése helyett célszerű osztással és szorzással meghatározni a szükséges értékeket.

A Föld tengelyforgási ideje például  $0,99726957$  nap<sup>23</sup>. A **6–18. példában** kiszámoljuk, hány naponként kerülünk pontosan ugyanabba a helyzetbe, mint amiben most

<sup>23</sup> Ez az úgynevezett csillagnap, a 360°-os fordulat megtételéhez szükséges idő.

vagyunk. Ezt úgy kaphatjuk meg, hogy egy változóhoz egymás után hozzáadjuk a tengelyforgási idő értékét:

```
Const ForgasIdo = 0.99726957
Osszeg = 0
For I = 1 To 10000
    Osszeg = Osszeg + ForgasIdo
    ' ide jön a megjelenítés
Next
```

A CD-n lévő példában a ciklusváltozó nagy végértéke miatt csak a végeredményt írtuk ki. Az eltérés kicsinek tűnik, de csillagászati szempontból nem engedhető meg. A **6–19. példában** összegezés helyett a forgásidő egész számú többszöröseit vesszük:

```
Const ForgasIdo = 0.99726957
For I = 1 To 10000
    Osszeg = ForgasIdo * I
    ' ide jön a megjelenítés
Next
```

Így már sokkal pontosabb eredményt kapunk.

### Lebegőpontos értékek azonosságának vizsgálata

A kerekítési hibák miatt fellépő eltérések megnehezítik két lebegőpontos szám egyenlőségének a vizsgálatát. A **6–20. példa** a Pitagorasz-tétel alapján ellenőrzi, hogy a megadott oldalakkal rendelkező háromszög derékszögű-e. Erre egy feltételes elágazással tudunk válaszolni. Ha a háromszög oldalait A, B, C-vel jelöljük, akkor:

```
Valasz = "nem derékszögű"
If A^2 + B^2 = C^2 Then
    Valasz = "derékszögű"
End If
```

A CD-n lévő példában egy eljárásba írtuk a vizsgálatot, melynek utolsó paramétere a megjelenítést végző SPAN-objektum indexe. A dokumentum betöltésekor megnyugodva tapasztaljuk, hogy a 3; 4 és 5 cm oldalú háromszög derékszögű. Ha azonban az oldalakat  $10^{-6}$ -nal vagy  $10^{16}$ -nal szorozzuk, akkor programunk már nem minősíti derékszögűnek!

A valós altípussal végzett műveletek a kerekítési hibák miatt nem vezetnek pontos eredményre. Ezért az egyenlőség helyett az eltérés relatív hibakorlátját kell megvizsgálni. A relatív hiba meghatározására itt nincs módunk kitérni, de a két érték eltéréseinek és az egyik értéknek a hányadosával becsülhetjük. Ha a relatív hiba egy általunk meghatározott értéknél kisebb, akkor egyenlőnek tekintjük a két számot.

Az eltérést a két mennyiség különbségének abszolút értéke adja meg (nem tudjuk, hogy melyik a kisebb). Ezért a **6–21. példában** az *If* utasítás módosított alakja:

```
Atfogo = C^2
If Abs(A^2 + B^2 - Atfogo) / Atfogo < Relativ Then
    Valasz = "derékszögű"
End If
```

A relatív hibakorlát megszabja, hogy a mennyiségek értékének hányadrészét engedjük meg eltérésként. Az értékes jegyek száma kétszeres pontosság esetén 15-16, a relatív hibakorlátot nyugodtan vehetjük  $10^{-9}$ -nek, ami körülbelül 9 értékes jegyet jelent. Így a számítás pontossága 0,0000001 % lesz, ami teljesen megnyugtató érték.

A program futtatásából láthatjuk, hogy mindegyik háromszöget elvárásainknak megfelelően minősíti. A négyzetre emelés miatt „csak”  $10^{150}$ -ig mehetünk el, de megjegyezzük, hogy Világegyetemünk átmérője „alig”  $3 \cdot 10^{28}$  cm, így ez a tartomány minden igényt kielégít.

### Számolás lebegőpontos értékekkel

Láthattuk, hogy a valós altípus alkalmazásánál mindig kell számítanunk a kerekítésből eredő hibákra. Ezért nem teljesülnek pontosan a matematikai azonosságok. Célszerű elkerülni sok adat összegezését, az egyenlőség helyett pedig a relatív hiba vizsgálatát ajánljuk.

A kerekített értékekkel végzett számítások tulajdonságait, a hibák terjedését, becslését a numerikus analízis tárgyalja. Itt csak ízelítőt adtunk ezekből a problémákból, és felhívtuk a figyelmet arra, hogy ha nem gondolunk rájuk, akkor furcsa eredményeket kaphatunk!

E téma befejezéseként a szerzők egyik kedvenc könyvéből idézünk:

*„A lebegőpontos szám olyan, mint egy homokkupac: ahányszor átlapátoljuk, mindannyiszor elvész egy kis homok, és mindannyiszor több lesz benne a pizok. És néhány számítás után a dolgok egészen tele lesznek szeméttel.”* [KERNIGHAM–PLAUGER]

### Pénznem típusú változók

Mint láttuk, a VBScriptben rendelkezésünkre áll egy nagy tartományt átölelő, de legfeljebb 4 tizedesjegyet kezelő altípus. A Currency használatával közel ezerbillióig lehet adatokat tárolni mind a pozitív, mind a negatív számtartományban. Ez a tizedesekkel együtt több értékes számjegyet jelent, mint amit a kétszeres pontosságú altípusnál elérhetünk.

Mint már utaltunk rá, a pénznem altípus tárolásánál a szám 10000-szerese, tehát egy egész érték kerül be a memóriába. A számításoknál és a kiírásnál azonban ismét az eredeti nagyságrendet kapjuk vissza. Az egész érték tárolásával azonban elkerüljük azt a kerekítési hibát, ami a kettes számrendszer használata miatt a valós típusoknál előfordul. Ennek egy kellemetlen következményével a törtszám értékű ciklusváltozók-nál már találkoztunk.

Currency altípusra a *CCur* konverziós függvény segítségével alakíthatjuk át az értékeket:

*CCur (Kifejezés)*



A számítási pontosságot a **6–22. példában** szereplő

```
1E14 + 0.1 és CCur(1E14) + 0.1
```

összeadások eredményeivel illusztráljuk. Az első összeadásnál nem használtunk konverziós függvényt, mert az interpreter eleve Double altípussal tárolja az értékeket.

A Currency altípussal tehát nagyobb pontosság érhető el, de figyelembe kell vennünk, hogy csak 4 tizedesjegyre számol. A Double vagy akár a Single típusok kevesebb értékes jeggyel dolgoznak, kis számok esetén azonban a számítási pontosság nagyobb. Ezt a **6–23. példa** mutatja be, melynek betöltésekor a következő eredményt láthatjuk:

```
(1 / 3) * 3 = 1
CSng(1 / 3) * 3 = 1
CCur(1 / 3) * 3 = 0,9999
```

A Currency 4 tizedesjegye miatt ugyanis az  $1/3 = 0,3333$ , amit ha megszorozunk 3-mal, nyilván nem kapjuk vissza az 1-et.

A pénznem altípus használata esetén módunk van az eredményt az operációs rendszer területi beállításainál megadott formában megjeleníteni. Ezt a *FormatCurrency* függvénnyel érhetjük el, amely a formázásra vonatkozó beállítások mellett kiírja a pénznem jelölését is. A függvény paraméterei megegyeznek a *FormatNumber* paramétereivel. A *FormatCurrency* használatát a **6–24. példában** mutatjuk be.

A nagy pontosság miatt a Currency altípust a pénzügyi számításoknál szokták alkalmazni. Ha nincs különösebben szükségünk az általa nyújtott pontosságra, akkor kerüljük a használatát, mert lényegesen lelassítja a kifejezések kiértékelését.

### 6.3. Karakterláncok kezelése

A programokban gyakran használunk karaktersorozatokat (karakterláncokat, sztringeket). A VBScriptben egy karakterlánc változó hosszúságú lehet, és több, mint 2 milliárd karaktert tartalmazhat. Ezt elég nehéz elérni.

Tudjuk azt is, hogy a beolvasásnál és a kivitelnél mindig karakterláncot kezelünk. Az eddigiekben megismerkedtünk az összefűzésükkel, összehasonlításukkal, a továbbiakban pedig áttekintjük a sztringekre vonatkozó függvényeket.

#### Váltás a kis- és nagybetűk között

A karaktersorozatok összehasonlításánál láttuk, hogy a VBScript megkülönbözteti a kis- és a nagybetűket egymástól. Ezért volt szükség a felhasználó figyelmeztetésére a hónapok hosszát meghatározó 4–14. példában, hogy kisbetűkkel gépelje be a hónap nevét. Ez gyakran okoz problémát az adatok beolvasásánál.

Az *UCase* (uppercase, nagybetű) függvénnyel egy sztring karaktereit nagybetűkké alakíthatjuk. A függvény paramétereként egy sztringet eredményező kifejezést kell megadni. A függvény hatástalan a nem betű karakterekre, például:

```
UCase("AbC12 + dEf34") = ABC12 + DEF34
```

Az *LCase* (lowercase, kisbetű) éppen ellenkezőleg, a karakterlánc minden betűjét kisbetűvé alakítja:

```
LCase("AbC12 + dEf34") = "abc12 + def34"
```

A **6–25. példában** átalakítjuk a 4–14. példa teszt kifejezését kisbetűsre:

```
Select Case LCase(Honap.value)
```

Ezzel nagybetűk használata esetén is felismeri programunk a hónap nevét. Megjegyezzük, hogy az *UCase* és *LCase* függvények nevében szereplő *Case* nincs kapcsolatban a *Select Case* utasítással.

### A felesleges szóközök elhagyása

Az adatbevitelnél a felhasználó véletlenül (vagy szándékosan) néhány szóközt szintén begépelhet a hónap neve előtt vagy után. A szóközök a sztringekben normál karakternek számítanak, ezért az összehasonlításban is részt vesznek. A "január" karakterlánc viszont nem egyenlő a " január" vagy a "január " sztringgel.

A *Trim* (jelentése rendbe hoz, igazít) függvények kitörlik a szóközöket a paraméterként megadott karaktersorozatból. Az *LTrim* (left, bal oldal) az elejétől, az *RTrim* (right, jobb oldal) a végétől, a *Trim* pedig mindkét oldaláról elhagyja a szóközöket. A többi karakter közé eső szóközök természetesen megmaradnak:

```
LTrim(" A és B ") = "A és B "  
RTrim(" A és B ") = " A és B "  
Trim(" A és B ") = "A és B "
```

A **6–26. példában** a *Trim* segítségével kitöröltük a fölösleges szóközöket a begépelte hónapnévből, így ez sem zavarja a napok számának meghatározását.

### A karakterlánc hossza

A továbbiakban szükségünk lesz a sztring karaktereinek a számára. Ezt a *Len* függvény határozza meg:

```
Len("A + B = C") = 9
```

A **6–27. példában** olyan weblapot készítünk, amely egy legalább 5 és legfeljebb 10 karakterből álló jelszót kér be a felhasználótól. A szövegmezőbe írható karakterek számát a *maxLength* tulajdonsággal korlátozhatjuk. (A szövegmezők tulajdonságait a 2–49. példa mutatta be).

A minimális hossz ellenőrzésére a *Len* függvényt alkalmazzuk:

```
If Len(Jelszo.value) < 5 Then  
    window.alert("Írjon be hosszabb jelszót!")  
    Jelszo.select()  
Else  
    window.alert("Rendben.")  
End If
```

## Karakterek elhagyása a sztring elejéről és a végéről

Egy karakterlánc elejéről a *Left*, végéről a *Right* függvénnyel vághatunk le karaktereket:

*Left*(*Sztring*, *Szám*) adott számú karaktert ad vissza a sztring elejétől kezdve  
*Right*(*Sztring*, *Szám*) a sztring adott számú karakterét adja vissza a végétől kezdve

Ha a szám nagyobb, mint a sztring hossza, akkor az egész sztring lesz a visszatérési érték. Ha a paraméterként megadott karakterlánc üres, akkor a függvényérték is üres sztring lesz.

Néhány példa a függvények visszatérési értékére:

<i>Left</i> ("Almafa", 4) = "Alma"	<i>Right</i> ("Körtefa", 2) = "fa"
<i>Left</i> ("Almafa", 1) = "A"	<i>Right</i> ("Körtefa", 1) = "a"
<i>Left</i> ("Almafa", 9) = "Almafa"	<i>Right</i> ("Körtefa", 9) = "Körtefa"

A **6–28. példa** kiírja a „Vakáció!” karakterláncot úgy, ahogy az év végén az iskolai táblákon szokott megjelenni (soronként egy-egy betűvel több, a felkiáltójeltől kezdve). A *Right* függvény segítségével ciklust szervezünk, amely egyre több betűt illeszt a kiírásra kerülő sztringhez:

```
For I = 1 To 8 ' a sztring 8 karakterből áll
    document.write(Right("Vakáció!", I) & "<BR>")
Next
```

A ciklusszámláló végértékének a megadásához nekünk kell meghatároznunk a karakterlánc hosszát. Ehhez felhasználhatjuk a *Len* függvényt, így ugyanaz az iteráció tetszőleges hosszúságú karaktersorozatot tud kezelni:

```
For I = 1 To Len("Vakáció!")
```

A *Left* és *Right* függvényekkel törölhetjük a felesleges elválasztójeleket a listák megjelenítésénél. A következő ciklus például egy tömb elemeinek kiírásához készít sztringet:

```
Lista = ""
For I = 0 To UBound(Tomb)
    Lista = Lista & Tomb(I) & "; "
Next
```

Az utolsó elem után álló pontosvesszőt és szóközt a *Left* függvénnyel vághatjuk le:

```
Lista = Left(Lista, Len(Lista) - 2)
```

## Részek kiemelése a sztringből

A *Mid* (middle, közép) függvény a kezdő pozíciótól indulva adott számú karakterrel tér vissza:

*Mid*(*Sztring*, *KezdőPozíció*, *Szám*)

A *Mid*("Visual Basic Script", 8, 5) például a sztring nyolcadik karakterétől kezdve ad vissza 5 karaktert, értéke tehát "Basic" lesz. Mint látjuk, a szóközöket is

beleszámítja a karakterek közé. Ha nincs elegendő karakter a sztringben, akkor a függvényérték a karaktersorozat végéig tartalmazza a karaktereket:

```
Mid("Visual Basic Script", 8, 25) = "Basic Script"
```

Ha a *KezdőPozíció* nagyobb, mint a *Sztring* hossza, akkor üres karakterlánccal ("" ) tér vissza.

Könnyű belátni, hogy

```
Mid(Sztring, 1, Szám) = Left(Sztring, Szám)
```

A **6–29. példa** programja egy megadott szót betűnként egymás alá, függőlegesen ír ki a képernyőre. Ehhez egy ciklusban a *Mid* függvény segítségével egyesével kiolvassuk, és kiírjuk a karaktereket úgy, hogy mindegyik után új sort kezdünk:

```
Fuggoteles = ""
For I = 1 To Len(Szoveg)
    Fuggoteles = Fuggoteles & Mid(Szoveg, I, 1) & "<BR>"
Next
FuggotelesKi.innerHTML = Fuggoteles
```

### Karakterek ismétlése

Ha egy karakter adott számú ismétlésével kell sztringet készítenünk, akkor a *String* függvényt használjuk:

```
String(Szám, Karakter)
```

A karaktert idézőjelben kell megadni, például:

```
String(5, "*") = "*****"
```

A **6–30. példában** úgy módosítjuk a vakációs példa kódját, hogy a kiírt karakterek elé annyi csillag kerül, amennyi éppen 8-ra egészíti ki a számukat. Ezt a ciklusban a 8-I kifejezés adja meg (I darab karaktert írunk ki a sztringből):

```
For I = 1 To 8
    document.write(String(8 - I, "*"))
    document.write(Right("Vakáció!", I) & "<BR>")
Next
```

A megjelenítés csak akkor mutat jól, ha egyforma széles karaktereket használunk. Ezért a CD-n lévő példában a sorokat egy PRE-objektumba tettük.

Megjegyezzük, hogy a karakter helyett a függvénynek megadhatjuk az ANSI-kódot is. Ha ez nagyobb, mint 255, akkor helyette a 255-tel való osztás maradékát veszi.

Ha szóközöket kell egymás után illesztenünk, akkor a *Space* függvényt használhatjuk. A *Space(Szám)* a megadott számú szóközből készít karakterláncot. A **6–31. példában** csillagok helyett szóközöket írtunk a vakáció részei elé. Ne felejtjük el, hogy a böngésző a szóközökből csak egyet jelenítene meg, ezért feltétlenül használunk kell a PRE-objektumot!

## Keresés a sztringben

Egy sztringen belül adott karaktersorozatot az *InStr* (in string, a sztringen belül) függvénnyel kereshetünk meg. Hívásának szintaxisa:

```
InStr(Start, AdottSztring, KeresettSztring, Mód)
```

A függvény az *AdottSztring* karaktersorozatban a *Start* pozíciótól kezdve megkeresi a *KeresettSztring* karakterláncot. Ha a *Mód* = 0, akkor az összehasonlításnál megkülönbözteti a kis- és a nagybetűket, ha a *Mód* = 1, akkor pedig nem. A *Mód* = 0 érték el is hagyható.

A függvény visszatérési értéke a keresett sztring kezdő pozíciója. Ha nem találta meg a karaktersorozatot, akkor nullát ad vissza. Ugyancsak 0 lesz a visszatérési érték, ha az *AdottSztring* üres, vagy a *Start* pozíció nagyobb, mint az *AdottSztring* hossza. Ha a *KeresettSztring* üres, akkor a függvény a *Start* értékét adja vissza.

Néhány példa az *InStr* függvény alkalmazására:

```
InStr(1, "Mehemed", "me", 0) = 5
InStr(1, "Mehemed", "me", 1) = 1
InStr(3, "Mehemed", "Me", 1) = 5
InStr(3, "Mehemed", "Me", 0) = 0
InStr(1, "Mehemed", "e", 0) = 2
```

Mint láthatjuk, ha a keresett sztring többször is előfordul, akkor csak az első pozíciójával tér vissza.

A **6–32. példában** olyan programot készítünk, amely egy megadott nevet felbont vezetéknévre és keresztnévre. A névben egy szóköz választja el a két részt egymástól, egyéb szóközöket pedig nem írunk be.

A megadott karakterláncban meg kell keresni a szóközt. A vezetéknév az elejétől az eggyel kisebb pozícióig tart, a keresztnév pedig a szóköz utáni karaktertől a sztring végéig. A keresztnév karaktereinek számát tehát úgy kapjuk meg, hogy a név hosszából kivonjuk a szóköz pozícióját:

```
SzokozHelye = InStr(1, Nev, " ", 0)
VezetekNev = Left(Nev, SzokozHelye - 1)
KeresztNev = Right(Nev, Len(Nev) - SzokozHelye)
```

A karakterláncban a végéről kezdve visszafelé is tudunk keresni az *InStrRev* (reverse, visszafelé) függvény segítségével, mely paraméterei megfelelnek az *InStr* függvény paramétereinek, de a sorrend más:

```
InStrRev(AdottSztring, KeresettSztring, Start, Mód)
```

A visszatérési érték természetesen az adott sztring elejétől kezdve határozza meg a keresett sztring kezdőpozícióját. Például:

```
InStrRev("Mehemed", "Me", 7, 1) = 5
```

Mindkét sztringkereső függvényt megadhatjuk a *Start* és a *Mód* paraméterek nélkül is:

```
InStr(AdottSztring, KeresettSztring)
InStrRev(AdottSztring, KeresettSztring)
```

Ekkor megkülönböztetik a kis- és nagybetűket (*Mód* = 0), a keresés pedig a sztring elejéről, illetve a végéről indul.

### A bevitel ellenőrzése az *InStr* függvénnyel

Az *InStr* függvényt a bevitel ellenőrzésénél is alkalmazhatjuk, ha csak egy-egy karaktert fogadunk, de hosszadalmas lenne az elágazásokkal történő vizsgálata. Ha például csak magánhangzó beírását akarjuk engedélyezni, akkor készítsünk egy karakterláncot a magánhangzókból (**6–33. példa**):

```
Const Maganhangzo = "aáééííoóöőúúüüű"
```

A vizsgálatnál keressük meg a sztringben a begépelt karaktert:

```
If InStr(Maganhangzo, Szovegmezo.value) = 0 Then  
    ' nem magánhangzót írtak be, hibaüzenetet adunk  
Else  
    ' magánhangzót írtak be, következhet a feldolgozás
```

Gondoljuk meg, hogy *If ... ElseIf*-ek láncolatával vagy a *Select Case* utasítással elég hosszadalmas lett volna az ellenőrzés! A módszer arra az esetre is bővíthető, amikor nem csak egyetlen karaktert ellenőrzünk.

### Sztringrészek helyettesítése

Egy karakterláncon belül sztringrészeket a *Replace* (helyettesít) függvénnyel cserélhetünk ki egy másik karaktersorozatra, melynek szintaxisa:

```
Replace(EredetiSztring, KeresettSztringRész, HelyettesítőSztring)
```

A függvény az *EredetiSztring* karaktersorozatban megkeresi a *KeresettSztringRész* összes előfordulását, és mindegyiket lecseréli a *HelyettesítőSztring* karakterláncra. A visszatérési érték az átalakított karakterlánc lesz. A következő utasítások végrehajtása után például:

```
RegiSztring = "Egyedem, begyedem, betegyem? Kivegyem?"24  
UjSztring = Replace(RegiSztring, "egye", "ögyö")
```

az *UjSztring* értéke "Egyedem, bögyödem, betögyöm? Kivögyöm?" lesz, mert minden „egye” sztringrészt lecserél „ögyö”-re. Mint látjuk, ebben a formájában megkülönbözteti a kis- és nagybetűket egymástól.

A függvény további paramétereit megadhatjuk úgy, hogy a keresést és a helyettesítést a *Start* pozíciótól végezze, a helyettesítések száma *Szamlal* legyen, és milyen módot alkalmazzon az összehasonlításnál:

```
Replace(Eredeti, Keresett, Helyettesítő, Start, Számlál, Mód)
```

A *Mód* paraméter ugyanazt a szerepet játssza, mint az *InStr* függvénynél (0 esetén megkülönbözteti a kis- és a nagybetűket, 1 esetén pedig nem). Használatakor meg kell adni a *Start* és a *Számlál* paramétereket is. Ha azt akarjuk, hogy az összes lehetséges helyettesítést elvégezze, akkor a *Szamlal* értékét állítsuk -1-re!

---

<sup>24</sup> Móricz Zsigmond: A veréb, részlet.

Ha az eredeti karaktersorozat üres, vagy a *Start* pozíció nagyobb, mint az eredeti karakterlánc hossza, akkor a függvény üres sztringet ad vissza. Ha a keresett sztring-rész üres, vagy a *Számlál* paraméter értéke 0, akkor pedig az eredeti sztring lesz a függvényérték.

Az előző versrészlet például a következőképpen alakul:

```
Replace(RegiSztring, "egye", "ögyö", 1, 2, 1) =  
                                         "ögyödem, bögyödem, betegyem? Kivegyem"  
Replace(RegiSztring, "egye", "ögyö", 5, 1, 0) =  
                                         "dem, bögyödem, betegyem? Kivegyem?"
```

Figyeljük meg, hogy a visszaadott sztring a *Start* pozíciótól indul, nem pedig az eredeti karakterlánc elejétől!

A **6–34. példa** a *Replace* függvény használatával olyan programot mutat be, amely a gyerekek által használt „madárnyelvre” fordít le egy szöveget, a magánhangzók „ava”, „ává” stb. helyettesítésével:

„tudsz így beszélni?” helyett „tuvudsz ívígy beveszévélnivi?”

A kereséshez a magánhangzókat egy tömbben helyezzük el (az *Array* függvényt az 5.1. fejezetben ismetettük):

```
Maganhangzo = Array("a", "á", "e", "é", "i", "í", "o", "ó", _  
                    "ö", "ő", "u", "ú", "ü", "ű")
```

A beolvasott szövegben egy ciklus segítségével az összes magánhangzót megduplázzuk úgy, hogy egy „v” betűt illesztünk közéjük:

```
For Each Betu in Maganhangzo  
    Szoveg = Replace(Szoveg, Betu, Betu & "v" & Betu)  
Next
```

Példánk csak a kisbetűket helyettesíti. Javasoljuk az Olvasónak, hogy egészítse ki a programot a nagybetűs magánhangzók „fordításával” is!

## A karakterlánc megfordítása

Bár az eddig megismert függvények segítségével könnyen fel tudnánk írni egy sztringet visszafelé, ezt egy újabb függvénnyel ciklus nélkül is megtehetjük. Az *StrReverse* (string reverse, karakterlánc megfordítása) fordított sorrendben adja vissza a sztring karaktereit:

```
StrReverse("Visual Basic") = "cisaB lausiV"
```

Készítsünk programot, amely megvizsgálja, hogy a begépelt karaktersorozat tükörmondat-e (azaz visszafelé olvasva ugyanaz a jelentése)! Ehhez csak össze kell hasonlítanunk a sztringet a fordítottjával. Ha egyenlők, akkor tükörmondatot (palindromát) írtunk be, ha nem, akkor az eredményhez hozzáfűzzük a „nem” szócskát:

```
VisszaSzoveg = StrReverse(Szoveg)
Eredmeny = "A szöveg"
If Szoveg <> VisszaSzoveg Then
    Eredmeny = Eredmeny & " nem"
End If
Eredmeny = Eredmeny & " palindroma."
```

A palindromák esetén meg szokták engedni a szóközök helyének változtatását, és nem veszik figyelembe a kis- és nagybetűk közti különbséget. Így tükörmondatnak tekintjük a „Géza kék az ég” karakterláncot is. Ezért a megfordítás előtt vegyük ki a szövegből a szóközöket (üres sztringgel helyettesítünk):

```
Szoveg = Replace(Szoveg, " ", "")
```

Az összehasonlításnál pedig mindkét karaktersorozatot alakítsuk át például nagybetűsre (használhatnánk az *StrComp* függvényt is):

```
If UCase(Szoveg) <> UCase(VisszaSzoveg) Then ...
```

A CD-n lévő **6–35. példa** már ilyen formában tartalmazza a kódot. Mivel a fordított szöveget is kiíratjuk, a *Szoveg* és a *VisszaSzoveg* sztringekből a szóközöket csak a megfordítás után töröljük.

Programunk még így sem teljes, mert az írásjelek miatt a „Géza, kék az ég!” mondatot nem tekinti palindromának. A módosítást az Olvasóra bízuk.

### Sztringek szétválasztása és egyesítése

A *Split* függvény a megadott karakterláncot a határoló karaktereknél elvágja, a sztringrészeket pedig egy dinamikus tömb elemeiként adja vissza úgy, hogy a határoló karaktereket elhagyja:

```
Split(karakterlánc, határoló_karakter)
```

Az értékadás hasonló az *Array* függvény használatához, a tömböt egyszerű változóként kell deklarálni. A

```
Dim Szavak
Szavak = Split("Géza kék az ég", " ")
```

hatására a *Szavak* tömb elemei:

```
Szavak(0) = "Géza" : Szavak(1) = "kék"
Szavak(2) = "az"   : Szavak(3) = "ég"
```

A tömb elemszámát az *UBound* függvénnyel határozhatjuk meg.

A *Join* függvény a *Split* fordítottjának tekinthető. Egy tömb elemeit egyesíti egyetlen karakterláncba. Előírhatjuk az elemeket elválasztó sztringet is:

```
Join(Tömbnév, elválasztó_sztring)
```



A tömb nevét indexek nélkül kell megadni. A sztring lehet üres, ekkor nem kerül elválasztójel az elemek közé. Ha nem adjuk meg a sztringet, akkor a VBScript szóközt alkalmaz. A

```
Tomb(0) = "Géza" : Tomb(1) = "kék" : Tomb(2) = "az ég"
```

esetén például:

```
Join(Tomb, "-") = "Géza-kék-az ég"
```

A függvények használatát a **6–36. példa** mutatja be, amelyben egy mondatot szavanként fordítunk meg. A *Split* függvénnyel szétvágjuk szavakra, egy ciklussal minden szót megfordítunk, majd a *Join* függvénnyel ismét összeállítjuk a mondatot.

## A karakterek ANSI-kódja

Az első fejezetben említettük, hogy a számítógép az egyes karakterek ANSI-kódját, illetve Unicode-értékét tárolja a memóriában vagy a háttértárakon. Az eddigiekben felsorolt függvények a tárolás módjának megfelelően, automatikusan kezelik ezeket a kódokat. Előfordulhat azonban, hogy egy karaktert az ANSI-kódból szeretnénk előállítani, illetve éppen fordítva, meghatározni egy karakter ANSI-kódját. Erre a célra a *Chr* és az *Asc* függvényeket használjuk.

A *Chr* függvény megadja a paramétereként szereplő ANSI-kódnak megfelelő karaktert:

```
Chr(65) = "A", Chr(97) = "a", Chr(177) = ± stb.
```

Emlékeztetünk arra, hogy az ANSI-kód 0 és 255 közé eső egész szám, de a 32-nél kisebb értékek nem jeleníthetők meg a képernyőn (Backspace, Enter, ESC stb.), a 32 pedig a szóköznek felel meg. A *Chr(0)* azonban nem a 0 számjegyet jelenti, hanem az úgynevezett NUL karaktert. Az is teljesül, hogy `"" < Chr(0)`.

A *Chr* függvényt főleg az eseménykezelésnél használjuk, a billentyűzetről történő bevitel vizsgálatára.

Az *Asc* függvény egy karakterből állítja elő az ANSI-kódját (a jelölés a régebben használt ASCII-kódrendszer maradványa):

```
Asc("A") = 65, Asc("a") = 97 stb.
```

A két függvényt egymás után alkalmazva visszakapjuk az eredeti értéket:

```
Karakterkód = Asc(Chr(Karakterkód))  
Karakter = Chr(Asc(Karakter))
```

A **6–37. példa** programja megadja egy karakter kódját, vagy egy kódhoz az általa meghatározott karaktert. Csak egyetlen szövegmezőt használunk. Ha a begépelt sztring egy karakterből áll, akkor megjelenítjük az ANSI-kódját, egyébként pedig kódként értelmezzük, és kiírjuk a megfelelő karaktert (az egyjegyű ANSI-kódok nem írhatók ki a képernyőre). Elvégezzük a hibás bevitel vizsgálatát is.

Ha csak egy karaktert írtak be a szövegmezőbe, akkor megjelenítjük az ANSI-kódját:

```
If Len(Adat) = 1 Then  
    Eredmeny.innerText = "A karakter ANSI-kódja: " & Asc(Adat)
```

Ha hosszabb a karakterlánc, de nem értelmezhető számként, akkor rossz adatot adtak meg:

```
ElseIf Not IsNumeric(Adat) Then  
    Eredmeny.innerText = "Nem ANSI-kódot írt be!"
```

A számok közül is csak a 0 és 255 közé esőket fogadjuk el:

```
ElseIf Adat < 0 Or Adat > 255 Then  
    Eredmeny.innerText = "Az ANSI-kód 0 és 255 közé esik!"
```

32-nél kisebb kód esetén nem tudjuk megjeleníteni a karaktert:

```
ElseIf Adat < 32 Then  
    Eredmeny.innerText = "Nem lehet megjeleníteni a karaktert."
```

Ha minden feltételnek megfelelt, akkor kiírjuk a kódnak megfelelő karaktert:

```
Else  
    Eredmeny.innerText = "A kódnak megfelelő karakter: " & Chr(Adat)  
End If
```

Nem vizsgáltuk meg, hogy egyáltalán egész számot írtak-e be, de a HTML-kódban korlátoztuk a beírható sztring hosszát, így ez a hiányosság nem zavarja meg a szkript működését.

A CD-n lévő **6–38. példában** egy táblázat segítségével megjelenítettük az összes ANSI-kódnak megfelelő karaktert. Javasoljuk az Olvasónak, hogy alaposan nézze át a program kódját! Átismételheti vele a táblázatok kezelését is.

Megjegyezzük, hogy a billentyűzeten begépelte és a képernyőn megjelenő karaktereket a Windows Területi beállításai, köztük a billentyűzet beállításai határozzák meg. Ettől függetlenül a dokumentumban meg lehet adni a böngésző által alkalmazott karakterkészletet (ezt a 2.1. fejezetben ismertettük a META-objektumoknál). Könyvünkben a magyar beállításokat, illetve a Windows-1250-es karakterkészletet használjuk.

### A karakterlánc-függvények paraméterei

A karakterlánc függvények paraméterei természetesen nem csak konkrét karakter-sorozatok vagy sztring típusú változók lehetnek, hanem tetszőleges olyan kifejezések, melyek eredménye sztringként értelmezhető. Például:

```
Mid("apad" & "lótusz", 2, 5) = "padló", Left(2 * 100, 2) = "20"
```

A program futása során az is előfordulhat, hogy a paraméterek értékét több lépésben állítottuk elő, és *Null* eredményt kaptunk. Ekkor általában *Null* lesz a sztringfüggvény visszatérési értéke.

Hibaüzenettel megszakad a program végrehajtása, ha

- az *StrReverse* paramétere *Null*,
- a *Replace* valamelyik paramétere *Null*,
- az *InStr* vagy az *InStrRev* *Start* paramétere *Null*.

A programokban a függvények használata előtt ellenőrizni kell a paramétereket, hogy futási hibák ne fordulhassanak elő.

A fentiekben nem soroltuk fel az összes lehetséges paraméterezést és speciális esetet. Ezeket megtalálhatjuk a VBScript dokumentációiban.

Itt említjük meg, hogy a karakterlánc-függvények az operációs rendszer által használt ANSI-kódnak vagy Unicode-nak megfelelően kezelik a sztringeket. Néhány függvénynek azonban ettől függetlenül létezik olyan változata, amely bájtokkal dolgozik. Ezeket egy B betű jelöli a függvény neve után:

*AscB*, *ChrB*, *InStrB*, *LeftB*, *LenB*, *MidB*, *RightB*

A *LenB* például ANSI-kód esetén a karakterek számát adja meg, Unicode esetén pedig ennek a kétszeresét. Egy Unicode-karaktert ugyanis két bájt tárol. Az *AscW* és *ChrW* függvények viszont a beállítástól függetlenül Unicode-ot használnak.

## Parancssori paraméterek beolvasása

A HTML-alkalmazások ismertetésénél említettük, hogy programunk indításánál megadhatunk paramétereket, melyeket a *HTA:Application*-objektum *commandLine* tulajdonsága tárol. A tulajdonság értéke egyetlen sztring, amely az idézőjelbe tett elérési utat és a paramétereket tartalmazza.

A paraméterek elválasztásához általában szóközöket alkalmazunk. Így a *Split* függvénnyel emelhetjük ki őket a *commandLine* értékéből. Az algoritmust a **6–39. példában** mutatjuk be, amelyben megjelenítjük és összeadjuk a paramétereket. A számokat a parancssorban adjuk meg a program indításánál, vagy a parancsikon tulajdonságlapjára írjuk be.

A *commandLine* idézőjelek közé írva tartalmazza a fájl elérési útját. Ezt követik a szóközökkel elválasztott paraméterek. A *commandLine* első karakterétől (idézőjel) eltekintve, a második karaktertől kezdve megkeressük az elérési utat lezáró idézőjelet. Idézőjelet nem adhatunk meg a keresett sztringben, ezért ANSI-kódjával (34) helyettesítjük. A *Right* függvénnyel elhagyjuk a sztringből az idézőjelig terjedő részt (az elérési utat):

```
Temp = App.commandLine
Kezd = InStr(2, Temp, Chr(34)) + 1
Temp = Right(Temp, Len(Temp) - Kezd)
```

A karakterláncot a *Split* függvénnyel a szóközöknél vágjuk szét. Mivel a különböző módokon (parancsikon, parancssor) történő indításnál a Windows változó számú szóközt illeszt a paraméterek elé, ezeket az *LTrim* függvénnyel hagyjuk el:

```
Temp = LTrim(Temp)
Param = Split(Temp, " ")
```

Végül összeadjuk az eredményként kapott tömb elemeit:

```
Osszeg = 0
For I = 0 To UBound(Param)
    Osszeg = Osszeg + Param(I)
Next
```

A CD-n lévő példa kiírja a *commandLine* tulajdonság és a paraméterek értékét is. Mivel csupán az eljárást akartuk bemutatni, nem végeztünk hibaellenőrzést. A futtatásnál tehát csak számokat adjunk meg paraméterként, közöttük pontosan egy szóközt hagyva.

Sajnos a batch-fájlok feldolgozásánál a Windows a régi ASCII-karakterkódokat használja. Ezért alkalmazásuknál ne írjunk ékezetes magánhangzókat a fájlnevekbe! A CD-n lévő *Parancs.bat* állomány az előző példát futtatja, de a fájlnevet átírtuk *Pelda6-39.hta*-ra. Windows 98 esetén készítsünk parancsikont a *.bat* állomány helyett a futtatáshoz, és tulajdonságainál adjuk meg a paraméterek értékét.

## 6.4. Dátumok és időpontok

A változók ismertetésénél megemlítettük a *Date* és *Time* altípusokat. Használatuk nagymértékben megkönnyíti a dátumok és időpontok kezelését, amit a szokásos numerikus értékekkel csak nagyon nehézkesen tudnánk elvégezni. A továbbiakban röviden áttekintjük a két altípusra vonatkozó tudnivalókat.

### Dátum és időpont megadása

A dátumot és az időpontot speciális karaktersorozatként adjuk meg. A konkrét értékeket kettős keresztek (#) közé írjuk. A Windows Területi beállításainál megadott évszámoknál elhagyható az első két számjegy (az évszázad). Sajnos ez a lehetőség néhány Windows változatnál a 2001 és 2012 közötti évszámokra nem alkalmazható.

A részek közé a dátumoknál / jelet vagy kötőjelet (pontot nem!), az időpontoknál kettőspontot írunk. Az időpontoknál használható az *am* (délelőtt), illetve a *pm* (délután) rövidítés. A dátum és az idő közé szóköz kerül, de bármelyikük el is hagyható:

```
SzuletesiDatum = #1986/04/21#      ' 1986. április. 21.
Napfogyatkozás = #99-08-11#       ' 1999. augusztus 11.
Buli = #2004/05/15 18:30#         ' 2004. május 15., este fél 7
TV_sorozat = #07:20pm#           ' este 7 óra 20 perc
Start = #14:05:11#               ' 14 óra 5 perc 11 másodperc
```

A vezető nullákat egyik résznél sem kell kiírni: *#1986/4/21#* vagy *#7:20am#*.

A VBScript még számos (például amerikai) formát elfogad, ezeket azonban nem tárgyaljuk. Megjegyezzük, hogy a sorrend szoros kapcsolatban van a Windows Vezérlőpultján a Területi beállításoknál megadott formátummal. Emlékeztetjük az Olvasót arra, hogy 100. január 1. és 9999. december 31. közötti dátumokat használhatunk.

Ha a dátum vagy az idő bármely része nem a megengedett tartományba esik, akkor hibaüzenetet kapunk.

## Műveletek a dátumokkal

A VBScript a dátum év, hónap, nap értéke helyett az 1899. december 30. óta eltelt napokat tárolja (az előtte lévő dátumok értéke negatív). Az időpontot ehhez adja hozzá a nap törtrészében kifejezve. Így például:

```
Cdbl(#1899-12-30 12:00#) = 0,5
```

Ha csak az időpontot adjuk meg, akkor a dátumot 1899. december 30-nak veszi, de ez az időpontok kezelését általában nem zavarja.

Ennek ismeretében a dátumokat, időpontokat összeadhatjuk vagy kivonhatjuk egymásból. Két dátum különbsége például a köztük eltelt napok számát adja meg:

```
EnnyiNaposVagyok = MaiDatum - SzuletesDatuma
```

Közben természetesen figyelembe veszi a hónapok hosszát és a szökőéveket is.

Dátumokkal és időpontokkal egyéb matematikai műveleteket is végezhetünk. Ezeket a tárolt numerikus értékekkel hajtja végre az interpreter.

## Dátum- és időfüggvények

A számítógépen beállított rendszeridőt a *Date* (dátum) és *Time* (időpont) függvényekkel kérdezhetjük le, amelyek nem rendelkeznek paraméterekkel (**6–40. példa**):

```
MaiDatum = Date()  
PontosIdo = Time()
```

A két értéket együttesen a *Now* (most) függvény adja meg: `Most = Now()`. Figyeljünk arra, hogy bár az idő kiírása a Területi beállításoknál szereplő formában történik, a függvényérték a nap törtrészében jelzi az időt!

Egy dátumtól kezdve mostanáig az eltelt napok száma például:

```
EnnyiNaposVagyok = Date() - SzuletesDatuma
```

Ha dátumot vagy időpontot kérünk be a felhasználótól egy szövegmezőben, akkor kényelmetlen lenne begépelni a kettős keresztek. Egy sztringet a *CDate* függvény alakít át *Date* vagy *Time* altípusú változóra:

```
CDate(Szovegmezo.value)
```

A megjelenítésnél nincs szükség a *CStr* függvényre, a konverzió a numerikus értékekhez hasonlóan automatikusan megtörténik:

```
MaiDatum.innerText = Date()
```

Mielőtt a *CDate* függvényt használnánk, célszerű ellenőrizni, hogy a szövegmezőbe írt adat értelmezhető-e dátumként, különben hibaüzenetet kapunk. Az *IsDate* függvény hasonló szerepet tölt be, mint az *IsNumeric* a numerikus változóknál:

```
If IsDate(Szovegmezo.value) Then  
    ' érvényes dátumot gépeltek be, kezdődhet a feldolgozás  
Else  
    ' nem dátumot gépeltek be, hibaüzenetet adunk  
End If
```

A CD-melléklet **6–41. példájában** az eddigi függvények felhasználását mutatjuk be. A *Weekday* (a hét napja) függvény meghatározza, hogy a paramétereként megadott dátum a hét hányadik napjára esik (ebben a formájában az első nap a vasárnap), a *WeekDayName* pedig a Területi beállításoknak megfelelően visszaadja a nap nevét.

A VBScript számos dátumot és időt kezelő függvénnyel rendelkezik. A CD-melléklet Dokumentumok mappájában lévő DátumIdő.htm fájl rövid magyarázattal ellátva bemutatja ezeket a függvényeket.

### Időmérés a szkriptekben

A *Now* vagy a *Time* függvények segítségével meghatározhatjuk egy szkript futási idejét úgy, hogy az elején és a végén egy-egy változóban eltároljuk az időpontokat, majd a két értéket kivonjuk egymásból. Ez azonban csak másodperc pontossággal tudja mérni az időtartamokat.

Pontosabb mérésekhez a *Timer* függvényt használjuk, amely az utolsó éjfél óta eltelt időt méri másodpercben, körülbelül huszad másodperc pontossággal.

A **6–42. példában** egy ciklus futási idejét határozzuk meg. Ne felejtjük el, hogy a Windows egyszerre több programot is futtat, és a szkriptünk csak időnként kapja meg a mikroprocesszort az utasítások végrehajtására. Ráadásul az indításhoz, az utasítások értelmezéséhez az interpreternek elég sok adminisztrációs tevékenységet kell végeznie, így a futási idő nem teljesen arányos a ciklusváltozó végértékével.

### A Területi beállítások módosítása

A dátumok használatánál gondot okozhatnak az eltérő területi beállítások. Nem célnünk ennek a kérdésnek a részletes taglalása, de megemlítenk néhány olyan függvényt, amely segíthet a megoldásban.

Ha a dátumok vagy időpontok egyes összetevőit külön szövegmezőkben kérjük be a felhasználótól, akkor a *DateSerial*, illetve *TimeSerial* függvényekkel készíthetünk belőlük megfelelő típusú értéket. Ezeket használjuk akkor is, ha az egyes összetevőket külön változókbán tároljuk:

```
DateSerial(2004, 5, 3) = #2004/05/03#  
TimeSerial(8, 11, 23) = #08:11:23#
```

Ezzel elkerüljük a különböző területi beállításokhoz tartozó sorrendből vagy elválasztójelből fakadó problémákat, viszont nem alkalmazhatjuk az *IsDate* függvényt, mert a *DateSerial* már előbb hibát fog jelezni.

Ha szükséges, a beállított nyelvterület kódját a *GetLocale* függvénnyel kérdezhetjük le:

```
Kod = GetLocale()
```

A környezetünkben előforduló leggyakoribb értékek és rövidítésük:

<i>Területi beállítás:</i>	<i>Decimális kód:</i>	<i>Rövidítés:</i>
magyar	1038	hu
amerikai angol	1033	en-us
brit angol	2057	en-gb
német	1031	de-de

Azt is megtehetjük, hogy a szkriptben megváltoztatjuk a területi beállítást a nekünk megfelelő értékre. Ezt a *SetLocale* függvénnyel érhetjük el, melynek paramétereként a decimális kód helyett a fenti rövidítést is megadhatjuk. A *SetLocale* az eredeti beállítással tér vissza, hogy félretehessük és visszaállíthassuk. A következő utasítás magyarra változtatja a területi beállítást:

```
EredetiBeallitas = SetLocale("hu")
```

A *SetLocale* csupán az adott dokumentumra érvényes, és nem változtatja meg az operációs rendszer beállításait. A függvények használatát a **6–43. példa** mutatja be.

A *SetLocale* és *GetLocale* függvényeket nem csak a dátumoknál, hanem a pénznem, tizedesjel és más hasonló beállításoknál is alkalmazhatjuk.

## 6.5. Változók az alprogramokban

Eddigi példáinkban sokszor készítettünk függvényeket és eljárásokat. Ezeknek az alprogramoknak adatokat adtunk át a paraméterek segítségével. A továbbiakban áttekintjük a paraméterek használatával kapcsolatos tudnivalókat.

Idézzük emlékezetünkbe, hogy az alprogram definíciójában a függvény vagy eljárás neve után zárójelben soroljuk fel a formális paramétereket. Ezeket az alprogram utasításai használják úgy, mintha változónevek lennének. A paraméterek felsorolása lokális deklarációnak számít.

A paraméterek helyére az alprogram hívásánál kifejezéseket írunk. A kifejezés az alprogram aktuális paramétere (argumentuma). Ez lehet egyszerűen egy konkrét érték, változóneve vagy függvényhívás. Az interpreter az alprogram utasításaiban a formális paraméter helyére az aktuális paramétert helyettesíti.

### Paraméterátadás cím és érték szerint

A paraméterek átadására kétféle módszert használhatunk. Amikor érték szerint adunk át egy paramétert, akkor az interpreter egy ideiglenes változót hoz létre, amelybe beírja az aktuális paraméternek az alprogram hívásakor meglévő értékét. Az alprogram az utasításokat az ideiglenes változóval hajtja végre. Ez a változó a többi lokális változóhoz hasonlóan eltűnik, ha kilépünk az alprogramból.

Érték szerinti paraméterátadás: az alprogram az aktuális paraméter értékét kapja meg egy ideiglenes változóban.

Ha az aktuális paraméter egyetlen változó, akkor is csak a másolata áll az alprogram rendelkezésére. Ebből következik, hogy ha az alprogram módosítja a paraméter értékét, akkor a másolat változik, nem pedig az eredeti változó értéke. A másolatot az alprogram végrehajtása után már nem használhatjuk fel.

A cím szerinti paraméterátadásnál az alprogram megkapja az aktuális paraméter tárolásának helyét (az úgynevezett címét) a memóriában, tehát az utasításokat az eredeti változóval hajtja végre. Így hozzáférhet a tartalmához, lehetősége van az aktuális paraméterként szereplő változó értékének végleges módosítására.

Cím szerinti paraméterátadás: az alprogram az aktuális paraméternek a memóriában elfoglalt helyét (címét) kapja meg.

Cím szerinti paraméterátadást csak akkor végezhetünk, ha az aktuális paraméter helyére egyetlen változónevet írunk. Ha az aktuális paraméter konstans vagy kifejezés, akkor csak érték szerinti paraméterátadás lehetséges. Az alprogram természetesen a kifejezésben szereplő változók értékét sem tudja módosítani.

Megjegyezzük, hogy az angol nyelvű megfelelője miatt a cím szerinti paraméterátadást néha referencia vagy hivatkozás szerintinek nevezik.

### A paraméterátadás módjának szabályozása

A paraméterátadás módját az alprogramok definíciójában szabhatjuk meg. A formális paraméter neve elé írt *ByVal* (by value, érték szerint) kulcsszó érték szerinti, a *ByRef* (by reference, hivatkozás szerint) pedig cím szerinti paraméterátadást ír elő.

A **6–44. példában** a következő eljárás definiáltuk:

```
Sub Modosit(ByRef P1, ByVal P2)
    P1 = "P1 új értéke"
    P2 = "P2 új értéke"
    window.alert("P1 = " & P1 & vbNewLine & _
        "P2 = " & P2)
End Sub
```

A program kiírja a változók értékét a

```
Call Modosit(A, B)
```

eljáráshívás előtt és után. Az eljárásban mindkét paraméter értéke megváltozott, de a módosítás csak a cím szerint átadott *A* változónál állandósult! Az eljárásból való kilépés után az érték szerint átadott *B* változó megtartotta eredeti értékét.

A **6–45. példában** az előző eljárást a

```
Call Modosit(A & " és " & B, B)
```

utasítás hívja meg. Mivel az első paraméter helyén egy kifejezés áll, az eljárás definíciójában szereplő *ByRef* kulcsszó ellenére érték szerinti paraméterátadás történik. A kifejezésben szereplő változók értéke nem módosul.

Ha a változónevet zárójelbe tesszük, akkor az interpreter kifejezésnek tekinti, így csak az értékét adja át az alprogramnak. Ebben az esetben az alprogram által végzett módosítások a kilépés után szintén eltűnnek, amit a **6–46. példa**



```
Call Modosit((A), B)
```

eljáráshívása mutat be. Az alprogram így nem tudja végérvényesen módosítani az A változó értékét.

### Az alapértelmezett paraméterátadási mód

Ha az alprogram definíciójában nem szabjuk meg a paraméterátadás módját, akkor a VBScript cím szerint végzi el. Így a

```
Sub Modosit(ByRef P1, ByVal P2)
```

definíció egyenértékű a

```
Sub Modosit(P1, ByVal P2)
```

definícióval. A *ByRef* kulcsszó tehát elhagyható, a *ByVal*-t azonban minden érték szerint átadásra kerülő paraméter elé ki kell írni a formális paraméterek felsorolásánál.

#### A 6–47. példában szereplő

```
Sub Csere(A, B)
    Dim Temp
    Temp = A
    A = B
    B = Temp
End Sub
```

eljárás felcseréli a változók értékét, pedig nem írtuk ki a *ByRef* kulcsszót a formális paraméterek elé. Javasoljuk az Olvasónak, hogy írja ki a *ByVal* kulcsszót valamelyik (vagy mindkét) formális paraméter elé! Így már nem hajtja végre az eljárás a változók cseréjét.

Eddigi példáinkban nem szabtuk meg a paraméterátadás módját, így az interpreter cím szerint adta át az alprogramoknak a paramétereket. Megjegyezzük, hogy több programozási nyelvben az érték szerinti paraméterátadás az alapértelmezett mód.

### A cím szerinti paraméterátadás használata

A cím szerinti paraméterátadás miatt az eljárásoknak is lehet „visszatérési érték”, pontosabban módosíthatják az aktuális paraméterként megadott változókat. Sok hiba forrása lehet, ha az alprogramok külön jelzés nélkül végzik ezt a módosítást. A *ByRef* kulcsszó alkalmazásával emeljük ki ezt a viselkedést! Így a 6–47. példa eljárásának definícióját célszerű a következő formában megadni:

```
Sub Csere(ByRef A, ByRef B)
```

Ez a definíció egyenértékű a fentivel, de felhívja a figyelmet arra, hogy módosítani fogjuk a paraméterek értékét.

Cím szerinti paraméterátadás esetén alkalmazzunk újabb változókat a módosított értékek visszaadására! Ha például egy (x, y) koordinátájú vektort szeretnénk a kétszeresére nyújtani, akkor használjuk a **6–48. példában** bemutatott eljárást:

```
Sub Nyujt(ByVal X_regi, ByVal Y_regi, ByRef X_uj, ByRef Y_uj)
    X_uj = 2 * X_regi
    Y_uj = 2 * Y_regi
End Sub
```

Ha akarjuk, az új koordinátákat beírhatjuk a régiéik helyére, de a definícióból világosan látszik, hogy mi fog történni a paraméterekkel.

Függvények esetén ne használjuk ki a cím szerinti paraméterátadás lehetőségét. A függvény csak egyetlen értéket adjon vissza, ne módosítsa változóinkat! Ha több változó módosítására van szükség, akkor készítsünk eljárást.

### Eljáráshívás és paraméterátadás

A VBScriptben kétféle, egymással egyenértékű módon lehet meghívni az eljárásokat. Leírhatjuk az eljárás nevét, utána zárójelek nélkül, vesszővel elválasztva felsoroljuk a formális paramétereket:

```
Eljárásnév Paraméter_1, Paraméter_2, ...
```

A *Csere* szubrutin hívása például:

```
Csere EgyikValtozo, MasikValtozo
```

A másik lehetőség a *Call* utasítás használata. Ebben az esetben a paramétereket zárójelbe kell tenni:

```
Call Eljárásnév(Paraméter_1, Paraméter_2, ...)
```

Például:

```
Call Csere(EgyikValtozo, MasikValtozo)
```

A két formát nem keverhetjük össze. Vagy kiírjuk a *Call*-t és a zárójeleket, vagy egyiket sem használjuk. Hibásak tehát az alábbi eljárás hívások:

```
Csere(EgyikValtozo, MasikValtozo)
Call EgyikValtozo, MasikValtozo
```

Mivel a programnyelvekben általános a paraméterek zárójelezése, több paraméter esetén mi is a zárójeles formát választottuk. Az egyetlen paraméterrel rendelkező eljárásoknál azonban nem írtuk ki a *Call* kulcsszót, de az általánosan használt szintaxis miatt zárójelbe tettük a paramétert. Ekkor ugyanis az interpreter a zárójel miatt kifejezésnek tekinti az eljárás neve után álló részt, így nem jelez szintaktikus hibát:

```
Eljárásnév(Paraméter)
```

Ne feledkezzünk meg arról, hogy kifejezések esetén az alprogram definíciójától függetlenül csak érték szerinti paraméterátadást végezhetünk. Ezért a fenti szintaxis használatakor az eredeti változó értékét nem tudjuk módosítani! Ezt a működést a **6–49. példa** szemlélteti.

Ha nem akarunk belebonyolódni a paraméterátadás eltérő módozataiba, akkor az eljárás hívásnál mindig írjuk ki a *Call* kulcsszót!

Megjegyezzük, hogy a *Call* kulcsszó kiírásával függvényeket is meghívhatunk önálló utasításként. Ebben az esetben a visszatérési értéket nem használjuk fel. Ezt a későbbiekben az *MsgBox* függvénynél alkalmazzuk.

### Függvényhívások a kifejezésekben

A cím szerinti paraméterátadás nem várt következményekkel járhat, ha a függvényhívást tartalmazó kifejezésben több helyen is szerepelnek a függvény aktuális paraméterei. A VBScript ugyanis a számítások hatékonyságának növelése érdekében a kiértékelés előtt átrendezi a kifejezéseket, ezért nem tudjuk, hogy a függvényhívás, és ezzel együtt az aktuális paraméterek értékének megváltoztatása mikor következik be.

Az *A* és a *B* pozitív egész számok legnagyobb közös osztóját a **6–50. példában** a következő algoritmussal számítjuk ki (az *A* a nagyobb):

```
Do
    Maradek = A Mod B
    A = B
    B = Maradek
Loop Until Maradek = 0
LegnagyobbKozosOsztos = A
```

Készítsünk ebből egy függvényt, amely meghatározza a két paraméter legnagyobb közös osztóját:

```
Function Lnko(A, B)
    ' a legnagyobb közös osztó meghatározása
End Function
```

Ha most a két szám közül az egyiket el akarjuk osztani a legnagyobb közös osztóval:

```
X = 24 : Y = 18                ' Lnko(24, 18) = 6
Hanyados = X / Lnko(X, Y)
```

akkor a balról jobbra történő kiértékelési szabály ellenére az interpreter először a függvényértéket határozza meg, így a hányados  $24/6 = 4$  helyett  $6/6 = 1$  lesz (a fenti algoritmus után az első paraméter értéke megegyezik a legnagyobb közös osztóval).

Arra sem mindig számíthatunk, hogy először a függvényértékek meghatározására kerül sor. Ha például a két számhoz hozzáadjuk a legnagyobb közös osztót:

```
Osszeg = X + Y + Lnko(X, Y)
```

akkor helyes eredményt kapunk. Erről a **6–51. példa** futtatásával győződhetünk meg.

Az ilyen nem kívánt mellékhatásokat úgy küszöbölhetjük ki, hogy az alprogramokon belül segédváltozókat vezetünk be, és nem módosítjuk az aktuális paraméterek értékét. Használhatunk egyszerűen érték szerinti paraméterátadást is:

```
Function Lnko(ByVal A, ByVal B)
    ' a legnagyobb közös osztó meghatározása
End Function
```

## Tömbök átadása a függvényeknek

Az alprogramok paraméterei között tömbök szintén szerepelhetnek. Ekkor különösen fontos az átadás módja. Érték szerinti paraméterátadásnál ugyanis még egyszer létrejön a tömb a memóriában! Ha nem határozzuk meg a paraméterátadás módját, akkor az a szabványos hívás esetén szerencsére cím szerint történik. Így az alprogram az eredeti tömb elemeivel dolgozik, akár meg is változtathatja azok értékét.

Ha egy egész tömböt szeretnénk átadni az alprogramnak, akkor a formális paraméter zárójelek és indexek nélkül szerepel a definícióban:

```
Function Függvénynév(Tömbnév)
```

A függvény utasításai között hivatkozhatunk a tömb elemeire, például:

```
X = Tömbnév(6)
```

Szükség esetén a tömb méretét az *UBound* függvénnyel határozhatjuk meg.

A függvény hívásánál az aktuális paraméter tömbjét ugyancsak zárójelek és indexek nélkül adjuk meg, például:

```
ElemekAtlaga = Atlag(TantargyJegyek)
```

A **6–52. példa** bemutatja ezt a függvényt, amely a paraméterként megadott tömb elemeit átlagolja. A függvénynek egy tömböt adunk át:

```
Function Atlag(Tomb)
```

Az átlagoláshoz először összegezzük az elemeket. Az összegezés algoritmusát már ismerjük. A ciklusváltozó felső határát, azaz a tömb maximális indexének értékét az *UBound* függvény szolgáltatja. Mivel a későbbiekben még szükség lesz rá, egy változóban tároljuk:

```
MaxIndex = UBound(Tomb)
Osszeg = 0
For I = 0 To MaxIndex
    Osszeg = Osszeg + Tomb(I)
Next
```

Az átlagot úgy kapjuk meg, hogy az összeget elosztjuk az elemek számával, ami eggyel nagyobb, mint a maximális index, hiszen a sorszámozás 0-tól indul:

```
Atlag = Osszeg / (MaxIndex + 1)
End Function
```

Mivel a függvény neve *Atlag*, ez az utasítás egyben megadja a visszatérési értéket is. Emlékeztetünk arra, hogy egy függvény utasításai között szerepelnie kell olyan értékadó utasításnak, melynek a bal oldalán a függvény neve áll a paraméterek nélkül.

A CD-mellékleten található példában véletlenszámokkal feltöltött tömböt használtunk, melynek feltöltését ciklussal végeztük. A ciklus összeállítja az elemek kiírásához szükséges sztringet is:

```
Lista = ""
For I = 0 to 9
    VeletlenSzam(I) = Int(100 * Rnd() + 1)
    Lista = Lista & VeletlenSzam(I) & "; "
Next
```

A tömbelemek átlagolásához meghívtuk az *Atlag* függvényt. Figyeljük meg, hogy az aktuális paraméterként szereplő tömb neve tetszőleges lehet, függetlenül a függvény definíciójában szereplő formális paramétertől! Az

```
Atlag(VeletlenSzam)
```

hatására az interpreter a számítást a *VeletlenSzam* tömb elemeivel végzi el.

Ha egy alprogram valamelyik aktuális paramétere egy tömb eleme, akkor a hívásnál egyszerűen megadjuk az elemet a paraméterek között. A derékszögű háromszög átfogóját meghatározó függvényünket például a következőképpen hívhatjuk, ha a befogókat az *A* és *B* tömbökben tároljuk:

```
Atfogo(A(I), B(I))
```

Ebben az esetben ugyanúgy használhatjuk a cím és az érték szerinti paraméterátadást, mint az egyszerű változóknál.

### Tömbök átadása az eljárásoknak

A függvényeknél elmondottak az eljárásokra is érvényesek. Ha az eljárásban meg akarjuk változtatni az eredeti tömb elemeit, akkor feltétlenül cím szerinti paraméterátadást kell alkalmaznunk. Az érték szerinti paraméterátadásnál létrejön a tömb másolata, és az eljárás utasításai erre a másolatra vonatkoznak.

Emlékezzünk vissza arra, hogy a változók értékének végleges megváltoztatására csak szabványos hívás, például a *Call* kulcsszó és zárójelek használatával van lehetőség. Ha a

```
Sub Rendez (Tomb)
...
End Sub
```

eljárás rendezzi a paraméterként megadott tömb elemeit, akkor a

```
Rendez (Tomb)
```

eljáráshívás még a `Sub Rendez (ByRef Tomb)` definíció esetén sem változtatja meg a tömbelemek eredeti sorrendjét. Az interpreter a hívásban szereplő zárójel (mint kifejezés) használata miatt létrehozza, majd rendezzi a tömb másodpéldányát. Ez azonban a kilépés után törlődik. A

```
Call Rendez (Tomb)
```

hívás következtében már rendezetté válik az eredeti tömb. Ugyancsak rendezzi a tömböt a zárójelek és a *Call* nélkül történő `Rendez Tomb` eljáráshívás is.

Az egyes hívások eredményét a **6–53. példa** szemlélteti.



## 7. ESEMÉNYKEZELÉS

A moduláris programozás fontos eszköze az események kezelése. A programot események segítségével vezéreljük, események indítják el az eljárások legnagyobb részét és a számításokat.

Már a könyv első részében megismerkedtünk a dokumentum objektummodell néhány eseményével, és az eseménykezelő eljárások készítésének módszereivel. Ebben a fejezetben főleg az események tulajdonságait hordozó *event*-objektummal foglalkozunk.

### 7.1. Az event-objektum

#### Az esemény modell

Az objektumokat érő hatásokat eseményeknek nevezzük. Események jönnek létre, ha a felhasználó egér- illetve billentyűzet-műveletet végez. Eseményeket válthatnak ki más perifériák, vagy az operációs rendszer belső objektumai is. A programunk szintén létrehozhat eseményeket, ha megváltoztatja egy szövegmező tartalmát, vagy bezárja az ablakot.

Egy esemény bekövetkezésekor a Windows üzenetet állít elő, amely tartalmazza az esemény jellemzőit (azonosítóját, az egér pozícióját, a lenyomott billentyű kódját stb.). Az üzeneteket az operációs rendszer osztja szét a futó programok<sup>25</sup> között. A programok üzenetkezelő ciklusa kiolvassa a sorban álló üzeneteket, majd továbbítja a megfelelő ablakhoz. Az üzeneteket az ablakkezelő függvény dolgozza fel. A böngésző ablakkezelő függvénye ellenőrzi, hogy létezik-e az adott eseményhez tartozó eseménykezelő eljárás. Ha igen, akkor utasítja az interpretert az eljárás végrehajtására.

Az esemény bekövetkezésekor létrejövő üzeneteket az *event* (esemény) objektum segítségével érhetjük el, melynek tulajdonságai hordozzák az eseményre vonatkozó információkat.

A tulajdonságokra való hivatkozásnál az eseményobjektumot mindig minősíteni kell a *window* előtaggal:

```
window.event.tulajdonságnév
```

Az *event*-objektum az eseménykezelés befejezése után eltűnik, így a tulajdonságaira a továbbiakban már nem hivatkozhatunk.

Az *event*-objektum nem rendelkezik metódusokkal.

#### Az event-objektum tulajdonságai

Az eseményobjektum tulajdonságai közül néhány csak speciális eseményeknél, például egér- vagy billentyűzet-műveleteknél kap értéket. Több tulajdonságot azonban minden eseménynél felhasználhatunk.

---

<sup>25</sup> Pontosabban a szálak (thread) között.

A *type* (típus) sztring tárolja az esemény nevét, de az *on* előtag nélkül, például: *click*. Az *srcElement* (source element, forrás-elem) megadja azt a DHTML-objektumot, amelyre az esemény vonatkozott. Segítségével elérhetjük az objektum tulajdonságait, illetve metódusait is.

Az objektum osztályát, azonosítóját és az esemény nevét például a következőképpen jeleníthetjük meg:

```
window.alert(window.event.srcElement.tagName & _  
              window.event.srcElement.id  
              window.event.type)
```

A **7–1. példában** az *onclick* eseménykezelőjét a *document*-objektumhoz rendeltük hozzá. Így egy eljárásban több objektumot is kezelhetünk, az *srcElement* segítségével pedig megtudhatjuk, hogy melyik objektumra vonatkozott az esemény.

Alakítsuk át az 5–17. példa szavakat színező programját úgy, hogy csak azt a kezdő szót színezzé ki, amire rákattintunk! A színt tízféle lehetőségből, véletlenszerűen választjuk ki. A színek angol elnevezését egy tömbben tároljuk.

A *document*-objektum *onclick* eseménykezelőjében megvizsgáljuk, hogy SPAN-objektumra kattintottunk-e. Ha igen, akkor az *Rnd* függvény segítségével megváltoztatjuk a színét. A hivatkozás egyszerűsítéséhez a **7–2. példában** objektumváltozót használunk:

```
Set Elem = window.event.srcElement  
If Elem.tagName = "SPAN" Then  
    Elem.style.color = Szin(Int(9 * Rnd()))  
End If
```

Még egyszer hangsúlyozzuk, hogy az *event*-objektum csak addig létezik, amíg az eseménybuborék terjedése le nem állt. Utána már hibaüzenetet kapunk, ha hivatkozunk rá.

## A szülőelem meghatározása

A DHTML-objektumok tartalmazhatják egymást. Előfordulhat, hogy arra az objektumra szeretnénk hivatkozni, amely az esemény által kijelölt objektumot tárolja. Ekkor a *parentElement* (szülő elem) tulajdonságot használjuk. A *parentElement* megadja azt az objektumot amely az eseménybuborékot indító objektumot tartalmazza az objektum-hierarchiában.

A **7–3. példában** az előző szkriptet úgy alakítjuk át, hogy egérkattintás esetén ne csak az első szót, hanem a teljes versszakot színezzé ki. A versszakokat bekezdésobjektumokba tettük, így a SPAN-objektumok szülőelemei a P-objektumok:

```
<P>  
    <SPAN>Hull</SPAN> a szilva a fáról,<BR>  
    ...  
</P>
```

A *document\_onclick* eseménykezelőben megvizsgáljuk, hogy az *srcElement* szülő-eleme bekezdésobjektum-e. Ha igen, akkor kiszínezzük:



```
Set Elem = window.event.srcElement
If Elem.parentElement.tagName = "P" Then
    Elem.parentElement.style.color = Szin(Int(9 * Rnd()))
End If
```

Ezzel elértük, hogy az első szóra kell kattintani, de a bekezdést színezzük ki. Bonyolultabb esetekben ellenőrizzük azt is, hogy az esemény által kijelölt objektum SPAN elem-e!

A *parentElement* segítségével tovább haladhatunk felfelé az objektum-hierarchiában. A legutolsó szinten *Nothing* lesz az értéke, ekkor már nem lehet feljebb lépni. A **7–4. példában** egy ciklus segítségével megjelenítjük az objektumok sorozatát attól kezdve, amelyre rákattintunk:

```
Set Elem = window.event.srcElement
Do Until Elem Is Nothing
    Lista = Elem.tagName & vbNewLine & Lista
    Set Elem = Elem.parentElement
Loop
```

## Eseménykezelő eljárások

A böngésző egy dokumentum betöltésekor regisztrálja, hogy az objektumok milyen eseménykezelő eljárásokkal rendelkeznek. Az esemény bekövetkeztekor végrehajtódik a megfelelő alprogram.

A 3.5. fejezetben részletesen ismertettük az eseménykezelő eljárások típusait. Javasoljuk az Olvasónak, hogy a 3–43. példa segítségével ismétlje át mind a négyféle lehetőséget! Itt néhány további tudnivalóval egészítjük ki az ismereteket:

- Ha az esemény a dokumentum betöltése közben következik be, akkor inline eseménykezelőt kell alkalmaznunk!
- Ha az eseménykezelőnél megadott azonosítóval több objektum is rendelkezik, akkor közülük csak az utolsónál bekövetkező esemény fogja elindítani az eljárást.
- Ha egy objektumnál ugyanarra az eseményre több eseménykezelő eljárást is definiáltunk, akkor a legalacsonyabb precedenciájú az inline, aztán következik az eseménykezelő szkript (*for* és *event* tulajdonságokkal), a legerősebb pedig az eseménykezelő szubrutin (*Sub ObjektumNév\_eseménynév*). Az egyforma precedenciájúak közül az érvényesül, amelyiket utoljára dolgozta fel a böngésző a dokumentum betöltése során.

Az eseménykezelő eljárásoknál nem használunk paramétereket. Néhány esemény azonban átad a szubrutinnak adatokat. Erre a későbbiekben látunk példát.

Az eseménykezelő eljárásokban gyakran hivatkozunk az eljárást hívó objektumra. Ezt a *Me* kulcsszóval tehetjük meg. A *Me* nem azt az objektumot jelöli, amelyre az esemény vonatkozott, hanem azt, amelyik az eseménybuborék terjedése során meghívta az eseménykezelőt.

A **7–5. példában** az első bekezdés *onclick* eseménykezelője kiírja az *srcElement* és a *Me*-objektumok osztályát. Ha a dőlt betűvel írt *Me* szóra kattintunk, akkor az

*srcElement* a dőlt betűs objektum, de a *Me* kulcsszó a bekezdésobjektumot jelenti, mert az aktivizálta az eseménykezelőt.

### Eseménykezelő eljárások utólagos hozzárendelése az objektumokhoz

Ha az objektum eredetileg nem rendelkezett eseménykezelő eljárással, akkor azt utólag is hozzá lehet rendelni vagy meg lehet változtatni. A hozzárendelést a *Set* utasítással és a *GetRef* függvénnyel végezzük el:

```
Set ObjektumAzonosító.eseménynév = GetRef("Eljárásnév")
```

Az eljárás nevét sztringként, zárójelek nélkül adjuk meg.

A *GetRef* paramétere kifejezés is lehet, ami egy eljárás nevével megegyező karakterláncot eredményez. Ezzel nagyon rugalmas lehetőségek birtokába jutunk, a program futása közben állíthatjuk elő a megfelelő eljárás nevét. Akár magát a szubrutint is létrehozhatjuk egy szkripttel, de a hozzárendeléskor már léteznie kell.

A **7–6. példában** a parancsgombra kattintás eseménykezelője változtatja a meghívott eljárást, amire a háttérszín változásából következtethetünk. Az eseménykezelőt először inline módon rendeljük a parancsgombhoz a HTML-kódban:

```
onclick = "Piros() "
```

A *Piros* eljárásban a háttérszín beállítása után megváltoztatjuk az eseménykezelő eljárást:

```
Set Valt.onclick = GetRef("Sarga")
```

A *Sarga* eljárásban pedig visszaállítjuk a *Piros*-ra:

```
Set Valt.onclick = GetRef("Piros")
```

A *GetRef* függvény a memóriában tárolt eljárás címét határozza meg, és adja át az interpreternek, amely a többi eseménykezelőhöz hasonlóan feljegyzi a regisztrációs táblájába. Így olyan objektumokhoz is rendelhetünk eseménykezelést, amelyek a dokumentum betöltésekor nem léteztek, például egy szkript hozza őket létre.

A **7–7. példa** kódja a Megjelenít parancsgombra való kattintáskor megjelenít egy szövegmezőt és egy újabb parancsgombot. A parancsgomb az *onclick* eseménykezelőben „megfordítja” szövegmezőbe írt karaktersorozatot.

A szövegmező és a parancsgomb HTML-kódjának karakterláncát egy változóban állítjuk elő:

```
MezoEsGomb = "<INPUT id = 'Szoveg' type = 'text'>" & _  
" <INPUT id = 'Fordit' type = 'button' value = 'Fordítás'>"
```

Ezzel egyszerűsítjük a Megjelenít gomb eseménykezelő kódját, amelyben először ki tesszük az újabb objektumokat a weblapra:

```
Tarolo.innerHTML = MezoEsGomb
```

majd a *GetRef* függvénnyel hozzárendeljük az eseménykezelő eljárást:

```
Set Fordit.onclick = GetRef("Fordit_onclick")
```

Az eljárás hiába van benne a szkriptben, a dokumentum betöltésekor a böngésző nem tudja hozzárendelni a *Fordit* parancsgombhoz, mert ez az objektum még nem létezik.

### A GetRef függvény használata

A *GetRef* függvényt használjuk akkor is, ha több objektumhoz szeretnénk ugyanazt az eseménykezelő eljárást hozzárendelni, de nem akarjuk mindegyiket külön leírni. A **7–8. példában** létrehozunk 5 parancsgombot. Mindegyikük a feliratának megfelelő színűre változtatja a dokumentum háttérét.

A parancsgombokat a BODY-ba helyezett ciklussal készítjük el. Létrehozunk egy *szin* nevű tulajdonságot is, amely a szín angol nevét tárolja. Ezt használjuk fel a háttérszín változtatásánál:

```
Gombok = ""
For I = 0 to 4
    Gombok = Gombok & "<INPUT id = 'Szinez' type = 'button' " & _
        "szin = ' ' style = 'width: 80'> "
Next
GombSor.innerHTML = Gombok
```

A *szin* értékét hozzáfűzhattük volna a sztringhez, de egyszerűbb, ha egy újabb ciklussal állítjuk be az *AngolSzin* tömb elemeinek felhasználásával. A gombok feliratát is ez a ciklus adja meg a *MagyarSzin* tömb segítségével. Végül kijelöli az *onclick* eseménykezelő eljárást. Emlékezzünk vissza arra, hogy az azonos nevű objektumokat indexelhetjük, így különböztetve meg őket egymástól:

```
For I = 0 To 4
    Szinez(I).value = MagyarSzin(I)
    Szinez(I).szin = AngolSzin(I)
    Set Szinez(I).onclick = GetRef("Fest")
Next
```

A *Fest* eljárásban egyszerűen hozzárendeljük a háttérszínhez a parancsgomb *szin* tulajdonságát. Az eseménykezelőt meghívó objektumra a *Me* kulcsszóval hivatkozunk:

```
Sub Fest
    document.bgColor = Me.szin
End Sub
```

Ezzel a módszerrel ciklusba foglalhatjuk az egymáshoz hasonló DHTML-objektumok létrehozását, tulajdonságaik beállítását, eseménykezelők hozzárendelését. A ciklus használata sokkal egyszerűbb, mintha egyesével írnánk bele őket a HTML-kódba.

### Az eseménybuborék

Gyakran előfordul, hogy egy esemény több objektumhoz is kapcsolható. Egy SPAN-objektum például benne lehet egy bekezdésobjektumban, a bekezdésobjektumot a BODY, azt pedig a *document* tartalmazza. Ilyenkor az objektumok egymásnak adják át az eseményt, belülről kifelé haladva objektumhierarchiában. Ezt a folyamatot eseménybuboréknak nevezzük.

Mint tudjuk, az eseménybuborék terjedése a következő utasítással állítható le:

```
window.event.cancelBubble = True
```

Az eseménykezelések végén a böngésző kapja meg az eseményt akkor is, ha az eseménybuborék terjedését leállítottuk. Bizonyos eseményekhez van alapértelmezés szerinti eljárása, másokhoz pedig nincs. A böngésző alapértelmezés szerinti eseménykezelését a *returnValue* tulajdonság hamisra állításával tilthatjuk le:

```
window.event.returnValue = False
```

Ezzel azt jelezzük a böngészőnek, hogy mi már kezeltük az eseményt, nem kell foglalkoznia vele.

Figyeljünk a *cancelBubble* és a *returnValue* közti különbségre! A *cancelBubble* a dokumentum objektumainak szól, a többi objektum már nem kapja meg az eseményt. A *returnValue* a böngészőnek szóló beállítás.

A két tulajdonság közti különbséget a **7–9. példa** mutatja be. A weblapon látható két hivatkozás egy-egy példa fájljait tölti be. Mindkettőt bekezdésobjektumokba helyeztük. A hivatkozásokhoz és a bekezdésekhez is készítettünk eseménykezelő eljárást. A 7-01. példára mutató hivatkozás eseménykezelőjében nem szakítjuk meg az eseménybuborék terjedését (alapértelmezés), de nem adjuk át az eseményt a böngészőnek:

```
window.event.cancelBubble = False  
window.event.returnValue = False
```

Ezért működésbe lép a bekezdés eseménykezelője (megjelenik a párbeszédablak), a böngésző viszont nem tölti be az újabb dokumentumot.

A 7-02. példára mutató hivatkozás eseménykezelőjében éppen fordítva:

```
window.event.cancelBubble = True  
window.event.returnValue = True
```

megszakítottuk az eseménybuborék terjedését, de engedélyeztük a böngésző eseménykezelését (alapértelmezés). Ezért nem jelent meg a bekezdésobjektum eseménykezelője által küldött üzenet, a böngésző viszont betöltötte az újabb weblapot.

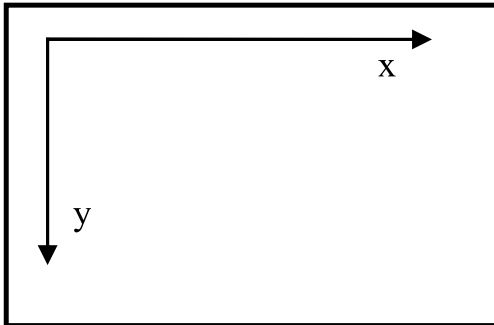
Az alapértelmezett értékeket nem kell megadni. Figyeljünk arra, hogy az eseménybuboréknál a megszakítást állítjuk be (a *False* engedélyezi a folytatást), a böngészőnél viszont a folytatást (a *True* engedélyezi az eseménykezelést)!

Néhány esemény nem indít el eseménybuborékot, velük majd később találkozunk.

## 7.2. Egér-események

A weblapok szkriptjeiben főleg az egérrel és a billentyűzettel kapcsolatos események kezelését kell elvégeznünk. A továbbiakban ezeket tekintjük át az *event*-objektum kapcsolódó tulajdonságaival együtt.

### Az egér mozgatása



7-1. ábra. Az ablak koordináta-rendszere

Az egér mozgatása egy objektum fölött az *onmousemove* esemény bekövetkezését okozza. Az *event*-objektum *x* és *y* tulajdonsága tárolja az egérmutató helyzetét az ablak belsejének bal felső sarkához viszonyítva, pixelben kifejezve. Az *x*-et vízszintesen mérjük balról jobbra, az *y*-t függőlegesen, felülről lefelé. A bal felső sarokban mindkét koordináta értéke 0.

Ha a tulajdonságok lekérdezésekor az egérmutató már elhagyta az objektum területét, akkor az *x*, illetve az *y* értéke negatív.

A **7-10. példa** kijelzi az egér pozícióját az állapotsorban, ha a megjelenített táblázat fölött helyezkedik el a mutató:

```
Sub Tablazat_onmousemove
    window.status = "x = " & window.event.x & _
                    "; y = " & window.event.y
End Sub
```

Figyeljük meg, hogy az állapotsorban csak akkor változnak a koordináták, ha a táblázat fölött mozgatjuk az egeret, hiszen a *TABLE*-objektumhoz rendeltük hozzá az eseménykezelést! Ha a *document*-objektumot használjuk, akkor már az egész dokumentum fölött jelzi a pozíciót (**7-11. példa**):

```
Sub document_onmousemove
```

Szükség esetén a képernyő bal felső sarkához is viszonyíthatjuk a koordinátákat, ebben az esetben a *screenX*, *screenY* tulajdonságokat használjuk (**7-12. példa**):

```
window.status = "x = " & window.event.screenX & _
                "; y = " & window.event.screenY
```

A példa betöltése után állítsuk kisebb méretűre a böngésző ablakát, majd helyezzük máshová a képernyőn, és így figyeljük meg az állapotsorban megjelenő koordinátákat!

Előfordul, hogy az eseményt fogadó objektumhoz akarjuk viszonyítani a koordinátákat, ekkor az *offsetX*, *offsetY* tulajdonságokat használjuk (**7-13. példa**):

```
Sub Tablazat_onmousemove
    window.status = "x = " & window.event.offsetX & _
                    "; y = " & window.event.offsetY
End Sub
```

A **7–14. példában** a táblázathoz és a sárgával jelölt cellához is készítettünk az egér pozícióját jelző eseménykezelő eljárást. A cella fölött mozgatva az egeret, a cella bal felső sarkához viszonyított koordináták jelennek meg az állapotsorban. Ehhez a cella eseménykezelő eljárásában le kell tiltatni az eseménybuborék továbbterjedését, különben megkapná a táblázat eseménykezelője is, és átírná a cella bal felső sarkához viszonyított koordinátákat a táblázat bal felső sarkához viszonyított koordinátákra!

Megjegyezzük, hogy a dokumentum objektumainak speciális elhelyezési lehetőségei (relatív pozicionálás) módosíthatják az *offsetX* és *offsetY* tulajdonságok jelentését, de ezt nem fogjuk használni.

Az egér mozgatása nem csak a böngészőnek, hanem a Windowsnak is munkát ad, hiszen létre kell hozni az eseményeket közvetítő üzeneteket, mozgatni kell a mutatót a képernyőn. Ezért, ha várakozás közben megszokásból tologatjuk az egeret, akkor az eseménykezeléssel folyamatosan megszakítjuk az operációs rendszer működését, így megnöveljük annak a tevékenységnek az időtartamát, aminek a végrehajtására éppen várunk!

### Az onmouseover és onmouseout események

Ha az egér egy objektum fölé ér, akkor bekövetkezik az *onmouseover* esemény (ehhez nem kell figyelemmel kísérnünk a koordinátáit). A parancsgombok színének változtatásával például kiemelhetjük az egérmutató alatt lévő gombot:

```
Sub Gomb_onmouseover
    Gomb.style.backgroundColor = "pink"
End Sub
```

A CD-n lévő **7–15. példában** több parancsgombot is elhelyeztünk a weblapon, ezért az eseménykezelőt a *document*-objektumhoz rendeltük. A megfelelő parancsgombot az *event*-objektum *srcElement* tulajdonságával választottuk ki.

Mint a példa kipróbálása során láthatjuk, a gombok színe megváltozik az egér hatására, de úgy is marad! A visszaállítást az *onmouseout* esemény segítségével végezzük el. Ez akkor következik be, ha az egérrel elhagyjuk egy objektum területét.

A **7–16. példában** a visszaállításhoz először egy globális változóban tároljuk az eredeti színt:

```
Dim EredetiSzin
Sub window_onload
    EredetiSzin = Gomb(0).style.backgroundColor
End Sub
```

amit *onmouseover* eseménynél megváltoztattunk, majd az *onmouseout* esemény bekövetkezésénél visszaállítunk:

```
Sub document_onmouseout
    Dim Elem
    Set Elem = window.event.srcElement
    If Elem.id = "Gomb" Then
        Elem.style.backgroundColor = EredetiSzin
    End If
End Sub
```

A **7–17. példában** egy táblázat celláit emeljük ki hasonló módon. Mivel itt tudtuk az eredeti színt (white, fehér), ezért nem mentettük el.

Ne felejtjük el, ha csak egyetlen objektumnál akarjuk felhasználni az *onmouseover* és *onmouseout* eseményeket, akkor nincs szükség az *srcElement* tulajdonságra! Ekkor *Gomb\_onmouseover* fenti forráskódjához hasonlóan közvetlenül is hivatkozhatunk az objektum azonosítójára.

Az egérmozgásra vonatkozó események szoros kapcsolatban vannak egymással. Az *onmouseover* és *onmouseout* események közé mindig beékelődik egy *onmousemove* esemény.

### Kattintás az egérgombbal

Az egyik leggyakrabban felhasznált eseménnyel már a könyv elején megismerkedtünk. Az *onclick* bekövetkezik, ha

- lenyomjuk a bal oldali egérgombot,
- egy fókuszban lévő parancsgomb esetén lenyomjuk az Enter vagy a szóköz billentyűt,
- lenyomunk egy objektumhoz kapcsolt billentyűparancsot (általában Alt + betű).

Az űrlapoknál még további eseményekkel is találkozni fogunk, melyek egyenértékűek az *onclick* eseménnyel.

Az egérgomb lenyomását természetesen követnie kell a felengedésének, egyébként más funkciót jelentene az operációs rendszer számára (például egyes objektumok mozgatását). Ha egy objektumon lenyomjuk az egérgombot, és a felengedése nélkül elvisszük róla az egérmutatót, akkor nem következik be az *onclick* esemény!

A gomb lenyomását és felengedését külön is jelzi az *onmousedown* és *onmouseup* esemény. Az események bekövetkezésének sorrendje: *onmousedown*, *onmouseup*, *onclick*.

A **7–18. példában** mindhárom eseményhez írtunk eseménykezelőt, amit a *document*-objektumhoz rendeltünk hozzá. Kipróbálásához nyomjuk le a bal oldali egérgombot, és egy darabig tartsuk lenyomva. Amíg nem engedjük fel, addig látható az *onmousedown* eseményhez rendelt piros háttér, majd a felengedésnél az *onmouseup* a szín átváltja sárgára, és megjelenik az *onclick* eseményt jelző párbeszédablak.

Az *onmousedown* és az *onmouseup* események nem kötődnek a bal oldali egérgombhoz, bármelyik gomb lenyomása esetén bekövetkeznek. A középső vagy a jobb oldali gomb használatánál azonban nem jelenik meg az *onclick*-et jelző párbeszédablak. A jobb oldali egérgombnál a böngésző még megjeleníti a szokásos felbukkanó menüt is.

A dupla kattintást *ondblclick* esemény jelzi. A dupla kattintás hatására bekövetkező események sora:

*onmousedown*, *onmouseup*, *onclick*, *onmouseup*, *ondblclick*

A CD-melléklet **7–19. példája** minden egyes eseménynél kiírja az esemény nevét. Próbáljuk ki a többi egérgombot is. Figyeljük meg, hogy az *onclick* és az *ondblclick* események csak a bal egérgomb használata esetén következnek be, a többi esemény viszont bármely egérgomb lenyomásakor létrejön!

### Az egérgombok megkülönböztetése

Az egérgombok megkülönböztetéséhez az *event*-objektum *button* (gomb) tulajdonságát használjuk, melynek lehetésetes értékei:

0:	egyik sem	2:	jobb egérgomb
1:	bal egérgomb	4:	középső egérgomb

Az események közül az *onmousedown*, az *onmouseup* és az *onmousemove* okozza a tulajdonság beállítását. A többi eseménynél (beleértve a nem egér-eseményeket is) a tulajdonság értéke 0.

A **7–20. példa** megjeleníti a bekövetkező egér-eseményeket. Minden esemény után zárójelben jelzi az egérgomb sorszámát. Az *onclick* és az *ondblclick* esetén 0-t kapunk, mert ekkor nem adódik át a gomb sorszáma!

A futtatásnál próbáljunk egyszerre több gombot is lenyomni (ezt legkönnyebben az egér mozgatásakor tudjuk megtenni). Ekkor az egérgomboknak megfelelő fenti sorszámok összegét kapjuk meg, a jobb és a bal egérgomb lenyomásánál például  $1 + 2 = 3$ -at. Könnyű belátni, hogy az összegből mindig meghatározhatjuk, mely gombok voltak lenyomva.

Itt említjük meg, hogy a középső egérgomb forgatása az *onmousewheel* esemény bekövetkezését okozza. Az eseményobjektum *wheelDelta* tulajdonsága a léptetés 120-szorosát adja meg. Az esemény használatát a **7–21. példa** segítségével tanulmányozhatjuk.

### Az egér-események használata

Az egér-események segítségével már sokkal látványosabban módosíthatjuk egy táblázat adatait. A **7–22. példában** kiegészítjük az 5–29. példát az adatok változtatásának lehetőségével. Ha egy cellára duplán kattintunk, akkor megjelenik benne egy szövegmező és egy Tárol feliratú parancsgomb. Ha a felhasználó a gombra kattint, az eseménykezelő eljárás a szövegmezőbe írt karaktersorozatot beírja a cellába, és az adatokat tároló tömb megfelelő elemébe.

A HTML-kódban a táblázat celláit két egymásba ágyazott ciklussal hoztuk létre. Az egyszerűbb hivatkozás érdekében a cellákat ellátjuk a sort és az oszlopot meghatározó tulajdonságokkal:

```
document.write("<TD sor = " & I & " oszlop = " & J & ">")
```

Elkészítünk egy sztringet, amit a dupla kattintásnál beleírunk a cellába:

```
Bevitel = "<INPUT id = 'AdatBe' type = 'text' size = 15>" &  
"&nbsp;&nbsp;&nbsp;<INPUT id = 'Tarol' type = 'button' value = 'Tárol'>"
```



### A szövegmező megjelenítése

A táblázat *ondblclick* eseménykezelőjében meg kell vizsgálnunk, hogy adatot tartalmazó cellára történt-e a dupla kattintás. Ehhez a *getAttribute* tulajdonságot használjuk fel, mert az első sor (a címsor) objektumai nem kaptak *sor* (és *oszlop*) tulajdonságot. Ha az eseményhez kapcsolódó objektumnak nincs *sor* tulajdonsága, akkor hibaüzenetet jelenítünk meg:

```
If IsNull(window.event.srcElement.getAttribute("sor")) Then
    window.alert("A cella nem módosítható!")
```

Emlékeztetünk arra, hogy ha egy tulajdonság nem létezik, akkor a *getAttribute* metódus *Null* értéket ad vissza.

Vegyük észre, hogy itt erősen kihasználtuk az eseménybuborék terjedését! A dupla kattintás ugyanis a cellán történik, de annak nincsen eseménykezelője (nem akartuk minden cellához hozzárendelni). Ezért továbbadja a cellát tartalmazó sorobjektumnak, az pedig a táblázatobjektumnak.

Ha adatot tartalmazó cellára kattintunk, akkor globális változókbán félretesszük a cellaobjektumra való hivatkozást, továbbá a sor és az oszlop számát, mert ezekre a módosításnál szükségünk lesz:

```
Else
    Set Objektum = window.event.srcElement
    I = Objektum.sor
    J = Objektum.oszlop
```

Ezt követően megjelenítjük a cellában a szövegmezőt és a parancsgombot, végül ráállítjuk a fókuszt a szövegmezőre:

```
Objektum.innerHTML = Bevitel
AdatBe.focus()
End If
```

### Az adatok tárolása

Az új adat tárolásához a *Tarol* parancsgomb *onclick* eseménykezelőjét kell elkészíteni. Mivel ez az objektum a weblap betöltésekor még nem létezik, a böngésző nem tudja hozzárendelni az eseménykezelőt. (Emlékezzünk vissza arra, hogy az eseménykezelő eljárások regisztrálása a betöltéskor megy végbe!) Ezért a *document*-objektumot használjuk, és megvizsgáljuk, hogy az esemény a *Tarol*-objektumra vonatkozik-e. Ha igen, akkor a szövegmező tartalmát a tömb megfelelő elemébe másoljuk, megjelenítjük a cellában, és felszabadítjuk az objektumváltozót. Ha a felhasználó üresen hagyta a szövegmezőt, akkor az Internet Explorer nem jeleníti meg a cella keretét, ezért ilyenkor egy nem törhető szóközt írunk a cellába:

```
If window.event.srcElement.id = "Tarol" Then
    Adat = AdatBe.value
    A(I, J) = Adat
    If Adat = "" Then
        Adat = "&nbsp;"
    End If
```

```
Objektum.innerHTML = Adat
Set Objektum = Nothing
End If
```

Nagyon fontos, hogy az *Objektum*, az *I* és a *J* változók globálisak legyenek, mert az *ondblclick* eljárásban kapnak értéket, amit az *onclick* eljárásban használunk fel. Az eljárások lokális változói törlődnek az eljárás befejezése után!

### *Másik cella módosításának letiltása*

A felhasználó megteheti, hogy mielőtt rákattintana a Tárol gombra, egy másik cellában is megjeleníti a szövegmezőt. Programunk ezt az esetet nem tudja kezelni, ezért a tárolás végrehajtása előtt bekövetkező újabb dupla kattintásnál hibaüzenetet adunk. Ehhez bevezetünk egy globális logikai változót, amellyel figyeljük a szövegmező megjelenítését. A weblap betöltésekor még nincs szövegmező, ezért a változó először *False* értéket kap:

```
Tilt = False
```

Az *ondblclick* eseménykezelőben a szövegmező megjelenítése után értékét *True*-ra állítjuk:

```
Objektum.innerHTML = Bevitel
Tilt = True
```

Az *onclick* eseménykezelőben pedig – az adattárolás és megjelenítés után – visszaállítjuk *False*-ra:

```
Objektum.innerHTML = Adat
Set Objektum = Nothing
Tilt = False
```

A *Tilt* segítségével az *ondblclick* eseménykezelő elején megvizsgálhatjuk, hogy van-e már szövegmező valamelyik cellában. Ha igen, akkor hibaüzenetet adunk, és befejezzük az eljárás végrehajtását:

```
If Tilt Then
    window.alert("Először kattintson a Tárol gombra!")
Exit Sub
End If
```

Ezzel a feladatot megoldottuk. Két rövid eseménykezelő eljárás segítségével nagyon látványos adatmódosítási lehetőséget hoztunk létre. A teljes kódot a CD-n lévő fájl tartalmazza. A megoldásban kissé zavaró, hogy a szövegmező mérete miatt a megjelenítéskor „ugrál” a táblázat. Ezen a cellák szélességének előírásával segíthetünk. Valóságos esetben természetesen mindig ellenőriznünk kell a begépelt adatokat a tárolás előtt.

Az eseménybuborék terjedését a fentiekhez hasonlóan akkor is kihasználhatjuk, ha nem táblázatban helyezkednek el azok az elemek, amelyekhez közös eseménykezelőt akarunk készíteni. Más konténer hiányában egy DIV-objektumba tehetjük az elemeket, és a DIV-hez rendelhetjük az eseménykezelést. Az eseményhez kapcsolódó objektumot az *event* tulajdonságainak a segítségével tudjuk meghatározni.

Eddigi ismereteink alapján készült a **7–23. példa** memóriajátéka. Javasoljuk az Olvasónak, hogy alaposan tanulmányozza a forráskódot!

### 7.3. Billentyűzet-események

A billentyűzet eseményeit főleg adatbevitelnél, a szövegmezőkbe írt értékek ellenőrzésénél használjuk fel.

#### Billentyűzet-események

A billentyűzet által kiváltott események közül már ismerjük az *onkeydown*-t, amely egy billentyű lenyomásakor jön létre. A billentyű felengedése az *onkeyup* eseményt váltja ki. A **7–24. példa** a háttérszín váltogatásával jelzi az események bekövetkezését. Figyeljük meg, hogy bármely billentyű lenyomására reagál (néhány billentyűnek egyéb hatása is van)!

A karakterek beírását végző billentyűk esetén a két esemény közé még az *onkeypress* is beékelődik. Az *onkeypress*-t az Enter szintén kiváltja, a Shift vagy az Alt azonban már nem. A billentyűk lenyomásakor bekövetkező eseményeket a **7–25. példa** segítségével tanulmányozhatjuk.

A lenyomott billentyű Unicode-értékét az *event*-objektum *keyCode* tulajdonsága adja meg. Ha a kód helyett a karakterre vagyunk kíváncsiak, akkor a *Chr* függvényt használjuk. A *keyCode* nem veszi figyelembe a Területi beállításokat. Enélkül az ő, Ő, ű és Ű magánhangzók Unicode-értéke nagyobb, mint 255. Ezért a *Chr* módosított változatát, a *ChrW* függvényt kell használnunk, amely már a 255-nél nagyobb értékeket is kezeli (**7–26. példa**):

```
Kod = window.event.keyCode
Karakter = ChrW(window.event.keyCode)
```

A karakterek vizsgálatát mindig az *onkeypress* eseménykezelőben végezzük, mert a másik kettőnél az *event* a nagybetű kódját adja meg akkor is, ha kisbetűt írtunk a szövegmezőbe!

A váltóbillentyűk állapotát az *event*-objektum *altKey*, *ctrlKey* és *shiftKey* tulajdonságai adják meg. Ha valamelyik billentyűt lenyomtuk, akkor a megfelelő tulajdonság értéke *True*, egyébként pedig *False*. Egyszerre természetesen több billentyűt is lenyomhatunk közülük. Külön a bal oldali váltóbillentyűk állapotát az *altLeft*, *ctrlLeft* és *shiftLeft* tulajdonságokkal vizsgálhatjuk. Ha egy jobb oldali „xx” váltóbillentyűt nyomtunk le, akkor az *xxKey* igaz, de az *xxLeft* hamis.

Mint említettük, ezen váltóbillentyűk állapotát az *onkeypress* nem adja meg, az *onkeydown* vagy az *onkeyup* azonban használható erre a célra. A tulajdonságokat a **7–27. példa** mutatja be. Hasonlítsuk össze a billentyűpárok hatását. Figyeljünk arra, hogy a bal oldali Alt elviszi a fókuszot a menüsorra!

## A billentyűzet felügyelete

A billentyűzet-események és a *keyCode* segítségével ellenőrizhetjük az adatok bevitelét. A lenyomott karakter ugyanis csak az *onkeypress* eseménykezelő befejezése után kerül be a szövegmezőbe. Ha az eseménykezelőben megváltoztatjuk a kód értékét, akkor másik karakter jelenik meg. Az alábbi utasítással az ábécének a lenyomott billentyű után következő betűjét írjuk ki:

```
window.event.keyCode = window.event.keyCode + 1
```

Felhasználtuk, hogy az angol ábécé betűi egymás után helyezkednek el az ANSI-kódok táblázatában.

A **7–28. példában** csak a kis- és a nagybetűk lenyomása esetén változtatjuk meg a kódot. A szkript a „z” helyett „a”-t, a „Z” helyett pedig „A”-t ír a szövegmezőbe. Egyszerű programunk az ékezetes magyar magánhangzókat változatlanul hagyja. Az Olvasó a forráskódba írt megjegyzések segítségével végigkövetheti az algoritmust.

Ha nem akarjuk, hogy a karakter megjelenjen a szövegmezőben, akkor adjunk a *keyCode*-nak 0 értéket:

```
window.event.keyCode = 0
```

Ha az előző példa kódjában szereplő *Select Case* utasítást kiegészítjük egy *Else* ággal, amely nullázza a *keyCode* értékét, akkor programunk csak az angol ábécé betűinek lenyomására fog reagálni. Ezt a változatot a CD-melléklet **7–29. példája** mutatja be. Az eljárás a *Select Case* utasítás bővítésével az ékezetes magyar magánhangzókra is alkalmazható.

Megjegyezzük, hogy a kód törlése helyett használhatnánk a

```
window.event.returnValue = False
```

értékadást, hiszen ez megakadályozza a böngésző alapértelmezés szerinti eseménykezelését, azaz a karakter megjelenítését a szövegmezőben.

A *keyCode* segítségével ellenőrizhetjük az adatbevitelt. A tömbök és táblázatok kezelésénél láttunk olyan példákat, melyeknél egy diák tantárgyi jegyeit olvastuk be. A tárolás előtt megvizsgáltuk, hogy a jegy 1 és 5 közé esik-e. A *keyCode* felhasználásával meg sem engedjük más karakter beírását.

A módszert a **7–30. példában** mutatjuk be, ahol kiszámítjuk a beírt jegyek átlagát is (most nem foglalkoztunk a bukással). Ennek algoritmusát már ismeri az Olvasó. A szövegmezőket nem láttuk el azonosítóval, a *document.all* kollekció *tags* metódusával hivatkozunk rájuk.

A bevitt karakter ellenőrzését a táblázat *onkeypress* eseménykezelőjében végezzük. Bár az eseményt egy szövegmező érzékeli először, az eseménybuborék terjedése során megkapja a táblázat is. Így nem kell minden szövegmezőhöz külön eseménykezelőt készíteni. Ha a beírt karakter nem felel meg a feltételeknek, akkor töröljük a *keyCode* értékét:

```
Sub Tablázat_onkeypress
    Dim Jegy
    Jegy = ChrW(window.event.keyCode)
    If Jegy < "1" Or Jegy > "5" Then
        window.event.keyCode = 0
    End If
End Sub
```

Megjegyezzük, hogy az *event*-objektum tulajdonságainak többsége nem módosítható a *keyCode* vagy a *cancelBubble* tulajdonsághoz hasonló módon.

### Az Enter vizsgálata

A szövegmezők kitöltésénél a legtöbb Windows programban lehetőségünk van arra, hogy a parancsgombon való kattintás helyett az Enter billentyű lenyomásával fejezzük be az adatbevitelt. Ehhez általában az űrlapok egy speciális parancsgombját használják fel, amivel a későbbiekben ismerkedünk meg.

A **7–31. példában** a *keyCode* segítségével létrehozzuk ugyanezt a funkciót. Ha a felhasználó lenyomja az Enter billentyűt, akkor meghívjuk a bevitelhez tartozó parancsgomb *onclick* eseménykezelőjét. A vizsgálathoz tudnunk kell, hogy az Enter billentyű ANSI-kódja: 13.

```
If window.event.keyCode = 13 Then
    Beolvas_onclick
End If
```

Figyeljünk arra, hogy a *Beolvas\_onclick* eljárás végeztével a vezérlés visszakerül a hívó eljáráshoz! Ha az *If* utasítás után még vannak utasítások, akkor azok is végrehajtódnak.

Bár egy eljárást egy másik eljárásból minden további nélkül meghívhatunk, elegánsabb megoldás, ha az objektum *click* metódusával szimuláljuk az *onclick* eseményt:

```
Beolvas.click()
```

Ekkor az operációs rendszer úgy reagál, mintha valóban végbement volna az esemény. Létrehozza a megfelelő üzenetet, és ha nem tiltjuk le, megindul az eseménybuborék terjedése.

### A fókusz felügyelete

Bár nem kifejezetten a billentyűzettel kapcsolatosak, de az adatbevitel során gyakran használjuk az *onfocus* és az *onblur* eseményeket. Az *onfocus* akkor következik be, amikor egy objektum megkapja, az *onblur* pedig amikor elveszti a fókuszot. Segítségükkel például tájékoztathatjuk a felhasználót a várt adatokra vonatkozó tudnivalókról. Sem az *onfocus*, sem az *onblur* nem vesz részt az eseménybuborék terjedésében, tehát csak annak az objektumnak az eseménykezelője aktivizálódik, amelyre az esemény vonatkozik.

A **7–32. példában** a már ismert módszerrel meghatározzuk, hogy a szövegmezőbe írt dátum a hét melyik napjára esik. Az adatok ellenőrzését azonban most nem a parancsgomb eseménykezelőjében, hanem már a bevitelnél elvégezzük.

A hibaüzeneteket a globális szkript elején konstansokban deklaráljuk, így rövidítjük az ellenőrzést végző kódot. Hiba esetén az üzenetet egy táblázat segítségével a szövegmező melletti, egyébként üres cellába írjuk. Ezzel a felhasználót megkíméljük a hibaüzenet párbeszédablakának OK gombjára való gyakori kattintgatástól.

A szövegmezőkhöz *onblur* eseménykezelőket rendelünk, melyekben megvizsgáljuk a begépelte értéket. A hónap ellenőrzésénél például először megnézzük, hogy egyáltalán számot írtak-e be:

```
If Not IsNumeric(Honap) Then  
    HonapHiba.innerHTML = NemSzam
```

Utána az intervallumot ellenőrizzük:

```
ElseIf Honap < 1 Or Honap > 12 Then  
    HonapHiba.innerHTML = RosszHonap
```

Ha az adat megfelelt a feltételeknek, akkor töröljük az esetleges előző hibaüzenetet, és befejezzük a eseménykezelést:

```
Else  
    HonapHiba.innerHTML = "&nbsp;"  
End If
```

Ez az eljárás akkor hajtodik végre, ha a felhasználó beírt egy karaktersorozatot a hónaphoz, és átlép egy másik szövegmezőbe (vagy rákattint a parancsgombra). Hibás adat esetén a szövegmező mellett megjelenik a figyelmeztető üzenet.

Ezzel a módszerrel a számítások megkezdése előtt, a parancsgomb *onclick* eseménykezelőjében, nem tudjuk, hogy megfelelő értékeket találunk-e a szövegmezőkben. Ezért a *JoEv*, *JoHonap*, *JoNap* globális logikai változókkal üzenünk a parancsgomb eseménykezelőjének. A szövegmezők *onblur* eseménykezelőinek elején hamisra állítjuk a megfelelő értéket, például:

```
JoHonap = False
```

és csak a hibavizsgálat utolsó, *Else* ágában állítjuk igazra, mert ekkor megfelelő az adat:

```
Else  
    HonapHiba.innerHTML = "&nbsp;"  
    JoHonap = True  
End If
```

Mivel a betöltés után a szövegmezők még üresek, a globális szkriptben is hamisra állítjuk a logikai változók értékét:

```
JoEv = False : JoHonap = False : JoNap = False
```

A parancsgomb eseménykezelőjében csak akkor végezzük el a számítást, ha mindhárom változó igaz értékű:

```
If JoEv And JoHonap And JoNap Then  
    ' a számítások elvégzése
```

Ezzel be is fejezhetnénk az eljárást, hibás adat esetén nem tenne semmit a szkript. „Barátságos” programunk azonban ilyenkor megjelenít egy hibaüzenetet, és visszaállítja a kurzort az első hibás adat szövegmezőjére:

```
Else
    window.alert("Töltse ki hibátlanul a szövegmezőket!")
    If Not JoEv Then
        EvBe.select()
    ElseIf Not JoHonap Then
        HonapBe.select()
    Else ' ekkor már csak a nap lehet rossz
        NapBe.select()
    End If
End If
```

Bár a nagyon hasonló elemek miatt a HTML-kódot és a szkriptet is rövidíthetnénk a kollekciók és az *event*-objektum alkalmazásával, a program megértését azonban ezzel megnehezítettük volna. Ezért maradtunk a hosszadalmasabb, de egyszerűbb megoldásnál.

A dokumentum objektummodell még tartalmaz néhány eseményt, főleg az adatbázis-kezeléssel kapcsolatban. Ezekkel azonban nem foglalkozunk.

## 7.4. Hibakezelés a szkriptekben

Már sokszor felhívtuk a figyelmet arra, hogy az adatbevitel során, a műveletek végzésénél, a függvények hívásánál mindig ellenőrizni kell az adatok érvényességét. A programokat fel kell készíteni a nem megfelelő adatok fogadására, különben hibajelzéssel megszakad a végrehajtás. Az eddigiekben feltételes elágazásokat használtunk az adatok vizsgálatánál, beleértve az *onkeypress* esemény által nyújtott lehetőségeket. Az eseménykezelés és az *Err* (error, hiba) objektum hatékonyabb módszert kínál a hibák kezelésére.

### A programhibák típusai

A program írása és futtatása során számos hibalehetőség leselkedik a gyanútlan programozóra. Ezeket három nagy csoportba sorolhatjuk.

*Szintaktikus hibák:* a nyelv formális előírásainak megszegéséből adódnak. Rosszul gépelünk be egy kulcsszót, lemarad egy *End If* stb. Szerencsére a böngésző betöltéskor ellenőrzi a szintaxist, és azonnal figyelmeztet ezekre a hibákra. Mire a weblap a felhasználó elé kerül, nem maradhat benne szintaktikus hiba.

*Logikai hibák:* a hibás algoritmus következtében a program nem úgy működik, ahogy elvárjuk tőle. Ezeknek a hibáknak a felderítése a legnehezebb. Alaposan át kell gondolni a használt algoritmust, és elemezni a forráskódot. A javítást hibakereső programok, úgynevezett debuggerek segítik. A Microsoft Script Debugger letölthető a Microsoft weblapjáról. Használatát a függelékben ismertetjük.

*Futási hibák:* akkor jönnek létre, amikor egy utasítás szabálytalan műveletet próbál végrehajtani. Nem létező objektumra hivatkozik, vagy negatív számból von gyököt. Bekövetkezésüknél az interpreter hibaüzenetet ad, és leállítja a szkript utasításainak a feldolgozását. Programunkat fel kell készíteni a futási hibákat okozó esetekre, hogy ne szakadjon meg a végrehajtás.

A futási hibák egyik gyakori forrása a hibás adatbevitel. Ilyenkor figyelmeztessük a felhasználót a tévedésre, és tegyük lehetővé, hogy kijavítsa a hibát! Írjunk „bolondbiztos” programot.

A felhasználók nagyon „találékonyak”, a program írásánál nem jut eszünkbe minden hibalehetőség. A VBScript két eszközt kínál a futási hibák kezelésére. Az *On Error Resume Next* utasítás segítségével úgynevezett kivételkezelést végezhetünk. Az interpreter a hibától függetlenül folytatja az utasítások végrehajtását, és ránk bízva, hogy mit teszünk. A másik lehetőség a dokumentum objektummodell *onerror* eseményének a felhasználása. Először ezzel foglalkozunk.

## Az Err-objektum

A futás közben bekövetkező hibák jellemzőit az *Err*-objektum tartalmazza, melynek legfontosabb tulajdonságai:

- Description:* a hiba leírását tartalmazó sztring. Szükség esetén tájékoztathatjuk vele a felhasználót.  
*Number:* a hiba sorszáma. A hiba azonosítására használjuk.  
*Source:* a hibát létrehozó objektum megnevezése. Szkriptjeink esetén ez a VBScript interpretere.

Az *Err*-objektum a dokumentum betöltésekor jön létre. Ha nincsen hiba, akkor tulajdonságainak értéke 0, illetve üres sztring. A leggyakoribb futási hibák sorszámát a CD-melléklet Dokumentumok mappájában találjuk.

A tulajdonságoknak mi is adhatunk értéket. Ezzel mintegy üzenetet közvetítünk a szkript és a hibakezelő eljárás között. Sorszámnak olyan értéket válasszunk, ami még nem foglalt. Ehhez a *vbObjectError* konstanst nyújt segítséget, amelyhez bármilyen pozitív egész számot hozzáadhatunk:

```
Err.Number = vbObjectError + 1      ' saját hibakód  
Err.Description = "Hibás adat."  
Err.Source = "Szövegmező"
```

Az *Err*-objektum *Clear()* metódusa törli az objektum tulajdonságainak értékét. Az *Exit Sub*, *Exit Function* és a később ismertetésre kerülő *On Error Resume Next* utasítások automatikusan meghívják a *Clear* metódust.

A *Raise(HibaSorszám)* metódus szimulálja a sorszámnak megfelelő hibát. Főleg arra használjuk, hogy kipróbáljuk a programunk reagálását a különböző hibákra, de a segítségével meghívhatjuk hibakezelő eljárásunkat a VBScript által nem kezelt hibák esetén is.

A *Raise* metódus paraméterei között megadhatjuk a hiba forrását és leírását:

```
Err.Raise(HibaKód, Forrás, Leírás)
```



Hívása előtt célszerű végrehajtani a *Clear*-t, mert az általunk be nem állított (és itt nem említett) tulajdonságok előző értéke megmarad. Az *Err*-objektum használatára az *onerror* esemény ismertetése után mutatunk példát.

**FIGYELEM!**

Az *onerror* esemény kezeléséhez le kell tiltani az Internet Explorer hibakereső szolgáltatását. Ehhez az Eszközök/Internetbeállítások menüpontnál kattintsunk a Speciális fülre. A Böngészés csoportnál tegyünk pipát a „Hibakeresés tiltása a parancsfájlokban” jelölőnégyzetbe.

A Microsoft Script Debuggerhez vagy a Microsoft Script Editorhoz hasonló programok telepítése után újra ellenőrizni kell a beállítást, mert megváltozhat.

**Az onerror esemény**

Az *onerror* esemény akkor következik be, ha valamilyen hiba lép fel a szkript utasításainak végrehajtásakor. Az eseménykezelő három paraméterrel rendelkezik, és a *window*-objektumhoz rendeljük:

```
Sub window_onerror(Üzenet, URL, Sor)
```

Az *Üzenet* karakterláncként megadja azt a hibaüzenetet, amit a böngésző is megjelenítene a bekövetkezésekor. A *URL* tartalmazza a fájl elérési útját, akár a helyi gépen, akár az Interneten, a *Sor* pedig annak a sornak a száma a dokumentumban, ahol a hiba bekövetkezett.

Ezek a paraméterek azonban nem sokat segítenek a hiba azonosításában. A *URL*-t úgyis ismerjük, az *Üzenet* elég hosszú, és függ a böngésző verziójától, a *Sor* száma pedig változhat, ha módosítjuk a programot. Ezért az azonosításhoz az *Err*-objektum *Number* tulajdonságát használjuk.

Az *onerror* arra ad lehetőséget, hogy a hibakezelést elkülönítsük a program fő tevékenységétől. A kódot nem kell szétszabdalni a beírt adatok ellenőrzését végző utasításokkal. A vizsgálatokat egy külön eljárásba, az *onerror* eseménykezelőjébe tesszük.

A hiba bekövetkezése, és az *onerror* eseménykezelő végrehajtása után megszakad a szkript utasításainak a feldolgozása. A későbbiekben egy rugalmasabb hibakezelési lehetőséggel is megismerkedünk.

Az *onerror* eseményt csak az alprogramok utasításai hozhatják létre. Nem következik be a globális szkript utasításainak végrehajtásánál és szintaktikus hibáknál sem. Szükség esetén a globális szkript helyett használjuk az *onload* eseménykezelőt.

Megjegyezzük, hogy néhány objektumhoz (például képhez vagy hivatkozáshoz) is hozzárendelhetünk *onerror* eseménykezelő eljárást. Ez akkor lép működésbe, ha a fájl betöltése során hiba lép fel.

## Az onerror használata

Készítsünk programot, amely megjelenít két szövegmezőt, és elosztja egymással a beírt számokat! Ezzel az egyszerű feladattal már találkoztunk. A szövegmezők tartalmát *If* utasítással ellenőriztük az osztás elvégzése előtt. Most az *Err*-objektum és az *onerror* eseménykezelő segítségével fogjuk elvégezni a hibakezelést. Így az osztást végző szubrutin egyszerűen:

```
Sub Oszto_onclick
    Hanyados.innerText = Szamlalo.value / Nevezo.value
End Sub
```

A művelettel kapcsolatban háromféle hiba fordulhat elő:

Sorszám:	Üzenet:	Hiba:
13	Típuseltérés	üres a szövegmező, vagy nem számot írtunk be
6	Túlcsordulás	mindkét érték nulla
11	Nullával való osztás	a nevező nulla, de a számláló nem az

A **7–33. példában** nem végeztünk ellenőrzést és hibakezelést. Hozzuk létre a fenti hibákat, és figyeljük meg a böngésző hibaüzeneteit! Ha kikapcsoltuk az üzenetek megjelenítését, akkor az állapotsorban látható sárga figyelmeztető háromszögre kell duplán kattintani.

A **7–34. példában** elkészítjük az *onerror* esemény kezelőjét. A hibák sorszáma helyett konstansokat használunk:

```
Const Tipuselteres = 13
Const Tulcsordulas = 6
Const NullavalOszto = 11
```

Az eseménykezelőben az egyes hibáknak megfelelő hibaüzenetet jelenítünk meg. Az esetek szétválasztását a *Select Case* utasítással végezzük:

```
Sub window_onerror(Uzenet, URL, Sor)
    Select Case Err.Number
        Case Tipuelteres
            window.alert("Nem számot írt be.")
        Case Tulcsordulas
            window.alert("Határozatlan érték.")
        Case NullavalOszto
            window.alert("Nullával nem lehet osztani.")
    End Select
```

Ha az *event*-objektum *returnValue* tulajdonságát *True*-ra állítjuk, akkor a böngésző nem jeleníti meg a saját hibaüzenetét. Ezzel befejeztük az eseménykezelő eljárást:

```
    window.event.returnValue = True
End Sub
```

Ha az osztásnál bekövetkezik a hiba, akkor az interpreter meghívja az eseménykezelő eljárást, amely kiírja a megfelelő hibaüzenetet. Utána megszakad a szkript végrehajtása, a felhasználónak módjában áll kijavítani a rossz értéket.

Megjegyezzük, hogy az eseménykezelő definíciójában akkor is fel kell sorolni a paramétereket, ha nem használjuk az értéküket.

Eseménykezelőnknek minden eshetőségre fel kell készülnie, mert minden hiba esetén meghívódik. Ezért egészítsük ki egy *Case Else* ággal, amelyben a többi hiba esetén megjelenítjük a *Uzenet* és a *Sor* paraméterek értékét:

```
Case Else
    window.alert("Hiba a " & Sor & ". sorban:" & _
                vbNewLine & Uzenet)
End Select
```

Mivel jelen példánkban sohasem kerülne erre az ágra a vezérlés, ezért a **7–35. példában** töröltük a típuseltérés felismerését. Így ha nem számot írunk be a szövegmezőbe, akkor kipróbálhatjuk a módosítást.

### Futási hiba generálása

A *Raise* metódus ismertetésénél említettük, hogy segítségével olyan esetben is meghívhatjuk az *onerror* esemény kezelőjét, amit a VBScript nem tekint hibának. Ha a 7–35. példában nem akarjuk megengedni a negatív számok begépelését, akkor deklaráljunk egy ezt jelző saját hibakódot. Rendeljünk hozzá például 10-et:

```
Dim NegativSzam
NegativSzam = vbObjectError + 10
```

A **7–36. példában** az *onerror* eljárásba bele vesszük az új hiba vizsgálatát:

```
Case NegativSzam
    window.alert("Ne írjon be negatív számot!")
End Select
```

Az osztást végző eljárást kiegészítjük a számláló és a nevező vizsgálatával. Mindkét esetben az *If ... Then* egyszerűsített szintaxisát használjuk. Ha az egész utasítást egy sorba írjuk, akkor nincs szükség az *End If*-re:

```
Err.Clear()
If Szamlalo.value < 0 Then Err.Raise(NegativSzam)
If Nevezo.value < 0 Then Err.Raise(NegativSzam)
```

A teljes kódot a CD-mellékleten lévő fájl tartalmazza. Ha negatív számot írunk be valamelyik szövegmezőbe, akkor meghívódik az *onerror* eseménykezelő. Ezzel elértük, hogy különválasztottuk a hibakezelést és a számítások végzését. Formálisan elkerültük az eljárásból való kiugrást is, ami megszegné a strukturált programozás szigorú értelemben vett szabályait.

## Kivételkezelés

A futási hibák miatt bekövetkező leállást az *onerror* esemény felhasználásán túlmenően kivételkezeléssel kerülhetjük el. A kivételek olyan esetek, melyek nem tartoznak bele a program által megvalósított algoritmus folyamatába. Kezelésük olyan tevékenység, amit csak bekövetkezésük esetén hajtunk végre. A VBScript kivételkezelése erősen korlátozott, a programozási nyelvek gyakran sokkal fejlettebb kivételkezelési lehetőségekkel rendelkeznek.

A kivételkezeléshez az *On Error Resume Next* utasítás használjuk. Az utasítás kapcsolja az interpreter hibakezelését. Az interpreter hibaüzenet nélkül abbahagyja a hibát okozó utasítás végrehajtását, a hibának megfelelően beállítja az *Err*-objektum tulajdonságait, és továbblép a következő utasítás végrehajtására.

Az *On Error Resume Next* utasítást általában azon utasítás elé helyezzük, amely hibát okozhat. Az utasítás után pedig megvizsgáljuk az *Err*-objektum *Number* tulajdonságát. Ha ez nulla, akkor sikeresen végrehajtódott az utasítás. Egyébként hibaüzenetet adunk, vagy meghívunk egy hibakezelő eljárást. A végrehajtás után célszerű visszakapcsolni az interpreter saját hibakezelését. Ezt az *On Error GoTo 0* utasítással tehetjük meg.<sup>26</sup> A hibakezelés algoritmus:

```
On Error Resume Next
a hibához vezető utasítás
If Err.Number <> 0 Then hibaüzenet vagy eljáráshívás
On Error GoTo 0
```

Az *On Error Resume Next* hatása akkor is megszűnik, ha kilépünk egy alprogramból, vagy meghívunk egy újabb alprogramot.

Hiba bekövetkezése esetén gyakran abbahagyjuk az alprogram végrehajtását, hiszen nem tudjuk felhasználni az utasítás eredményét. Az *Err.Number* értékét célszerű törölni a hibakezelő eljárásban, különben a következő *On Error Resume Next* utasításig megmarad.

A 7–37. példában a 7–34. példa hibakezelését kivételkezeléssel oldjuk meg:

```
On Error Resume Next
Hanyados.innerText = Szamlalo.value / Nevezo.value
If Err.Number <> 0 Then Hibakezeles()
On Error GoTo 0
```

A *Hibakezeles* eljárás szinte teljesen megegyezik az *onerror* eseménykezelőjével. Mivel itt nem jön létre az *event*-objektum, nem adunk neki visszatérési értéket, az *Err.Number*-t viszont nullára állítjuk.

Hiba bekövetkezésekor nem ugrunk ki a parancsgomb eseménykezelő eljárásából, mert az a fókusz beállítása után az amúgy is véget ér.

---

<sup>26</sup> A *GoTo* nem ugrik sehova, az utasítás visszakapcsolja az interpreter hibakezelését.

## 8. DHTML HALADÓKNAK

A továbbiakban kiegészítjük a DHTML-objektumokra vonatkozó ismereteinket. Megtanuljuk, hogyan kell képeket beilleszteni, űrlapokat kezelni. Külső objektumok segítségével bővítjük weblapjaink hatékonyságát. A felhasználóval történő kommunikációt saját készítésű párbeszédablakkal tesszük változatosabbá.

A felsorolt témák részletes kifejtése külön kötetet igényelne. Itt a teljesség igénye nélkül, néhány példán keresztül mutatjuk be a programozó rendelkezésére álló eszközöket. Bővebb ismereteket a Microsoft Development Network weblapján találunk.

### 8.1. Képek a weblapon

A World Wide Web népszerűsége nagymértékben köszönhető a weblapokat díszítő ábráknak, fotóknak. Manapság a legtöbb weboldalon még a szöveget is grafikus formában helyezik el, így színesítik a látványt.

#### FIGYELEM!

Az Internet Explorer 6-os változata automatikusan átméretezi a képeket a weblapon. Kapcsoljuk ki ezt a funkciót az Eszközök/Internetbeállítások menüben a Speciális fülnél a Multimédia csoportban. Célszerű ugyanitt visszavonni a Kép eszköztár engedélyezését.

#### Képek beillesztése

A weblapon látható rajzokat, fényképeket az IMG (image, kép) objektum segítségével helyezzük el a dokumentumban. Az IMG más objektumokat nem tartalmaz, és nincs záró tagja. A képfájl nevét és elérési útját az *src* (source, forrás) tulajdonság határozza meg:

```
<IMG src = "elérési_út">
```

Az elérési út megadását a 2.2. fejezetben, a hivatkozások beillesztésénél ismertettük. Ne felejtsük el megadni a fájlnev után a kiterjesztést! Bár a böngésző más típusokat is felismer, a leggyakrabban .jpg vagy .gif kiterjesztésű állományokkal találkozunk. Ezek a rövidítések a kódolás módjára utalnak.

A **8–1. példában** egy légi felvétel látható a weblapon. A fotót a Képek almappába helyeztük, hogy ne keveredjen össze a példafájlokkal:

```
<IMG src = "Képek\Repülő.jpg">
```

A képfájl bárhol lehet, ahol elérhető a weblap megjelenítésekor. Akár az Interneten lévő képre is hivatkozhatunk, ekkor a megtekintéséhez élő Internet-csatlakozás szükséges. A **8–2. példa** a Nemzetközi Űrállomás pillanatnyi helyzetét bemutató ábrát illeszti a weblapra. Mivel az eredeti képet a NASA néhány percenként frissíti, a böngésző Frissítés gombjára kattintva követhetjük az űrállomás mozgását. Figyeljük meg az *src* értékét a HTML-kódban!

Az Internet esetén fennáll a veszély, hogy az eredeti helyéről törlik a képet, így a mi weblapunkon sem fog megjelenni. Az Internetre kerülő fájlok nevében ne használjunk szóközt és ékezetes magánhangzókat!

### A képobjektum tulajdonságai

A képobjektum számos tulajdonsággal rendelkezhet. Az *src* mellett lehet azonosítója (*id*). A *border* tulajdonság segítségével be is keretezhetjük. A tulajdonság értéke a keret pixelben mért vastagsága:

```
<IMG id = "Repulo" src = "Képek\Repülő.jpg" border = 5>
```

A keret színét a stílus tulajdonság *border-color* eleme határozza meg:

```
<IMG id = "Repulo" src = "Képek\Repülő.jpg" border = 5  
style = "border-color: green">
```

Ha nem adjuk meg a keret vastagságát, akkor a színe sem látszik. A szín a szkriptekben a nyitó tagtól kissé eltérő szintaxissal szerepel. Figyeljünk arra, hogy itt már nincsen kötőjel:

```
Repulo.style.borderColor = "green"
```

A **8–3. példa** segítségével kipróbálhatjuk a különböző vastagságú és színű keretek megjelenítését.

A kép szélességét a *width*, magasságát pedig a *height* tulajdonsággal adjuk meg. Mindkettőt pixelben mérjük:

```
<IMG id = "Repulo" src = "Képek\Repülő.jpg"  
width = 400 height = 300>
```

Ha értékeik nem arányosak az eredeti mérettel, akkor a böngésző torzítva jeleníti meg a képet. Az eredeti méretet a legtöbb grafikus programmal meghatározhatjuk. Vegyük figyelembe, hogy a nagyításnál romlik a minőség. A felsorolt tulajdonságok használatát a **8–4. példa** szemlélteti.

Ha a **8–5. példában** a kép felett helyezkedik el az egérmutató, akkor a középső egérgomb forgatásával folyamatosan nagyíthatjuk, illetve kicsinyíthetjük a képet. Az Olvasó eddigi ismeretei alapján megpróbálkozhat a forráskód elkészítésével. A böngésző alapértelmezett eseménykezelését le kell tiltani (`returnValue = False`), különben nagyméretű képek esetén mozgatni fogja a görgetősávot.

Megjegyezzük, hogy ezt a hatást a *zoom* (nagyítás) stíluselem változtatásával szintén elérhetjük. A *zoom* eredeti értéke 1, az arányt és a százalékos értéket azonban módosíthatjuk. A *zoom* használatát a **8–6. példa** mutatja be.

Az Internetre kerülő fájlok esetén akkor is célszerű megadni a *width* és a *height* tulajdonságokat, ha a képet eredeti méretében jelenítjük meg. Ekkor ugyanis a böngésző kihagyja a megfelelő méretű helyet, és a letöltés után nem kell átrendeznie a szöveget.

Megjegyezzük, hogy a képobjektumoknak még számos tulajdonságát beállíthatjuk. Szűrők (filters) használatával pedig különleges hatásokat érhetünk el. Erről a lehetőségről a Microsoft Development Network weblapján található további információ.

## A képek src tulajdonsága

A fenti példákban az objektumok *src* tulajdonságát a relatív elérési úttal adtuk meg. A tulajdonság lekérdezésekor azonban mindig a teljes elérési utat kapjuk vissza a következő formában:

```
"file:///meghajtó:/mappák/fájlnév.kiterjesztés"
```

például:

```
"file:///D:/Fejezet08/Képek/Tavaszi_hérics.jpg"
```

Bár az elérési útban mi a Windowsnak megfelelő \ jelet használjuk, az *src* sztringjében az Interneten elterjedt / jel szerepel. A böngészőt ez nem zavarja, de a szkriptekben figyelembe kell vennünk.

A képobjektumokhoz rendelt fájlnevet ugyanúgy meg lehet változtatni, mint a többi tulajdonság értékét. A **8–7. példában** három virág fényképét helyeztük el a weblapra. Az állományokat a virágokról neveztük el. Amelyik kép fölé visszük az egérmutatót, az elhalványodik, az ablak állapotsorában pedig megjelenik a virág neve.

A képek elhalványítását egyszerűen úgy oldjuk meg, hogy az eredeti kép helyett egy másikat illesztünk be a dokumentumba. A halványabb képnél a fájlnevet egy H betűvel egészítettük ki, a „Tavaszi hérics.jpg” párja például „Tavaszi héricsH.jpg”.

Az *src* tulajdonság változtatását a *document*-objektum *onmouseover* eseményéhez kötjük. Először a már ismert módon megvizsgáljuk, hogy az esemény képre vonatkozik-e:

```
Set Elem = window.event.srcElement
If Elem.tagName = "IMG" Then
```

Ha igen, akkor a későbbi visszaállításhoz egy globális változóban tároljuk az *src* tulajdonságot:

```
Eredeti = Elem.src
```

majd megkeressük benne a ".jpg" karaktersorozatot, és helyettesítjük "H.jpg"-vel:

```
Elem.src = Replace(Eredeti, ".jpg", "H.jpg")
```

Kereshettük volna csak a kiterjesztést elválasztó pontot is, de ha a mappák nevében szerepelne a kiterjesztés, akkor hibásan működne a programunk.

Ezzel a képet elhalványítottuk, a böngésző az *onmouseover* esemény bekövetkeztekor kicseréli a képeket. Az állapotsor megváltoztatásához ki kell emelnünk az *src*-ből az állomány nevét, a kiterjesztés nélkül. Ehhez a végéről kezdve megkeressük benne az első / és . (pont) karaktert. A fájlnev a / jel után kezdődik, hossza pedig éppen a két pozíció különbsége lesz. A kiemeléshez a *Mid* függvényt használjuk:

```
Kezd = InStrRev(Eredeti, "/") + 1
Veg = InStrRev(Eredeti, ".")
Nev = Mid(Eredeti, Kezd, Veg - Kezd)
```

```

Nev = Unescape(Nev)
window.status = Nev
End If

```

```
Set Elem = window.event.srcElement
If Elem.tagName = "IMG" Then
    Elem.src = Eredeti
    window.status = ""
End If
```

## Az images kollekció

A **8–9. példa** weblapjával kocka-pókert lehet játszani. Megjelenítünk 5 dobókockát, majd a parancsgombra való kattintás esetén mindegyiket véletlenszerűen változtatjuk.

```
<IMG src = "Kocka\K1.gif">&nbsp;&nbsp;&nbsp;
```

```
For I = 0 To 4
    Dobas = Int(6 * Rnd() + 1)
```

```
document.images(I).src = "Kocka\K" & Dobas & ".gif"
```

Next

A példában megtehetjük volna, hogy állománynévnek egyszerűen egy sorszámot választunk K betű nélkül, de be akartuk mutatni, hogyan lehet fájlneveket összeállítani sztringek összefűzésével.



## Az objektumok pozícionálása

A böngésző úgy kezel egy képet, mintha egyetlen karakter lenne, így akár a szövegrészek közé is kerülhet (inline beillesztés). A **8–10. példában** a szavak közé helyeztük az IMG-objektumot. Mint látjuk, a kép méretének megfelelően megnőtt a sortávolság.

A **8–11. példában** a dokumentum hátterét a kép hátterével egyező színűre állítottuk, így beleolvad a pereme az égboltba. A .gif kiterjesztésű képek hátterét képszerkesztő programokkal átlátszóvá lehet tenni. A .png típusú kódolás is ismeri az átlátszó részleteket.

Bár a tulajdonságok segítségével korlátozott mértékben megszabhatjuk a szöveg és a képek egymáshoz viszonyított elrendezését, erre a célra inkább táblázatokat használunk. A táblázatnak nem rajzolunk keretet, hogy ne zavarja a weblap megjelenítését. A **8–12. példa** bemutatja ezt a tördelési módot.

Izgalmasabb lehetőség áll a rendelkezésünkre a stílusok használatával. A stíuselemek segítségével pontosan megadhatjuk az objektumok helyét a böngésző ablakában. A pozícionálást a *left* (balra) és *top* (felül) stíuselemekkel végezzük el, melyek pixelben mérve meghatározzák a kép bal felső sarkának a dokumentum-ablak bal oldali és felső szélétől mért távolságát. Mivel a pozícionálás alapértelmezés szerint az objektum eredeti helyéhez viszonyítva történik (amivel nem foglalkozunk), nekünk a *position* tulajdonság értékét mindig *absolute*-ra kell állítanunk:

```
<IMG src = "Képek/Hérics.jpg"
    style = "position: absolute; left: 80; top: 50">
```

A CD-melléklet **8–13. példájában** beírhatjuk a kívánt pozíciót a szövegmezőkbe. Ha túl nagy az érték, akkor a böngésző görgetősávot hoz létre. A két koordináta negatív is lehet, ekkor a kép balra, illetve felfelé „kilóg” az ablakból.

A weblapon megjelenítettük a *style.left* és *style.top* tulajdonságok értékét, melyet karakterláncként kapunk vissza. A szám után szerepel a mértékegység (px), amit az értékadásnál nem kell megadni. Ha sztringek helyett numerikus értékeket szeretnénk kapni, akkor a *pixelLeft* és *pixelTop* tulajdonságokat használjuk:

```
X = Virag.style.pixelLeft : Y = Virag.style.pixelTop
```

Ezeket már numerikus kifejezésekbe is beírhatjuk. A kétfajta tulajdonság közötti különbség csak a lekérdezésnél mutatkozik. Ha értéket adunk a tulajdonságoknak (pixelben), akkor mindegy, hogy a *left*-et vagy a *pixelLeft*-et, illetve a *top*-ot vagy a *pixelTop*-ot alkalmazzuk.

A pozícionálást nem csak képekre, hanem többek között a DIV-, SPAN-, INPUT-objektumokra, táblázatokra is elvégezhetjük. A **8–14. példában** a 8–12. példához hasonló elrendezést pozícionálással hoztuk létre.

Ennek a megoldásnak az a hátránya, hogy ha a felhasználó megváltoztatja a szöveg méretét, akkor módosul a kép és a szöveg egymáshoz viszonyított elrendezése is.

## A With utasítás

A 8–13. példa forráskódjában sokszor előfordult a *Virag.style* hivatkozás. Az objektumok tulajdonságainak vagy metódusainak minősítését a *With ... End With* utasítással egyszerűsíthetjük. A *With* után meg kell adni az objektum nevét (vagy egy objektum-hivatkozást), melyet aztán a *With* és az *End With* között nem kell kiírni:

```
With Objektumnév
    .tulajdonságnév = ...
    egyéb utasítások
End With
```

A *With* utasítással az *Objektumnév.style* hivatkozást is rövidíthetjük. A *Virag*-objektum stíluselemeire például a következőképpen hivatkozhatunk:

```
With Virag.style
    .left = 400
    .top = 200
    további utasítások
End With
```

Az interpreter minden ponttal kezdődő tulajdonságot kiegészít a *With* után szereplő *Virag.style* minősítéssel.

A *With* és az *End With* között természetesen bármilyen utasítás állhat. Más objektumokra a szokásos módon hivatkozunk. A *With* utasításokat egymásba lehet ágyazni, de csak a legbelső *With* által meghatározott minősítést hagyhatjuk el.

A 8–15. példa a *Virag*-objektum esetén mutatja be a *With* használatát.

## Az átfedések beállítása

A pozícionálás következtében az egyes elemek egymásra kerülhetnek. Ilyenkor a stílus *z-index* eleme határozza meg az átfedések sorrendjét. Amelyik objektumnak nagyobb a *z-indexe*, az feljebb helyezkedik el, és eltakarja a kisebb *z-indexű* elemeket. A *z-index* megadása a többi stíluselemhez hasonlóan történik. A HTML-kódban:

```
<IMG id = "Piros" src = "Piros.gif" width = 100 height = 100
    style = "position: absolute; left = 100; top = 100;
    z-index = 10">
```

a szkriptekben pedig:

```
Piros.style.zIndex = 10
```

A kódban a *z-indexek* értékét magunk határozhatjuk meg. Ha két objektumnak egyforma *z-indexet* adunk, akkor a HTML-kód sorrendjében kerülnek egymásra. Ha egy objektumnak negatív a *z-indexe*, akkor az a többi elem alá kerül.

A 8–16. példában elhelyeztünk három különböző színű, egymást átfedő négyzetet. Amelyikre rákattintunk, az kerül felülre. A legnagyobb *z-indexet* egy *Maximum* nevű globális változóban tartjuk nyilván. Az *onclick* eseménykezelőben 10-zel megnöveljük az értékét, majd átírjuk a kiválasztott négyzet *z-indexébe*:

```
Maximum = Maximum + 10
window.event.srcElement.style.zIndex = Maximum
```

Ezzel a megoldással a *Maximum* értéke ugyan folyamatosan nő, de elég sokáig kéne kattintgatni az egérrel, mire elérnénk a számtartomány felső határát.

A CD-n lévő példában az eseménykezelőt a *document* objektumhoz rendeltük, és megvizsgáltuk, hogy IMG-objektumra kattintottunk-e. A fájlméret csökkentésének érdekében egészen kicsi (1x1 pixel) a négyzetek mérete, a *width* és *height* tulajdonság segítségével nagyítottuk fel őket.

### Az objektumok mozgatása

A pozíció változtatásával lehetőségünk van a dokumentum objektumainak áthelyezésére. A **8–17. példa** weblapján egy négyzetet lehet mozgatni az egér segítségével.

Az elmozdításhoz először meg kell fogni a négyzetet, azaz le kell nyomni rajta a bal egérgombot. Ezt az állapotot egy logikai változóban tároljuk, aminek kezdetben *False* értéket adunk. A globális szkriptben feljegyezzük a négyzet eredeti pozícióját is:

```
Mozgat = False
NegyzetX = 350
NegyzetY = 100
```

Ha lenyomjuk a bal egérgombot a négyzet fölött, akkor engedélyezzük a mozgást, és globális változóknak tároljuk az egérmutató pozícióját:

```
Sub Negyzet_onmousedown
    Mozgat = True
    EgerX = window.event.x
    EgerY = window.event.y
End Sub
```

Ha bekövetkezik az *onmousemove* esemény, és engedélyeztük a mozgást (lenyomott állapotban van az egérgomb), akkor kiszámítjuk, hogy mennyivel mozdult el az egér a gomb lenyomásának helyéhez viszonyítva:

```
Sub Negyzet_onmousemove
    Dim XMozdul, YMozdul
    If Mozgat Then
        XMozdul = window.event.x - EgerX
        YMozdul = window.event.y - EgerY
    End If
End Sub
```

és ugyanennyivel mozdítjuk el a négyzetet az előző helyzetéhez képest:

```
Negyzet.style.left = NegyzetX + XMozdul
Negyzet.style.top = NegyzetY + YMozdul
```

Végül jelezzük a böngészőnek, hogy kezeltük az eseményt (egyébként megjelenítene például egy stoptáblát a négyzet fölött):

```
window.event.returnValue = False
End If
End Sub
```

Ha a felhasználó felengedte az egérgombot, akkor letiltjuk a további mozgást (különben minden *onmousemove*-ra mozogna a négyzet):

```
Sub Negyzet_onmouseup  
    Mozgat = False
```

és feljegyezzük a négyzet pozícióját, amire a következő mozgatásnál lesz szükség. Mint már említettük, a *style.left* és *style.top* tulajdonságok értéke sztring, a szám után áll a mértékegység (px). Ezért most a numerikus értéket visszaadó *pixelLeft* és *pixelTop* tulajdonságokat használjuk:

```
    NegyzetX = Negyzet.style.pixelLeft  
    NegyzetY = Negyzet.style.pixelTop  
End Sub
```

Ha a négyzet z-indexének negatív értéket adunk, akkor mozgítás közben nem takarja el a weblap szövegét. Ezt a **8–18. példában** mutatjuk be. Mivel nem tiltottuk le, a négyzetet végleg kivihetjük az ablakból!

### A weblap háttere

A dokumentum megjelenését csinosíthatjuk, ha képet illesztünk be háttérnek. Az elérési utat a BODY *background* tulajdonságaként adjuk meg. Ha a kép kisebb, mint az ablak mérete, akkor a böngésző vízszintesen és függőlegesen is többször kirajzolja (**8–19. példa**). A **8–20. példában** ezt a hatásos megjelenítéshez használjuk fel.

Ha a *bgProperties* tulajdonságot *fixed*-re állítjuk, akkor a görgetősávok használatával sem mozdul el a háttérkép (**8–21. példa**). A beállítás törléséhez az üres sztringet rendeljük a tulajdonsághoz.

Az ismétlést a *background-repeat* (háttér-ismétlés) stíluselem segítségével lehet letiltani. A *background-position* stíluselem alkalmazásával a képet mind vízszintesen, mind függőlegesen középre igazíthatjuk (**8–22. példa**):

```
style = "background-repeat: no-repeat;  
        background-position: center center"
```

A stílusok segítségével további beállításokat végezhetünk. Ezeket a CD-melléklet Dokumentumok mappájában található Stíluselemek.htm fájlban ismertetjük. A háttérképet a szkriptekben a *backgroundPositionX* és *backgroundPositionY* stíluselemekkel pozícionálhatjuk (**8–23. példa**). A pozícionáláshoz a *background-repeat* stíluselemet *no-repeat*-re kell állítani.

A *background* a többi tulajdonsághoz hasonlóan elérhető és módosítható a szkriptekből. Ügyeljünk arra, hogy háttérnek csak eléggé halvány, kontrasztmentes képet alkalmazzunk, amely nem zavarja a szöveg olvasását.

Megemlítjük, hogy a *background-image* stíluselem segítségével táblázatokhoz, cel-lákhoz, szövegmezőkhöz, parancsgombokhoz, SPAN, DIV és más elemekhez is rendelhetünk háttérképet.

## 8.2. Időzítők használata

Eddigi példáinkban egy esemény bekövetkezése, vagy egy alprogram közvetlen meghívása okozta az utasítások végrehajtását. A VBScript módot nyújt ahhoz, hogy megadott idő múlva, vagy megadott időközönként hajtsunk végre egy alprogramot. Ezt az időzített futtatást a *window*-objektum *setTimeout* és *setInterval* metódusaival érhetjük el.

### Utasítások időzített végrehajtása

A *setTimeout* függvény egy utasítást hajt végre az ezredmásodpercben megadott idő elteltével:

```
Azonosító = window.setTimeout("Utasítás", Időtartam)
```

Az utasítást idézőjelek között, karakterláncként kell megadni. Mindkét paraméter helyén természetesen a megfelelő értéket előállító kifejezések is szerepelhetnek. A függvény visszatérési értéke egy Long altípusú azonosító, melyet az időzített végrehajtás leállításához használunk fel.

A **8–24. példában** a parancsgombra való kattintás után 5 másodperc múlva megváltoztatjuk a háttérszínt. Ehhez az eseménykezelőben meghívjuk a *setTimeout* függvényt, melynek paramétereként megadjuk a háttérszínt beállító utasítást, valamint az 5000 ezredmásodpercet:

```
Temp = window.setTimeout("document.bgColor = 'red'", 5000)
```

Bár az időtartam ezredmásodpercekben szerepel az utasításban, az időmérés körülbelül huszadmásodperc pontossággal történik.

Az indítás után letiltottuk a parancsgomb használatát, hogy ugyanolyan azonosítóval ne indulhasson újabb időzítés. A tiltást a weblap frissítésével tudjuk feloldani.

A *setTimeout* első paramétereként általában egy külön eljárást adunk meg:

```
Temp = window.setTimeout("Piros()", 5000)
```

Ebben már ismét engedélyezni tudjuk a parancsgomb használatát:

```
Sub Piros()  
    document.bgColor = "red"  
    Start.disabled = False  
End Sub
```

Az ismétléshez a **8–25. példában** a Start gomb *onclick* eseménykezelőjében visszaállítottuk az eredeti háttérszínt.

Az időzítő leállításához a *clearTimeout* metódust használjuk fel, amely a várakozási idő letelte előtt megakadályozza a végrehajtást. A függvény paramétere a *setTimeout* által visszaadott azonosító:

```
window.clearTimeout(Azonosító)
```

A **8–26. példában** a háttérszín változtatása előtt a Mégse gomb segítségével módunk van leállítani a végrehajtást. A *clearTimeout* meghívását a parancsgomb *onclick* eseménykezelőjébe tettük, ahol a Start gomb tiltását is feloldjuk:

```
Sub Megse_onclick
    window.clearTimeout(Temp)
    Start.disabled = False
End Sub
```

Lényeges, hogy a *Temp* globális változó legyen, különben a Start gomb eseménykezelőjének befejezése után eltűnik, így nem tudjuk leállítani az időzített végrehajtást.

### A `setTimeout` ismételt meghívása

Ha egymás után többször meg akarjuk hívni a *setTimeout* függvényt, akkor nem használhatunk ciklust. Így ugyanis minden ismétlés során újabb időzítőt indítanánk el, még mielőtt az előző végrehajtódna. Ciklus helyett a *setTimeout* által meghívott eljárásban indítjuk el a következő *setTimeout*-ot.<sup>27</sup> A számlálást egy globális változó segítségével végezzük.

A módszert a **8–27. példa** mutatja be, amely tízszer véletlenszerűen megváltoztatja a háttérszínt. A szín választásához a 3–48. példában megismert *VeletlenSzin* függvényt használjuk, melynek forráskódját csatoljuk a dokumentumhoz.

A háttérszínt a *Valt* eljárással változtatjuk, amit először a parancsgomb *onclick* eseménykezelőjében hívunk meg. Ekkor nullázzuk a számlálást végző globális változót, és letiltjuk a parancsgomb használatát:

```
Szamlalo = 0 : Start.disabled = True : Valt()
```

Ha a *Valt* eljárásban a számláló értéke kisebb mint 10, akkor megváltoztatjuk a háttérszínt, megnöveljük a számlálót, és meghívjuk a *setTimeout* metódust. Időtartamnak most 2 másodpercet választunk:

```
If Szamlalo < 10 Then
    document.bgColor = Veletlenszin()
    Szamlalo = Szamlalo + 1
    Temp = window.setTimeout("Valt()", 2000)
```

A *setTimeout* 2 másodperc elteltével újra végrehajtja a *Valt* eljárást, ami ismét meghívja a *setTimeout*-ot és így tovább.

Ha a számláló értéke elérte a 10-et, akkor üzenetet küldünk a felhasználónak, és engedélyezzük a Start gomb használatát:

```
Else
    window.alert("Vége.")
    Start.disabled = False
End If
```

Mivel nem hívjuk meg újra a *Valt* eljárást, leáll a háttérszín változtatása.

A **8–28. példában** a kockadobást szimuláljuk azzal, hogy néhányszor váltogatjuk a számokat, mintha gurulna a kocka. A hatásosabb megjelenítés érdekében egy *Do ... Loop* ciklussal elkerüljük egymás után kétszer ugyanannak a számnak a megjelenítését. Az összehasonlításhoz az *Elozo* változóban tároljuk az előző dobás értékét.

---

<sup>27</sup> Ez nem rekurzió!

Megjegyezzük, hogy különböző azonosítók használatával egyszerre több időzített végrehajtást is elindíthatunk.

### Utasítások folyamatos ismétlése

A *window*-objektum *setInterval* metódusa meghatározott időközönként addig ismétli a megadott utasítást, amíg le nem állítjuk:

```
Azonosító = window.setInterval("Utasítás", Időköz)
```

Az időközt ezredmásodpercben adjuk meg.

A *setTimeout*-tal ellentétben a *setInterval* meghívását nem kell ismételgetni, és a leállítása a *clearInterval* metódussal történik:

```
window.clearInterval(Azonosító)
```

A **8–29. példában** egy kiségeret mozgatunk vízszintesen a képernyőn. A kép x-koordinátáját egy globális változóban tároljuk, melynek a betöltésnél adunk értéket:

```
Dim X, Temp  
X = 10
```

Az indítást végző parancsgomb *onclick* eseménykezelőjében meghívjuk a *setInterval* függvényt, amely 5 századmásodpercenként lefuttatja a *Mozgat()* eljárást:

```
Temp = window.setInterval("Mozgat()", 50)
```

A *Mozgat()* eljárásban áthelyezzük a képet, de figyelünk arra, hogy ne fusson ki az ablakból:

```
If X < 600 Then  
    X = X + 2  
    Eger.style.left = X
```

Ha a vízszintes koordináta értéke elérte a 600-at, akkor leállítjuk az időzítőt:

```
Else  
    window.clearInterval(Temp)  
End If
```

A példában elhelyeztünk egy másik parancsgombot is, amellyel leállíthatjuk a mozgást. Ehhez az *onclick* eseménykezelőben meghívtuk a *clearInterval* metódust. Közben a megfelelő módon letiltottuk, illetve engedélyeztük a Start parancsgomb használatát, és állítottuk a fókuszt is.

A mozgás sokkal életszerűbb, ha közben kissé változik a kép. A **8–30. példában** ezt a hatást úgy érjük el, hogy két képet jelenítünk meg egymással váltogatva. Az *Eger1.gif* és *Eger2.gif* képeken másképpen helyezkedik el az egér farka.

A képek váltogatását a *Mozgat()* eljárásban végezzük. Megtehetnénk, hogy sztringek összefűzésével állítjuk elő a fájlneveket, helyette azonban egy áttekinthetőbb és gyorsabb módszert alkalmazunk. A *Kep* nevű globális változóban tartjuk nyilván, hogy éppen melyik egeret jelenítettük meg. A változó aktuális értékét *Select Case* utasítással választjuk ki, majd a fájlnévvel együtt megváltoztatjuk:

```
Select Case Kep
  Case 1
    Fajl = "Képek\Egér2.gif"
    Kep = 2
  Case 2
    Fajl = "Képek\Egér1.gif"
    Kep = 1
End Select
```

Ezt követően beállítjuk a kép *src* tulajdonságát, és elmozdítjuk az egeret a képernyőn:

```
Eger.src = Fajl
X = X + 4
Eger.style.left = X
```

Az előző példához képest megnöveltük az időtartam hosszát, hogy elkerüljük a kép vibrálását. Ezzel együtt megnöveltük a léptetés mértékét is.

A **8–31. példában** körbejár a kisegér. Ehhez a mozgásirányoknak megfelelően négy képet és négy eljárást alkalmazunk. A mozgatus az eddigieknek megfelelően történik. Ha az egér elérte a határvonalat, akkor leállítjuk az ismétlést, megváltoztatjuk a képet, és elindítjuk a következő iránynak megfelelő ismétléseket. A sarkokon a képméretnek megfelelően változtatni kell a koordinátákon, hogy ne ugorjon túl nagyot a kisegér.

### Több időzítő indítása

A *setTimeout*-hoz hasonlóan a *setInterval* metódussal több időzítőt is elindíthatunk.

A **8–32. példában** három számlálót helyeztünk el a weblapon, melyek bizonyos időközönként eggyel megnövelik a kiírt számot. A számlálókat parancsgombokkal lehet elindítani és leállítani, egymástól függetlenül. A forráskódban mindent háromszor megismételtünk, hogy áttekinthetőbb legyen a program.

A globális szkriptben minden egyes számláló értékét nullázzuk. Ne felejtjük el, hogy a számláláshoz globális változókat kell használni, különben az alprogramok végrehajtása után törlődne az értékük!

Az indítást végző parancsgombok *onclick* eseménykezelőjében letiltjuk a gomb további használatát, és elindítjuk az időzített ismétlést:

```
Sub Indulj1_onclick
  Indulj1.disabled = True
  Temp1 = window.setInterval("Folyamat1()", 1211)
End Sub
```

A periodikusan meghívódó, számlálást végző eljárásban eggyel megnöveljük, és kiírjuk a számláló értékét:

```
Sub Folyamat1()
  Szamlalo1 = Szamlalo1 + 1
  Szamlalo1ki.innerText = Szamlalo1
End Sub
```



A leállítást végző parancsgomb *onclick* eseménykezelőjében töröljük az ismétléseket, és engedélyezzük az indítógomb használatát:

```
Sub Allj1_onclick
    window.clearInterval(Temp1)
    Indulj1.disabled = False
End Sub
```

A **8–33. példában** bemutatjuk, hogyan lehet ezt a programot megírni fölösleges ismétlések nélkül. A számlálók, az azonosítók és az időtartamok értékét tömbökben tároljuk. Az eljárásoknak paraméterként átadjuk a parancsgombok indexét, ami az előző megoldásban az eljárások nevében szereplő szám volt. Az indexelés miatt ugyanazzal az azonosítóval láttuk el a parancsgombokat.

Az időzített ismétlések indításánál a *Folyamat* nevű eljárás hívását sztringek összefűzésével végezzük (az időtartamot a T tömbben tároltuk). Így adjuk át neki a parancsgomb indexét:

```
Temp(N) = window.setInterval("Folyamat(" & N & ")", T(N))
```

A leállításnál egyszerűbb a dolgunk, csak az azonosítókat tartalmazó tömb megfelelő elemét kell megadni:

```
window.clearInterval(Temp(N))
```

Vegyük észre, hogy a HTML-kódot is létrehozhatjuk ciklusokkal. Ennek módját a **8–34. példában** mutatjuk meg, melynek elemzését az Olvasóra bízunk!

### Az időzítők használata

A *setInterval* függvény használatára két látványosabb alkalmazást mutatunk be. A **8–35. példában** négy részeg tengerész bolyong egy lakatlan szigeten (felülről nézzük őket).

A bolyongást a koordináták véletlenszerű változtatásával érjük el. A tengerészek között vannak fürgébbek, ők gyakran, de kicsit lépnek, a lomhábbak ritkábban, de nagyobbat. A tengerészek annyira azért nem részegek, hogy belemenjenek a vízbe. Ezt az *Ellenoriz* függvény figyeli. A betöltés során az induló helyzetet is véletlenszerűen állítjuk elő. A szigetet egy zöld háttérrel rendelkező DIV-objektum rajzolja ki a képernyőre.

Az időzítőket a 4.3. fejezetben ismertetett *Execute* utasítással indítjuk el és állítjuk le. Az *Execute* a paramétereként megadott sztringet utasításként értelmezi, és végrehajtja. A *Temp0*, ..., *Temp3* azonosítóval ellátott időzítőket például a következő ciklus állítja le:

```
For I = 0 To 3
    Execute("window.clearInterval(Temp" & I & ")")
Next
```

A **8–36. példában** fogadást lehet kötni az egérversenyen. Az egerek véletlenszerűen változó sebességgel haladnak a cél felé. A program leghosszabb része az adatbevitel ellenőrzése, illetve az eredmény kiírása! Az egerek helyzetét és az időzítők azonosítóit tömbökben tároljuk. A mozgatót a tömbelemekre vonatkozó ciklussal indítjuk.

### 8.3. Űrlapok

Eddigi programjainkban szövegmezőket vagy a meglehetősen kényelmetlen *InputBox* függvényt használtuk adatbevitelre. A továbbiakban megismerkedünk még néhány lehetőséggel. Megmutatjuk, hogyan tudnak az egyes weblapok kommunikálni egymással. Így, ha a megjelenítés szükségessé teszi, akkor váltani is tudunk a dokumentumok között.

#### Vezérlőelemek

Az *INPUT* objektumosztály objektumait vezérlőelemeknek vagy űrlap-mezőknek nevezzük. Közülük már ismerjük a parancsgombot, a szövegmezőt és a jelszóbekérő szövegmezőt. A típusok teljes listája:

<i>button:</i>	parancsgomb	<i>password:</i>	jelszóbekérő szövegmező
<i>checkbox:</i>	jelölőnégyzet	<i>radio:</i>	választógomb
<i>file:</i>	állománylista	<i>reset:</i>	„alaphelyzet” gomb
<i>hidden:</i>	rejtett mező	<i>submit:</i>	„elküld” gomb
<i>image:</i>	képmező	<i>text:</i>	szövegmező

Az objektumok a HTML-kódba a már ismert módon illeszthetők be:

```
<INPUT id = "Azonosító" type = "típus" tulajdonságok>
```

Bár nem *INPUT*-objektumok, de ide sorolhatjuk a görgethető szövegmezőt (*TEXTAREA*) és a listát (*SELECT/OPTION*) is. A vezérlőelemeket legfontosabb tulajdonságaikkal együtt a **8–37. példa** mutatja be. Stílusok alkalmazásával kibővíthetjük a megjelenítési formákat.

A Windows alatt futó programokban gyakran találkozunk ilyen vezérlőelemekkel.

#### A választógombok használata

A választógomboknál az azonos *name* (név) tulajdonságú elemek közül egyszerre csak egy gomb lehet bejelölve. Erről a böngésző automatikusan gondoskodik, ha rákattintunk a csoport egy másik gombjára, akkor törli az előző bejelölést. Célszerű egy csoporton belül egyforma azonosítót (*id*) választani.

A választógomb bejelölését a *checked* (bejelölve) tulajdonság *True* értéke jelzi. Ha egy objektumnál a nyitó tagban is megadjuk ezt a tulajdonságot, akkor a böngésző a weblap megjelenítésénél bejelöli a gombot.

A választógombok más vezérlőelemekhez hasonlóan gyakran rendelkeznek egy *name–value* tulajdonságpárral. Ezeket úgy tekinthetjük, mintha változók nevét és értékét adnánk meg. Űrlapok segítségével továbbküldhetjük a tulajdonságpárokat egy másik weblapnak. Megjegyezzük, hogy a gomb *value* tulajdonságának és a weblapon a gomb mellett megjelenő szövegnek nincsen köze egymáshoz, tetszés szerint megadhatjuk őket.

A **8–38. példában** a címsor betűinek színét választógombok segítségével változtatjuk (egyszerre csak egyféle színű lehet). A gombok *value* tulajdonságaként a szín angol nevét adtuk meg, amit a színezéshez használunk fel. A változtatást végző parancs-

gomb eseménykezelőjében megkeressük a bejelölt választógombot. A vizsgálatnál felhasználjuk, hogy a HTML-kódban az egyik gombnál megadtuk a `checked = True` értéket, tehát egy gomb biztosan be van jelölve (több meg nem lehet):

```
I = 0
Do While Not Szin(I).checked
    I = I + 1
Loop
Cimsor.style.color = Szin(I).value
```

A példa másik választógombcsoportja – teljesen hasonló módon – a címsor igazítását végzi.

## A jelölőnégyzetek használata

A jelölőnégyzetek egymástól független tulajdonságokat változtatnak, bár általában logikailag összetartozó csoportokban jelennek meg a weblapon. A *name* tulajdonságot az űrlapoknál használjuk, de nem kötelező megadni. A bejelölt állapotot itt is a *checked* tulajdonság értéke jelzi (*True* vagy *False*).

A 8–38. példában a címsor betűinek stíluselemeit jelölőnégyzetekkel változtatjuk. Ezek egymástól független tulajdonságok, egyszerre többet is bejelölhetünk. Mivel szükségünk lesz rá, a stíluselem nevét a *name*, értékét pedig a *value* tulajdonsággal adjuk meg. Ezeket használjuk fel a változtatást végző eljárásban.

Az eseménykezelőben egy ciklus segítségével végignézzük a négyzetek állapotát. Bejelölt állapot esetén a *name* által meghatározott stíluselemnek a *setAttribute* metódussal átadjuk a *value* tulajdonság értékét, a többi stíluselem értékét pedig töröljük:

```
For I = 0 To 2
    Set Elem = Stilus(I)
    If Elem.checked Then
        Call Cimsor.style.setAttribute(Elem.name, Elem.value)
    Else
        Call Cimsor.style.setAttribute(Elem.name, "")
    End If
Next
```

A 8–39. példában külön parancsgomb helyett a *document onclick* eseménykezelőjében végezzük el a változtatásokat. Ekkor a bejelölés után azonnal látható az eredmény.

Az objektumot a *type* tulajdonság segítségével választjuk ki. Mivel olyan elemekre is rá lehet kattintani, melyeknek nincsen *type* tulajdonsága, a vizsgálathoz a *getAttribute* metódust kell használni, különben hibaüzenettel leállna a program végrehajtása. Feltételes elágazás alkalmazása helyett mindjárt azt is kiválasztjuk, hogy választógombra vagy jelölőnégyzetre kattintott-e a felhasználó:

```
Set Elem = window.event.srcElement
Select Case Elem.getAttribute("type")
    Case "radio"
        ...
    Case "checkbox"
        ...
```

A módosítás az előző példához hasonlóan történik. A törlésnél üres sztring helyett a *removeAttribute* metódust használjuk, ami szintén törli a tulajdonság értékét.

Emlékeztetjük az Olvasót az objektumok *disabled* tulajdonságára. Ha értékét *True*-ra állítjuk egy jelölőnégyzetnél, akkor szükség esetén letilthatjuk a kiválasztását.

### A lista vezérlőelem

A vezérlőelem alkalmazásával egy listából választhat a felhasználó. Ezzel segítjük az adatbevitelt, és megakadályozzuk a meg nem engedett értékek beírását.

A listát a *SELECT*-objektum tartalmazza. Ne tévesszük össze a *Select Case* utasítással, vagy a szövegmező tartalmát kiválasztó *select* metódussal! A lista minden egyes elemét egy *OPTION*-objektum képviseli a *SELECT*-en belül.

Mivel nem kötelező az *OPTION* záró tagját kiírni, ezért az *innerText* tulajdonsága helyette a *text* tulajdonságát is használhatjuk. A *selected* tulajdonság jelzi a megfelelő listaelem kiválasztott állapotát (*True* vagy *False*).

A *SELECT*-objektum *size* tulajdonsága a lista megjelenítését szabályozza. Ha értéke 1, akkor legördülő listát kapunk. Nagyobb értékek esetén a *size* megszabja az egyszerre látható elemek (sorok) számát. Szükség esetén görgetősáv is megjelenik.

Ha a felhasználó kiválaszt egy listaelemet, akkor a *SELECT*-objektum *value* tulajdonsága felveszi a kiválasztott *OPTION*-objektum *value* tulajdonságának értékét. A változtatást az *onchange* (változás) esemény bekövetkezése jelzi, melynek eseménykezelőjébe írjuk be a végrehajtandó utasításokat.

A **8–40. példában** a háttérszínt egy legördülő listával lehet megváltoztatni. Az *OPTION*-objektumokat egy ciklus segítségével készítjük el. A változtatáshoz szükséges angol elnevezést a *value* tulajdonságban tároljuk.

Az *onchange* esemény kezelőjében a háttérszínt a *value* tulajdonság értékére módosítjuk:

```
Sub Szinek_onchange
    document.bgColor = Szinek.value
End Sub
```

Ha először a listában megjelenő vörösre kattintunk, akkor nem módosítjuk a választást, ezért nem következik be az *onchange* esemény. (Más szín után már működik a vörös is!) Ezért a lista első elemének a helyére célszerű egy tájékoztató feliratot írni, amelynek a választására nem reagálunk. Ennek módját a **8–41. példa** mutatja be.

### A lista elemeinek indexelése

A listaelemek azonosításához nincs szükség a *value* tulajdonságra. A *SELECT*-objektum *selectedIndex* tulajdonsága megadja a kiválasztott elem indexét. Az indexelés nullával kezdődik.

A **8–42. példában** nem használjuk a *value* tulajdonságot. A szín angol nevét a *selectedIndex* tulajdonság segítségével választjuk ki a neveket tartalmazó tömbből. Mivel a lista 0 indexű eleme egy tájékoztató felirat, a tömbelem indexe eggyel kisebb, mint a *selectedIndex* értéke. A tájékoztató felirat kiválasztását itt sem vesszük figyelembe:

```
Sub Szinek_onchange
  If Szinek.selectedIndex > 0 Then
    document.bgColor = AngolSzin(Szinek.selectedIndex - 1)
  End If
End Sub
```

Az OPTION-objektumokat a *document.all* kollekció mellett a SELECT-objektum *options* kollekciójával is elérhetjük. Az elemek számát a *length* tulajdonság adja meg. Így a 8–41. példában a háttérszín megváltoztatását a következő utasítással szintén elvégezhetjük:

```
document.bgColor = Szinek.options(Szinek.selectedIndex).value
```

Maga a SELECT-objektum is tekinthető egy olyan kollekciónak, melynek elemei az OPTION-objektumok, így egyszerűsíthetjük a hivatkozást:

```
document.bgColor = Szinek(Szinek.selectedIndex).value
```

A módosítások kipróbálását az Olvasóra bízunk.

Megjegyezzük, hogy a *selectedIndex* segítségével megváltoztathatjuk a listaelemek tulajdonságainak értékét.

A SELECT-objektum *multiple* (többszörös) tulajdonságának *True* értéke esetén – a Shift vagy a Ctrl billentyűkkel – egyszerre több elem is megjelölhető. Ekkor a megjelölt elemeket a *selected* tulajdonság segítségével választhatjuk ki, melynek használatát a 8–43. példa mutatja be.

Ha nem adjuk meg a *multiple* tulajdonságot, akkor értéke *False* lesz.

## A lista bővítése

Új listaelemet a *document*-objektum *createElement* metódusával lehet létrehozni (8–44. példa):

```
Set UjElem = document.createElement("OPTION")
```

A létrehozás után megadjuk a szükséges tulajdonságokat, melyeket a példában egy-egy szövegmezőből olvasunk be:

```
UjElem.text = MagyarBe.value
UjElem.value = AngolBe.value
```

Majd az új objektumot az *add* metódussal hozzáadjuk a megfelelő kollekcióhoz. A metódus második paramétere az új elem indexe:

```
Call Szinek.add(UjElem, 1)
```

Ha nem adjuk meg az indexet, akkor az új elem a lista végére kerül.

Megjegyezzük, hogy a *createElement* metódussal a dokumentumba más DHTML-objektumok is beilleszthetők, amire a 8.5. fejezetben mutatunk példát.

## Listaelemek törlése

A SELECT-objektum *remove* módszere törli a lista egy elemét. Paraméterként az elem indexét kell megadni.

A 8–44. példában a magyar név szövegmezejébe beírt színt törölni tudjuk a listából. Indexének meghatározásához felhasználjuk az 5.5 fejezetben szereplő *Keres* függvényt, melynek visszatérési értéke a keresett elem indexe. Sikertelen keresés esetén a függvény -1-et ad vissza.

A függvényben a lista elemeinek indexelését a SELECT kollekciójának segítségével végezzük. A listaelemek számát az objektum *length* tulajdonsága adja meg:

```
Do While Szinek(Hely).text <> Szin And _  
    Hely < Szinek.length - 1  
    Hely = Hely + 1  
Loop
```

Ha megtaláltuk az elemet, akkor töröljük:

```
Szinek.remove(Index)
```

A 8–44. példában nem engedjük meg a magyarázó felirat törlését. Nem létező szín esetén is hibaüzenetet adunk.

## A vezérlőelemek elérése

Bár a legtöbb felhasználó az egér segítségével helyezi át a fókuszt a megfelelő objektumra, a Windows lehetőséget biztosít arra, hogy ugyanezt a billentyűzettel is megtegyük. Nagy mennyiségű adat bevitelét lelassítja és kényelmetlenné teszi, ha felváltva kell használni a két eszközt.

A képernyőn megjelenő objektumok között a tabulátor billentyűvel és az úgynevezett gyorsbillentyűk segítségével válthatunk. Ez utóbbi módszernél az Alt billentyűvel együtt kell lenyomni az objektumnál aláhúzással megjelölt betűt.

*Váltás a tabulátor billentyűvel*

A böngésző automatikusan biztosítja a tabulátor billentyűvel történő váltást. A fókuszt a HTML-kód sorrendjében halad végig a vezérlőelemeken, de fölmegegy a böngésző címsorába is, ahol a fájl elérési útja látható. A sorrendet az objektumok *tabIndex* tulajdonságának használatával módosíthatjuk.

Ha megadjuk, akkor a fókuszt a *tabIndex*-ek növekvő sorrendjében halad végig az elemeken. Utánuk következnek azok az objektumok, melyeknek nincs *tabIndex* tulajdonsága, de a választógombok esetén egy csoporton belül csak az egyik gomb kerül sorra. A többire a kurzorvezérlő billentyűkkel léphetünk. A jelölőnégyzetek állapotát a szókész billentyűvel változtathatjuk meg.

A *tabIndex* használatát a **8–45. példa** mutatja be. A *focus* módszer segítségével megadtuk, hogy a parancsgommbal kezdődjön a sor.

Ha egy objektum *tabIndex* tulajdonságát -1-re állítjuk, akkor a tabulátor billentyűvel nem vihető rá a fókuszt. Az egér segítségével persze ekkor is kiválaszthatjuk.

### *Gyorsbillentyűk használata*

Gyorsbillentyűt az *accessKey* tulajdonsággal rendelhetünk az objektumhoz. Az *accessKey* értéke az a karakter, melyet az Alt-tal együtt lenyomva az objektum megkapja a fókuszt. A gyorsbillentyű betűjét nekünk kell aláhúzással jelölni a weblapon. Figyeljünk arra, hogy az *accessKey* tulajdonságot a vezérlőelemnél adjuk meg, a betűt viszont a feliratánál húzzuk alá! Az objektum akkor is az *accessKey*-nek megfelelő billentyűvel érhető el, ha másik betűt jelölünk meg!

A parancsgombok esetén a gyorsbillentyű lenyomása egyben létrehozza az *onclick* eseményt is, a jelölőnégyzeteknél pedig megváltoztatja a bejelölt állapotot.

Az *accessKey* alkalmazását a **8-46. példa** mutatja be. Mivel a parancsgombok *value* tulajdonsága nem teszi lehetővé egy betű aláhúzását, ezért a *button* típusú INPUT-objektum helyett a vele egyenértékű BUTTON-objektumot használtuk, melynek nyitó és záró tagja közé kell írni a gomb feliratát. Itt már tetszőleges formátumot megadhatunk.

### **Az űrlapobjektum**

Az Interneten gyakran találkozunk olyan weblapokkal, melyek a vezérlőelemek kitöltése után elküldik adatainkat a webkiszolgálónak. Ezt a FORM (űrlap) objektum segítségével hajtják végre.

A FORM konténerobjektum, amely nem jelenik meg a képernyőn, de tetszőleges DHTML-objektumokat tartalmazhat. Az Elküld (*submit*) gombra való kattintás után a vezérlőelemek *name-value* értékpárjait (a bejelölt választógombot, illetve a kiválasztott listaelemet) elküldi a webkiszolgálónak. Az űrlapon általában elhelyezünk egy Alaphelyzet vagy Mégse (*reset*) gombot is, amely az űrlapmezők eredeti értékét állítja vissza.

A FORM-objektum *action* tulajdonsága adja meg annak a dokumentumnak az elérési útját, amelyet a böngésző letölt az Elküld gombra való kattintás után. Ennek a funkciónak a használatához nincsen szükség webkiszolgálóra. A **8-47. példában** az Elküld gombra való kattintás hatására a böngésző mindenféle eseménykezelő eljárás nélkül betölti a Minta almappából a Felel.htm weblapot:

```
<FORM action = "Minta\Felel.htm">
```

A rovatok kitöltése után próbáljuk ki a „Mégse” gomb működését!

A Felel.htm lapon elhelyeztünk egy parancsgombot, amellyel visszaléphetünk az űrlapra, bár ezt a felhasználó a böngésző eszköztárában található Vissza gombbal is megteheti. A Vissza gomb *onclick* eseménykezelőjében a *window*-objektum *navigate* metódusát használjuk, amely a paramétereként megadott dokumentumot megjeleníti az ablakban (a két pont a szülőkönyvtárat jelöli):

```
window.navigate("../Pelda8-47.htm")
```

Mind az *action* tulajdonságnál, mind pedig a *navigate* metódusnál az Interneten található weblapokat is megadhatunk.

A FORM-objektum elkülöníti az elemeit a dokumentumtól. A tulajdonságokra és metódusokra az űrlap azonosítójával minősítve kell hivatkoznunk:

```
ŰrlapAzonosító.VezérlőelemAzonosító.value
```

Ez a szabály nem vonatkozik az eseménykezelő eljárások nevére. Az űrlapon belül elhelyezkedő objektumhoz kapcsolódó eseménykezelőt a szokásos módon nevezzük el, például:

```
VezérlőelemAzonosító_onclick
```

Ha a fókusz az űrlap egy vezérlőelemén áll, akkor az ESC billentyű lenyomása egyenértékű a Mégse, az Enter lenyomása pedig az Elküld gombra való kattintással. Mindkét művelet esetén bekövetkezik az *onclick* esemény.

Megjegyezzük, hogy egy dokumentum több űrlapobjektumot tartalmazhat. Az egyes űrlapokat a *document*-objektum *forms* kollekciójával szintén elérhetjük. Egy űrlapobjektumon belül több *submit*, illetve *reset* gombot is elhelyezhetünk.

### Az onsubmit és onreset események

Az űrlapokat adatok továbbítására fogjuk használni. A továbbítás előtt célszerű ellenőrizni a szövegmezőkbe írt értékeket. Az ellenőrzést az *onsubmit* eseménykezelőben végezhetjük el.

Az *onsubmit* esemény akkor következik be, amikor a felhasználó rákattint egy *submit* típusú parancsgombra. Ha le akarjuk állítani az adatok továbbítását és az új weblap betöltését, akkor az eseménykezelőben a *returnValue* tulajdonságnak *False* értéket adunk:

```
window.event.returnValue = False
```

A **8–48. példában** megvizsgáljuk, hogy a felhasználó kitöltött-e minden adatot, és elfogadható évszámot írt-e be. Ha nem, akkor hibaüzenetet küldünk, és meggátoljuk az új weblap betöltését. A példában az *onreset* esemény kezelését is bemutatjuk, melynek bekövetkeztekor egy párbeszédablak figyelmezteti a felhasználót a törlésre.

Az *onsubmit* és *onreset* események nem vesznek részt az eseménybuborék terjedésében, így csak az űrlapobjektumra vonatkoznak.

## 8.4. Kommunikáció a weblapok között

Előző példánkban ugyan kitöltöttük az űrlap szövegmezőit, de nem tettünk semmit a beírt karakterláncokkal. Az *action* tulajdonsággal megadott dokumentum azonban képes az adatok fogadására, így a két weblap kommunikálni tud egymással.

### Az űrlapadatok továbbítása

Az adatok továbbításához kétféle módszert használhatunk, melyek közül a FORM-objektum *method* (metódus, módszer) tulajdonságával választunk. A *method* értéke *get* vagy *post* lehet.

A *post* módszer hatékony lehetőséget biztosít az Interneten történő adatküldéshez, de csak webkiszolgáló képes fogadni az adatokat. A *get* módszert akkor is használhat-



juk, ha weblapjaink egymásnak üzennek. Ha nem írjuk elő a *method* tulajdonság értékét, a böngésző a *get*-et alkalmazza. Mi azonban hangsúlyozni szeretnénk, hogy melyik módszert használjuk az adatok küldésére, ezért mindig meg fogjuk adni a *method* értékét HTML-kódban.

A *get* használata esetén a böngésző az *action* tulajdonságnál megadott elérési utat kiegészíti az űrlapmezők *name–value* értékpárjaival. Ezt a kiegészítést keresőkifejezésnek nevezzük, mert a weblapok keresését végző webhelyek alkalmazták először.

A 8–47. példa weblapján töltjük ki a szövegmezőket, és kattintsunk az Elküld gombra. Figyeljük meg, hogy az eszköztár alatt, az úgynevezett címsorban (ne keverjük össze magának az ablaknak a címsorával) a fájlnev után megjelenik a keresőkifejezés! Egy kérdőjelet követően, & jelekkel elválasztva megkapjuk a *name* és *value* tulajdonságokat. Az értékpárokat egyenlőségjel köti össze:

```
file:///D:/Fejezet08/Minta/Felel.htm?
                                Nev=Kiss+Lajos&SzulEv=1972&SzulHely=Budapest
```

A böngésző a szóközöket + jelekkel, az írásjeleket és az ékezetes karaktereket pedig úgynevezett escape kódokkal helyettesíti.

A *get* használata esetén ügyelnünk kell arra, hogy a címsor a böngészőtől függően akár 3000 karakterből is állhat, de az Interneten az adatokat továbbító számítógépek (routerek) ezt a méretet jelentősen csökkenthetik.

## Az adatok dekódolása

Bár egy webkiszolgáló közvetlenül képes kiolvasni a keresőkifejezésből az adatokat, nélküle most magunknak kell a dekódolást elvégeznünk. Az egyes lépéseket a 8–49. példa segítségével követhetjük nyomon. Írjunk be adatokat a szövegmezőkbe, és kattintsunk az Elküld gombra.

A Minta\Válasz.htm fájlban először egy változóban tároljuk a *document*-objektum *location* tulajdonságának *search* elemét, amely a kérdőjeltől kezdve megadja a keresőkifejezést:

```
Temp = document.location.search
```

majd elhagyjuk az elejéről a kérdőjel karaktert:

```
Temp = Right(Temp, Len(Temp) - 1)
```

Az írásjeleket és az ékezetes karaktereket a már ismert *Unescape* függvénnyel alakítjuk vissza:

```
Temp = Unescape(Temp)
```

Végül a + jeleket szóközökkel helyettesítjük (itt a kódolás kissé eltér attól, amivel a képjelöltek *src* tulajdonságánál találkoztunk):

```
Temp = Replace(Temp, "+", " ")
```

Ezeket a lépéseket természetesen egyetlen kifejezésben is leírhattuk volna.

## Az adatok szétválasztása

Az értékpárok szétválasztásához a *Split* függvényt használjuk. A *Temp* karakterláncát az & karaktereknél szétvágjuk, a sztringrészeket pedig az *AdatParok* tömb elemeibe írjuk:

```
AdatParok = Split(Temp, "&")
```

Emlékezzünk vissza arra, hogy a *Split* elhagyja a határoló karaktereket.

Utolsó lépésként a tömb minden egyes eleméből töröljük az "=" karakterig tartó részt. Az így kapott sztringek megadják az elküldött értékeket:

```
For I = 0 To 3
    Kezd = InStr(AdatParok(I), "=")
    Adatok(I) = Right(AdatParok(I), Len(AdatParok(I)) - Kezd)
Next
```

Ha nem ismerjük az adatként számát, akkor az *Adatok* statikus tömb helyett dinamikus tömböt használhatunk, melynek méretét az `UBound(AdatParok)` értéke adja meg.

Ha nem magunk állítottuk össze az űrlapot, akkor természetesen a *name* tulajdonságok megnevezésére is kíváncsiak vagyunk. Ekkor a *Right* függvény helyett a *Split*-et használjuk. Bár az eljárás kissé hosszadalmasnak tűnhet, mindig pontosan ugyanezt kell tenni, így a megírt kódot bárhová beilleszthetjük. A CD-n található példában a megjegyzéssel ellátott sorok nem tartoznak hozzá az algoritmushoz, csak a részeredmények kiírását végzik.

Az űrlapok segítségével elválaszthatjuk egymástól az adatok beírását, és az eredmények megjelenítését. Ennek bemutatására a **8–50. példában** kiegészítettük az előző példa kódját a horoszkóp értékelésével. A szkriptben a bevitt nem ellenőriztük. A horoszkóp intelmeit ne vegye túl komolyan az Olvasó!

## Rejtett vezérlőelemek használata

Az űrlapmezők felsorolásánál említettük, hogy a *hidden* típusú INPUT-objektumok nem jelennek meg a weblapon. Segítségükkel a felhasználó elől elrejtett adatokat továbbíthatunk. A **8–51.** és **8–52. példa** látszólag egyforma weblapokat jelenít meg. Mind a kettő ugyanazt a `Minta\Titkos.htm` fájlt tölti be az Elküld gombra kattintás után. Tartalmaznak azonban egy *hidden* típusú vezérlőelemet, melynek értéke különbözik a két fájlban:

```
<INPUT type = "hidden" name = "Enged" value = "Igen">, illetve
<INPUT type = "hidden" name = "Enged" value = "Nem">
```

A `Titkos.htm` fájlban a dekódolás után megvizsgáljuk az *Enged* értékét, és ennek megfelelően üzenünk a felhasználónak.

Szkriptekkel a többi objektumhoz hasonlóan a rejtett mezők tulajdonságait is megváltoztathatjuk.

## A kép vezérlőelem

Röviden bemutatjuk a képmező használatát. Az *images* típusú INPUT-objektum a *submit* gombot helyettesíti. Kattintás esetén az űrlap többi vezérlőelemének névtulajdonság értékpárjai mellett elküldi a kattintás helyének x- és y-koordinátáit a kép bal felső sarkához viszonyítva.

A **8–53. példában** egy 96x66 pixel méretű képet használunk, amely parancsgombot formáz. A kattintásra betöltött Minta\Kártya.htm fájlban a dekódolás után az x- és y-koordinátáknak megfelelő kártyalapot jelenítjük meg a képernyőn. Mivel a koordinátákon kívül most más adatokat nem küldünk, a dekódolás sokkal egyszerűbb. Az x-koordináta értéke az első egyenlőségjel és & jel között helyezkedik el:

```
Temp = document.location.search
Kezd = InStr(Temp, "=") + 1
Veg = InStr(Temp, "&")
X = Mid(Temp, Kezd, Veg - Kezd)
```

Az y-koordináta értéke az utolsó egyenlőségjel után áll:

```
Kezd = InStrRev(Temp, "=")
Y = Right(Temp, Len(Temp) - Kezd)
```

A koordináták meghatározása után feltételes elágazásokkal megállapítjuk a kért lap típusát, és elvégezzük a megjelenítést. A szint véletlenszerűen választjuk ki egy tömb segítségével. Az algoritmust a CD-n lévő fájl forráskódja tartalmazza.

A kép vezérlőelem csak akkor működik az itt bemutatott módon, ha egy űrlapon belül helyezzük el.

Megemlíjtük, hogy az AREA-objektum segítségével egy kép (IMG-objektum) bármely kör vagy sokszög alakú területéhez rendelhetünk hivatkozást. Ennek módját a HTML-ről szóló szakkönyvekben találja meg az Olvasó.

## Adattárolás sütik segítségével

A kereső kifejezések mellett az Interneten egy másik kommunikációs lehetőséget is gyakran felhasználnak. Bármely weblap adatokat tárolhat egy speciális fájlban, az úgynevezett cookie-ban, amelyhez más dokumentumok szintén hozzáférhetnek. A cookie-t eléggé idétlen és szolgai fordítással sütinnek nevezik. Sajnos nagyon elterjedt ez az elnevezés, így mi is ezt használjuk.

A sütiket a böngésző tárolja a számítógépünkön. A webkiszolgálókhöz hasonlóan szkriptekkel írhatjuk, olvashatjuk ezeket a fájlokat. Biztonsági okokból korlátozható az alkalmazásuk. Az Internet Explorer az Eszközök/Internetbeállítások menüben az Adatvédelem fülnél teszi lehetővé a sütik használatának tiltását vagy engedélyezését. A beállításhoz kattintsunk a Speciális gombra, és szükség esetén jelöljük be a belső cookie-k elfogadását.

**FIGYELEM!**

A süti használatát bemutató szkriptek működéséhez engedélyezni kell a böngészőben a belső cookie-k elfogadását. A cookie-ban tárolt adatok törléséhez az egyes példák betöltése és kipróbálása után be kell zárni a böngésző ablakát, majd a következő példánál újra megnyitni.

A süti a kereső kifejezéshez hasonlóan változónév–érték párokat tárolnak, kiegészítve az érvényességi körrel és az érvényességi idővel. Az érvényességi kört a böngésző fűzi a fájlhoz. Azok a dokumentumok tartoznak hozzá, melyek elérhetik a sütit. Az érvényességi kör esetünkben a fájl tartalmazó mappa elérési útja lesz. A sütit az őt létrehozó dokumentummal azonos mappában, vagy annak almappáiban lévő weblapok olvashatják és módosíthatják.

Az érvényességi időt nekünk kell megadnunk. Az érvényességi idő lejártá után a süti törlődik a háttértárról. Ha nem adunk meg időpontot, akkor a süti a böngészőből való kilépésnél törlődik, és nem kerül a háttértárra.

**A süti írása és olvasása**

A süti tartalma a *document*-objektum *cookie* tulajdonságával érhető el. Sztringeket írhatunk bele, melyeknek tartalmaznia kell az egyenlőségjellel összekötött változónév–érték párokat. Ha több értékadás szerepel, a böngésző mindegyiket tárolja a süti-ben:

```
document.cookie = "Nev=Kovács Laci"  
document.cookie = "SzulEv=1972"
```

A sütikben tárolt változóneveknél tartsuk be a VBScript vonatkozó előírásait. A változók értéke ne tartalmazzon pontosvesszőt!

A **8–54. példában** az így tárolt süti értékét a *document.write* utasítással jelenítjük meg:

```
document.write(document.cookie)
```

Mint látjuk, egyetlen sztringet kaptunk, melyben az értékpárokat egy pontosvessző és egy szóköz választja el egymástól.

A **8–55. példában** egy almappában lévő dokumentum jeleníti meg az állomány által elkészített süti tartalmát. Kipróbálása után zárjuk be a böngésző ablakát, és csak a Süti\Megjelenít.htm fájl nyissuk meg. Láthatjuk, hogy a süti tartalma megsemmisült.

Ha a süti-ben létező változónévhez rendelünk újabb értéket, akkor a böngésző megváltoztatja az előző értéket. A **8–56. példában** a szövegmezők tartalmát tároljuk a süti-ben, majd megjelenítjük az értékét. A tárolást sztringek összefűzésével végezzük:

```
document.cookie = "Nev=" & NevBe.value  
document.cookie = "SzulEv=" & SzulEvBe.value
```

A megjelenítéshez beírjuk a süti tartalmát egy bekezdés-objektumba:

```
Suti.innerText = document.cookie
```

Egy sütifájlban legfeljebb 20 név–érték pár tárolható. Ezeket a sütit létrehozó dokumentum mappájában vagy almappájában lévő fájlokból érhetjük el (érvényességi kör).

## A süti használata

A sütiben tárolt adatok eléréséhez a kereső kifejezéseknél bemutatott eljárást alkalmazzuk. A *document.cookie* sztringjéből ki kell emelni az egyenlőségjelek és pontosvesszők közé eső részeket, így megkapjuk a változók értékét.

A **8–57. példában** négy változót tárolunk a sütiben, majd a Süti\Szétválaszt.htm fájlban szétválasztjuk a neveket és értékeket. Mivel ugyanazt a sütit használjuk, mint az előző példák, először zárjuk be a böngésző ablakát, majd nyissuk meg újra. Ezzel töröljük az eddig tárolt változókat.

A nevek és értékek szétválasztáshoz a *Split* függvénnyel a pontosvesszőknél szétvágjuk a sütit:

```
Suti.innerText = Temp
AdatParok = Split(Temp, "; ")
```

majd minden részben megkeressük az egyenlőségjeleket:

```
For I = 0 To 3
    Kezd = InStr(AdatParok(I), "=")
```

Ha nincs egyenlőségjel, akkor a felhasználó nem adott meg értéket, a sütiben csak a változó neve szerepelt:

```
If Kezd = 0 Then
    Nev(I) = AdatParok(I)
    Ertek(I) = ""
```

Egyébként pedig az egyenlőségjeltől balra a változó neve, jobbra pedig az értéke áll:

```
Else
    Nev(I) = Left(AdatParok(I), Kezd - 1)
    Ertek(I) = Right(AdatParok(I), Len(AdatParok(I)) - Kezd)
End If
Next
```

A **8–58. példában** süti segítségével adjuk át a horoszkóp elkészítéséhez szükséges adatokat a SütiHoroszkóp.htm fájlban. Itt nincs szükségünk a FORM-objektumra.

## Az érvényességi idő beállítása

Az eddigi példákban szereplő név–érték pároknál nem adtuk meg az érvényességi időt, így a böngészőből való kilépés után törlődtek. A tároláshoz az érvényességi időt minden egyes változó után pontosvesszővel elválasztva, az *expires* (lejár) tulajdonság értékeként kell beírni a sütibe, egy speciális, úgynevezett GMT formátumban. A

```
document.cookie = "Nev=Kovács Laci" & _
    "; expires=Friday, 31-Dec-2004 23:59:59 GMT"
```

értékadás hatására például a süti 2004 végéig tárolja az adatokat.

Megjegyezzük, hogy a hét napjának angol elnevezését nem ellenőrzi a böngésző, tehát bármilyen szót beírhatunk a helyére. A fenti formát azonban pontosan be kell tartanunk. Ha egy név–érték párnál a rendszer dátumnál régebbi időpontot adunk meg, akkor a böngésző kilépéskor törli a sütiből a nevet és az értéket.

A *document.cookie* tulajdonság nem adja vissza az érvényességi időt, így az adatok kiemeléséhez továbbra is a fentiekben részletezett algoritmust használjuk.

A CD-melléklet Számláló mappájában található Számlál.htm fájl a süti segítségével számlálja a betöltéseket és frissítéseket. Azért helyeztük külön mappába, hogy ne zavarja meg a fejezetet többi példáját. Emlékezzünk vissza arra, hogy az azonos mappában lévő fájlok közös sütit használnak. A betöltések számát a *Szamlalo* nevű változóban tároljuk.

A kód egyszerűsítéséhez kihasználhatnánk, hogy a süti egyetlen adatot tartalmaz, de ha egy másik fájl is ugyanebbe a mappába kerül, akkor az hibás működéshez vezethet. Ezért a sütiben megkeressük a *Szamlalo* nevű értéket:

```
Temp = document.cookie  
Kezd = InStr(Temp, "Szamlalo=")
```

Ha nem találjuk meg (az *InStr* értéke 0), akkor először töltődött be az állomány:

```
If Kezd = 0 Then  
    Szamlalo = 1
```

Ha már rendelkezik valamilyen értékkel, akkor az egyenlőségjel után megkeressük a pontosvesszőt. Erre azért van szükség, mert ugyanez a süti egy másik fájl változóit is tartalmazhatja:

```
Else  
    Kezd = Kezd + 9  
    Veg = InStr(Kezd, Temp, "; ")
```

Ha nincs benne pontosvessző, akkor a süti egyetlen adatot tárol. A *Veg* változót ráállítjuk a szám utáni karakterre:

```
If Veg = 0 Then  
    Veg = Len(Temp) + 1  
End If
```

Kiemeljük a *Kezd* és a *Veg* által kijelölt részt, majd megnöveljük az értékét:

```
Szamlalo = CInt(Mid(Temp, Kezd, Veg - Kezd))  
Szamlalo = Szamlalo + 1  
End If
```

A számláló új értékét visszaírjuk a sütibe, és megjelenítjük a weblapon:

```
document.cookie = "Szamlalo=" & Szamlalo & _  
    "; expires=Thursday, 31-Dec-2020 23:59:59 GMT"  
SzamlaloKi.innerText = Szamlalo
```

Ez a fájl 2020 végéig számlálja a betöltéseket, vagy addig, amíg le nem töröljük a sütit a háttértárról. Megemlítjük, hogy az érvényességi idő megadásakor a böngésző a süti tartalmát a következő mappába menti:

Windows 98: c:\Windows\Profiles\felhasználónév\Cookies  
Windows 2000 és XP: c:\Documents and Settings\felhasználónév\Cookies

Az állománynév a felhasználó és az érvényességi körnek megfelelő mappa nevéből áll. Ha a gyakorlás során összekeverednek a különböző weblapokhoz tartozó értékpárok, akkor töröljük ki a megfelelő cookie-fájlt.

## 8.5. Objektumok beillesztése

Weblapjainkra elkészülésük után is illeszthetünk be DHTML-objektumokat. Külső objektumok segítségével pedig megnövelhetjük hatékonyságukat, és látványosabb dokumentumokat hozhatunk létre. A továbbiakban ezeket a lehetőségeket tekintjük át.

### DHTML-objektumok beillesztése

Az űrlapok készítésénél már bemutattuk, hogyan bővíthetjük, illetve törölhetjük egy lista elemeit. A *document*-objektum *createElement* metódusával nem csak lista-elemet, hanem tetszőleges DHTML-objektumot is létrehozhatunk.

A *createElement* visszaadja az új objektumra való hivatkozást, amit hozzá kell rendelnünk egy objektumváltozóhoz:

```
Set Változónév = document.createElement(osztálynév_vagy_HTML_kód)
```

A metódus paramétere egy DHTML-osztály neve, vagy egy objektum érvényes HTML-kódja. Ez utóbbi esetben közvetlenül is megadhatjuk az új objektum tulajdonságait:

```
Set UjKep = document.createElement("<IMG src = 'Kocka\K1.gif'>")
```

Az objektumot a létrehozása után az *insertAdjacentElement* (szomszédos elem beillesztése) metódussal adjuk hozzá a dokumentumhoz. Ezzel egyben meg is jelenítjük a képernyőn.

Az *insertAdjacentElement* egy már létező DHTML-objektum metódusa. Az új objektumot be lehet illeszteni a nyitó tagja elé (*beforeBegin*), a nyitó tagja után, de a tartalma elé (*afterBegin*), a záró tag elé, de a tartalma után (*beforeEnd*), illetve a záró tag mögé (*afterEnd*). A helyet és az új objektumra való hivatkozást a metódus paramétereként kell megadni:

```
LétezőObjektum.insertAdjacentElement("hely", UjObjektum)
```

A **8–59. példában** egérekattintásra dobókockákat jelenítünk meg a weblapon. A kockán lévő számot véletlenszerűen választjuk ki. A kép az egérekattintás helyére kerül. Az új objektum létrehozása után megadjuk a tulajdonságait, majd beillesztjük a dokumentumba:

```
I = Int(6 * Rnd() + 1)  
Set Kocka = document.createElement("IMG")  
Kocka.src = "Kocka\K" & I & ".gif"
```

```
Kocka.style.position = "absolute"
Kocka.style.left = window.event.x
Kocka.style.top = window.event.y
Call Torzs.insertAdjacentElement("beforeEnd", Kocka)
```

A fenti utasítások helyett használhattuk volna a

```
Kocka = "<IMG src = 'Kocka\K" & I & ".gif'" & _
        " style = 'position: absolute" & _
        "; left: " & window.event.x & _
        "; top: " & window.event.y & ">"
Set Kocka = document.createElement(Kocka)
Call Torzs.insertAdjacentElement("beforeEnd", Kocka)
```

utasítássorozatot is, ahol először összeállítjuk az IMG-objektumot leíró sztringet, majd létrehozuk és beillesztjük az objektumot.

### DHTML-objektumok törlése

A dokumentumból törölhetjük is az objektumokat. Ehhez az objektum *removeNode* metódusát használjuk fel. A metódus paramétere egy logikai érték, amely jelzi, hogy az objektum tartalmát képező objektumok szintén törlődjenek-e (*True*):

```
Objektumváltozó.removeNode(a_tartalom_törlése)
```

Alapértelmezett esetben (*False*) az objektum tartalma megmarad.

A **8–60. példában** a képernyőn megjelenő dobókockákat újabb kattintással törölni lehet. A törléshez megvizsgáljuk, hogy milyen objektumra kattintottunk:

```
If window.event.srcElement.tagName = "IMG" Then
    ' az objektum törlése
Else
    ' újabb kép létrehozása és megjelenítése
End If
```

Az objektumot az előző példában látott módon hozzuk létre. A törlést végző utasítás:

```
window.event.srcElement.removeNode()
```

### Hanggenerálás

Mint említettük, weblapjainkra nem csak DHTML-objektumokat, hanem bármilyen fájlt beilleszthetünk, melyet a böngésző felismer és képes a kezelésére. A beillesztést az EMBED (beágyaz) objektum segítségével végezzük el, melynek csak nyitó tagja van. Az *src* tulajdonságával adjuk meg az állomány elérési útját. A **8–61. példa** kódjában egy közismert dallamot szólaltatunk meg:

```
<EMBED id = "Zene" SRC="Hang\Promenád.wav" autostart = False>
```

A böngésző a beillesztés hatására megjeleníti a médialejátszó kezelőpultját. Ha nem állítjuk az *autostart* tulajdonság értékét hamisra, akkor egyből elindul a lejátszás. Az objektum *width* és *height* tulajdonságaival a médialejátszó megjelenését szabályozzuk. Ezzel elkerüljük a lejátszófelület felvillanását a betöltésnél (**8–62. példa**).



Ha a beillesztett objektum *hidden* tulajdonságát *True*-ra állítjuk, akkor a lejátszó nem jelenik meg a képernyőn. A **8–63. példában** egy eseménykezelőből indítjuk a lejátszást az objektum *play* (lejátszás) módszerével:

```
Zene.play()
```

A **8–64. példában** hibás adatbevitel esetén a piros feliratok mellett hangjelzéssel is figyelmeztetjük a felhasználót.

A *c:\Windows\Media* mappában további hangfájlokat talál az Olvasó. Javasoljuk, hogy próbálja meg beilleszteni őket a weblapra!

A hangfájlokhoz hasonlóan olyan videófájlokat is felhasználhatunk, melyeket a böngésző felismer. Megjegyezzük, hogy a hang- és videófájlokat az úgynevezett ActiveMovie vezérlőelem jeleníti meg, melynek tulajdonságait a **8–65. példa** mutatja be.

## ActiveX komponensek

Anélkül, hogy túlságosan elmerülnénk a részletekben, az ActiveX komponenseket olyan objektumosztályoknak tekintjük, amelyek kibővítik a Windows-alkalmazások lehetőségeit. A komponensek segítségével ActiveX objektumokat hozhatunk létre, melyeket a dokumentum megjelenítésénél, illetve a szkriptekben egyaránt felhasználhatunk, elérhetjük tulajdonságaikat, meghívhatjuk metódusaikat.

Az ActiveX komponenseket több csoportba soroljuk. A kódkomponensek a programokból meghívható alprogramokat tartalmazzak, a vezérlőelem-komponensek megjeleníthetők a weblapon, az automatizmus komponensek (Automation component) segítségével kezelhetjük más Windows-alkalmazások objektumait.

Az ActiveX komponensek az ügyfél–kiszolgáló modell szerint működnek együtt a dokumentumainkkal. A kiszolgáló szerepét a komponens segítségével létrehozott objektumok játsszák, az ügyfél pedig a szolgáltatásokat felhasználó dokumentum vagy szkript. A modell szerint az ügyfél szolgáltatásokat kér a kiszolgálótól, a kiszolgáló pedig biztosítja ezeket a szolgáltatásokat.

Az ActiveX komponensek egy hosszú (128-bites) osztály-azonosítóval (*classID*) rendelkeznek, amely lehetővé teszi a Windows számára az osztály egyértelmű azonosítását. A *Windows\System* illetve *System32* mappa *.dll* és *.ocx* kiterjesztésű fájljaiban sok ActiveX komponens található.

## A rendszerleíró adatbázis

A számítógépen elérhető ActiveX komponenseket a Windows rendszerleíró adatbázisa (system registry) tartalmazza. Az egyes programok telepítése során az operációs rendszer feljegyzi az adatbázisba a telepített ActiveX komponensek osztály-azonosítóját. Ezt a folyamatot regisztrálásnak hívjuk. Csak olyan ActiveX objektumokat hozhatunk létre, melyek osztálya regisztrálva van a rendszerleíró adatbázisban.

A regisztrált osztályokat a Registry Editor segítségével tekinthetjük meg. Indításához kattintsunk a Start gombra, majd válasszuk a Futtatás menüpontot. A szövegmezőbe gépeljük be a *regedit* szót, és kattintsunk az OK gombra.

### FIGYELEM!

A rendszerleíró adatbázis hibás módosítása súlyos fennakadásokat okozhat az operációs rendszer működésében. Ne töröljük és ne változtassuk meg a bejegyzéseket!

Az ActiveX komponensek osztályazonosítóit a Registry Editor

HKEY\_LOCAL\_MACHINE\SOFTWARE\Classes

mappájában találjuk. Itt kereshetjük meg a számunkra szükséges osztályokat. A kereséshez válasszuk a Szerkesztés/Keresés menüpontot, és gépeljük be például a progressbar szót. Ezzel egy folyamatjelző komponenst fogunk keresni. Minden bizonnyal több ilyen bejegyzést is találunk. A keresést az F3 billentyű lenyomásával folytathatjuk. Ismételgessük addig, amíg rátalálunk a

Microsoft ProgressBar Control, version 6.0

vezérlőelemre. A megnevezést a jobb oldali ablakban láthatjuk az Érték oszlopában. A komponenshez tartozó osztályazonosító:

35053A22-8589-11D1-B16A-00C0F0283628

Ez a bal oldali ablakban a nyitott mappa mellett jelenik meg.

Ha a számítógépre nem a Microsoft Internet Explorer 6-os változatát telepítettük, akkor próbálkozhatunk más folyamatjelző komponenssel is (például „Flat ScrollBar Control”).

### ActiveX vezérlőelem beillesztése a dokumentumba

### FIGYELEM!

Az alábbi példák csak akkor működnek, ha a felhasznált ActiveX komponens szerepel a rendszerleíró adatbázisban.

Az ActiveX vezérlőelemeket az OBJECT-objektum segítségével tudjuk beilleszteni a dokumentumba. A vezérlőelem *classid* tulajdonsága szabja meg a komponens osztályazonosítóját, melyet a rendszerleíró adatbázisból másolhatunk át. A másolást az egér jobb oldali gombjával is elvégezhetjük. A Registry Editorban válasszuk ki a „Kulcsnév másolása”, a HTML-kódban pedig a Beillesztés menüpontot. Töröljük ki a fölösleges karaktereket és írjuk be a kód elejére a „clsid:” megjelölést (**8–66. példa**):

```
<OBJECT id = "PBar"  
        classid = "clsid:35053A22-8589-11D1-B16A-00C0F0283628"
```

Adjuk meg a folyamatjelző szélességét (hosszát) és magasságát:

```
width = 200 height = 20>
```

A folyamatjelző értékét az objektum *value* tulajdonsága határozza meg. A példában ezt egy szövegmezőbe írhatjuk be:

```
PBar.value = AdatBe.value
```

Az ActiveX vezérlőelemek paraméterekkel is rendelkezhetnek. A paraméterek nevét és értékét a PARAM-objektumok *name*, illetve *value* tulajdonságaival adjuk meg az OBJECT nyitó és záró tagja között. A folyamatjelző vezérlőelem minimális és maximális értékét például a következőképpen állíthatjuk be (**8–67. példa**):

```
<PARAM name = "Min" value = 10>
<PARAM name = "Max" value = 20>
```

Így már csak 10 és 20 közé eső értékeket adhatunk át az ActiveX objektumnak, különben hibaüzenetet kapunk. Felhívjuk a figyelmet arra, hogy az OBJECT nyitó tagjában szereplő tulajdonságokat a böngésző használja fel az objektum azonosításához és megjelenítéséhez. A paraméterek értékét viszont az ActiveX vezérlőelem kapja meg, ezekkel szabályozzuk a működését.

A **8–68. példában** egy hosszú ciklus futását szemléltetjük a folyamatjelző alkalmazásával.

## A vezérlőelemek eseményei

Az ActiveX komponenseket bármely programnyelven meg lehet írni, amely képes kommunikálni a Windows-zal. A vezérlőelem tulajdonságait, eseményeit a készítője szabja meg. A komponensek által létrehozott események elnevezésére gyakran a VBScriptben használt szavakat használják, de az *on* előtag nélkül. Egy ActiveX komponens létrehozhat például *click*, *keypress*, *change* stb. eseményeket.

A **8–69. példában** görgetősáv vezérlőelemmel változtatjuk a háttérszín vörös, zöld és kék összetevőit. Az osztályazonosítóhoz a *scrollbar* szót kerestük meg a rendszerleíró adatbázisban. A „Microsoft Forms 2.0 ScrollBar”-t választottuk, de a „Flat ScrollBar Control”-t is kipróbálhatjuk. A *value* tulajdonság értéke a két szélső helyzetben itt is a *Min* és *Max* paraméterekkel állítható be.

A görgetősáv változtatásakor a *scroll* esemény jön létre. Erre reagálunk az eseménykezelőben a háttérszín változtatásával. Használhatnánk a *change* eseményt is. Ebben az esetben az eseménykezelő csak akkor hívódna meg, amikor a felhasználó már felengedte a mozgató után az egérgombot.

A csúszka helyzetét úgy is változtathatjuk, ha melléje kattintunk a görgetősávon. A *value* ekkor a *LargeChange* paraméterben megadott értékkel változik. Ha a sáv valamelyik végén lévő fekete háromszögre kattintunk, akkor a csúszka a *SmallChange* paraméter értékével mozdul el. Beállítás nélkül mindkét paraméter értéke 1. A kattintások hatására a *change* esemény jön létre.

## Kódkomponensek használata

A kódkomponensek segítségével olyan objektumokat használhatunk fel, melyek előre elkészített függvényeket és eljárásokat (metódusokat) tartalmaznak. Ezeket a metódusokat a szkriptekből hívhatjuk.

A **8–70. példában** a Windows Dialog Helper (párbeszédsegítő) ActiveX objektumával jelenítjük meg a színválasztó párbeszédablakot. A rendszerleíró adatbázisban HtmlDlgSafeHelper néven kereshetjük meg ezt a komponenst.

A Dialog Helper a vezérlőelemekkel ellentétben nem látható a weblapon, azonban a *ChooseColorDlg* metódusa megjeleníti a színválasztó párbeszédablakot, és visszaadja a felhasználó által kiválasztott szín kódját. Bár egyéni színeket is definiálhatunk, ezek nem tárolódnak.

Bár a Dialog Helper nem jelenik meg a weblapon, a böngésző kihagy neki helyet. Ezt úgy akadályozhatjuk meg, hogy az objektum *width* és *height* tulajdonságának nulla értéket adunk.

Még egyszer felhívjuk a figyelmet arra, hogy példáink csak akkor működnek, ha a felhasznált ActiveX komponensek szerepelnek a rendszerleíró adatbázisban.

Ha egy weblap olyan ActiveX komponenst használ fel, amely nem található meg az adatbázisban, akkor a böngésző a dokumentumot szolgáltató szerverről próbálja meg lekérni. Az Interneten sok olyan helyet találunk, ahonnan változatos ActiveX komponenseket tölthetünk le. Az Olvasó figyelmébe ajánljuk például a következő címet:

<http://www.active-x.com>

Szükség esetén a *codebase* tulajdonsággal mi is megadhatjuk a komponens internetes címét (a URL-t) az OBJECT nyitó tagjában.

## Automatizmus komponensek

Mint említettük az ActiveX komponensek segítségével más Windows-alkalmazások objektumait is elérhetjük. Kezelésükhöz ismerni kell a Visual Basic for Applications (VBA) programozási nyelvet, melynek elsajátításához jó kiindulási alapot jelent a VBScript.

A Windows-alkalmazásokat kezelő komponensek úgynevezett folyamaton kívüli kiszolgálók, azaz újabb programot indítanak el. Ezek tehát nem a böngészőben jönnek létre, így nem DHTML-objektumként hivatkozunk rájuk, hanem objektumváltozót rendelünk hozzájuk. A későbbiekben hasonló módon kezeljük a fájlrendszer objektumait is.

Az objektumot a VBScript *CreateObject* függvényével hozzuk létre, melynek paramétereként meg kell adni az objektum osztályát. Az osztály megjelölésében szerepel az ActiveX kiszolgáló és annak típusa:

```
Set Objektumváltozó = CreateObject("Kiszolgáló.Típus")
```

A folyamaton kívüli kiszolgálók létrehozásánál az Internet Explorer biztonsági okokból rákérdez az engedélyezésre. Az engedélyezés az Eszközök/Internetbeállítások

menüpontban is beállítható. A Biztonság fülénél válasszuk az Egyéni szintet, és szükség esetén engedélyezzük az ActiveX vezérlők futtatását. Az alábbi példához megfelelnek az Internet Explorer eredeti beállításai.

Ha el akarjuk kerülni az állandó rákérdezést, akkor példáinkat HTML-alkalmazásként is futtathatjuk. Ehhez a .htm kiterjesztést .hta-ra kell átírni.

## Excel-táblázatok kezelése

Nincs módunk a VBA részletes ismertetésére, itt a **8–71. példán** keresztül mutatjuk be ezt a lehetőséget. A dokumentumban lévő szkript megnyit egy Excel-táblázatot, módosítja a benne tárolt adatokat és képleteket, majd bezárja az alkalmazást.

### FIGYELEM!

A példa kipróbálásához a CD-melléklet Fejezet08\Excel mappájában elhelyezkedő Táblázat.xls fájlt az Intézővel a c: meghajtó főkönyvtárába kell másolni (c:\). A futtatáshoz az Office 97-es vagy újabb változataira van szükség.

A táblázatban megadtuk két dolgozó fizetését, és képletekkel kiszámoltuk a jutalmakat, melyek értéke a fizetés 50 %-a. A példában a második dolgozó fizetését felemeljük 183 ezer forintra, a jutalom pedig mindkét dolgozónál a fizetés 80 %-a lesz.

A változtatásokat végző szkriptben először létrehozunk egy új alkalmazásobjektumot. A kiszolgáló esetünkben az Excel, típusa pedig alkalmazás (Application):

```
Set Alkalmazas = CreateObject("Excel.Application")
```

Megnyitással hozzáfűzünk egy munkafüzetet a *Workbooks* kollekcióhoz:

```
Set Munkafuzet = Alkalmazas.Workbooks.Open("c:\Táblázat.xls")
```

A munkafüzetnek (Excel-fájlnak) már léteznie kell a megadott helyen, különben hibaüzenettel leáll a szkript végrehajtása.

A munkafüzet első munkalapját hozzárendeljük egy objektumváltozóhoz:

```
Set Munkalap = Munkafuzet.Worksheets(1)
```

A munkalap 3. sorának 2. cellájába (B3-as cella) új értéket írunk be:

```
Munkalap.Cells(3, 2).value = 183000
```

A 2. és 3. sor 3. cellájába új képletet írunk. Figyeljünk arra, hogy a számoknál tizedespontot kell használnunk:

```
Munkalap.Cells(2, 3).formula = "=0.8*B2"
```

```
Munkalap.Cells(3, 3).formula = "=0.8*B3"
```

Bezárjuk a munkafüzetet, és kilépünk az alkalmazásból:

```
Munkafuzet.Close()
```

```
Alkalmazas.Quit()
```

Végül felszabadítjuk az objektum-hivatkozásokat:

```
Set Munkalap = Nothing
Set Munkafuzet = Nothing
Set Alkalmazas = Nothing
```

Reméljük, a figyelmes Olvasó észrevette, hogy nem mentettük el a változtatásokat!

A példa kipróbálásához másoljuk a CD-melléklet Példa08\Excel mappájában lévő Táblázat.xls fájlt a c: merevlemez főkönyvtárába. A szkript végrehajtásánál ne legyen nyitva a munkalap, és zárjuk be az Excelt is. Miután a parancsgombra kattintottunk, a szkript elvégzi a módosításokat a táblázatban. Mivel nem mentettük el a munkafüzetet, az Excel megjelenít egy párbeszédablakot, amellyel rákérdez a mentésre! Válasszuk az Igen gombot, majd az Excellel ellenőrizzük a változtatásokat.

A **8–72. példában** már a szkript menti el a táblázatot a változtatások után. Ehhez a munkafüzet *Save()* metódusát használja fel:

```
Munkafuzet.Save()
```

Természetesen az egyes cellák tartalmának a megváltoztatásához ciklusokat használhatunk, a képleteket sztringekből fűzhetjük össze. Megjegyezzük, hogy ilyen jellegű szkriptek esetén nagy figyelmet kell fordítani a hibakezelésre.

Az Excelhez hasonló módon érhetjük el a Word vagy az Access fájljait is.

## 8.6. Párbeszédablakok és menük

A továbbiakban röviden bemutatjuk a VBScript azon lehetőségeit, melyekkel hatékonyabbá tehetjük a felhasználóval való kommunikációt.

### Az MsgBox függvény

Az eddigiekben a *window*-objektum *alert* metódusát használtuk arra, hogy üzenetet közöljünk a felhasználóval. Bár az *alert* által kiírt szöveget a szkript állíthatja össze, így változók értékét is megjeleníthetjük, egyetlen OK gombja csak az üzenet tudomásul vételét teszi lehetővé. A VBScript *MsgBox* (message box, üzenetablak) függvénye már több választási lehetőséget kínál:

```
MsgBox(üzenet, gombok, cím)
```

Az *üzenet* és a *cím* tetszőleges karakterlánc lehet. A *gombok* paraméter értéke határozza meg a párbeszédablakban megjelenő parancsgombokat és ikont az alábbi táblázat alapján (zárójelben állnak a különböző Windows változatok feliratai):

Érték:	Konstans:	Gombok, ikonok:
0	vbOKOnly	OK
1	vbOKCancel	OK, Mégse
2	vbAbortRetryIgnore	Kilépés (Megszakítás, Leállítás), Ismét (Újra), Tovább (Kihagyás)
3	vbYesNoCancel	Igen, Nem, Mégse
4	vbYesNo	Igen, Nem

5	vbRetryCancel	Ismét (Újra), Mégse
16	vbCritical	kritikus üzenet (piros kör fehér x-szel)
32	vbQuestion	kérdőjel
48	vbExclamation	felkiáltójel
64	vbInformation	információ (i betű)

Paraméterként a gomboknál és ikonoknál választott értékek összegét kell beírni. Egy kérdőjel és az Igen, Nem gombok esetén például  $32 + 4 = 36$ -ot. Használhatjuk a megadott konstansokat is:

```
MsgBox("Folytassuk?", vbQuestion + vbYesNo, "Új játék")
```

A felhasználó választását a függvény visszatérési értéke adja meg:

Érték:	Konstans:	Gomb:
1	vbOK	OK
2	vbCancel	Mégse
3	vbAbort	Kilépés (Megszakítás, Leállítás)
4	vbRetry	Ismét (Újra)
5	vbIgnore	Tovább (Kihagyás)
6	vbYes	Igen
7	vbNo	Nem

A függvény használatát a **8–73. példa** mutatja be.

Megjegyezzük, hogy a függvényt a *Call* utasítással eljárásként (azaz kifejezés helyett önálló utasításként) is meghívhatjuk. Ekkor a visszatérési érték törlődik.

## Modális és nem modális párbeszédablakok

Az eddigiek során az *alert* metódus, illetve az *InputBox* és az *MsgBox* függvények segítségével jelenítettünk meg párbeszédablakokat. Ezek úgynevezett modális párbeszédablakok voltak, amíg nem zárultak be, nem lehetett másik objektumnak átadni a fókuszot. A nem modális párbeszédablakot a felhasználó nyitva hagyhatja, közben a weblap más elemeivel is végezhet tevékenységeket.

Modális párbeszédablak: bezárásáig megtartja a fókuszot, a dokumentum többi objektumát nem lehet kiválasztani.

Nem modális párbeszédablak: bezárása nélkül is elvesztheti a fókuszot. Lehetőségünk van visszatérni a böngésző ablakához, elérni a többi objektumot.

Modális párbeszédablakot a *window*-objektum *showModalDialog* metódusával, nem modális párbeszédablakot pedig a *showModelessDialog* metódussal hozhatunk létre. Mindkettő egy külön ablakot jelenít meg. Az ablakban lévő weblap általában valamilyen üzenetet vagy vezérlőelemeket tartalmaz. A metódusok hívásának formája:

```
showModalDialog(elérési_út, paraméter, megjelenítés)
showModelessDialog(elérési_út, paraméter, megjelenítés)
```

Az elérési út megadja a megnyitásra kerülő dokumentum helyét és a fájl nevét. A modális ablaknak paraméterként változót vagy tömböt tartalmazó változót, a nem modális paraméterablaknak pedig a *window*-objektumra való hivatkozást adjuk át. A megnyíló ablakban a paraméter értékét a *window*-objektum *dialogArguments* tulajdonságával érjük el.

A megjelenítés sztringje a párbeszédablak helyét és méretét szabályozza, a stílus-  
elemekhez hasonló szintaxissal. Legfontosabb összetevői:

<i>dialogWidth</i> :	a párbeszédablak szélessége
<i>dialogHeight</i> :	a párbeszédablak magassága
<i>dialogTop</i> :	távolság a képernyő tetejétől
<i>dialogLeft</i> :	távolság a képernyő bal szélétől
<i>center</i> :	elhelyezés a képernyő közepén ( <i>yes</i> vagy <i>no</i> )
<i>edge</i> :	az ablak síkja süllyesztett ( <i>sunken</i> ) vagy kiemelkedő ( <i>raised</i> )
<i>resizable</i> :	átméretezhető az ablak ( <i>yes</i> vagy <i>no</i> )
<i>scroll</i> :	szükség esetén görgetősáv megjelenítése ( <i>yes</i> vagy <i>no</i> )
<i>status</i> :	állapotsor megjelenítése ( <i>yes</i> vagy <i>no</i> )

A stíluselemek közül megadhatjuk a *font-size*, *font-weight* és *font-style* tulajdonságokat is. Az ablak méreténél kivételesen meg kell adni a pixelt (px).

A metódusokat függvényként hívjuk. A visszatérési értéket a *window returnValue* tulajdonsága hordozza. Ennek a modális dialógusablaknál tetszőleges értéket adhatunk, a nem modális ablaknál pedig a megnyíló új ablakra hivatkozik.

### Modális párbeszédablak készítése

A **8–74. példa** megjelenít egy párbeszédablakot, amelyben egy lista segítségével a felhasználó kiválaszthatja kedvenc macskafajtáját. A lista elemeit a *Fajta* tömbben tároljuk. A párbeszédablakot definiáló dokumentumot a Dialógus almappába tettük Cicalap.htm néven, így a metódus hívása:

```
Cica = window.showModalDialog("Dialógus\Cicalap.htm", Fajta, forma)
```

A *forma* sztringje a forráskódban látható. A választást a *Cica* nevű változó tartalmazza, amelynek az új ablakban a *returnValue* segítségével adunk értéket.

A Cicalap.htm fájlban a *window dialogArguments* tulajdonságával érjük el az átadott paramétert, azaz a fajtákat felsoroló tömböt:

```
Cicafajta = window.dialogArguments
```

A listát a szokásos módon készítjük el:

```
<SELECT id = "Lista" size = 1>  
<SCRIPT>  
  For I = 0 To UBound(Cicafajta)  
    document.write("<OPTION>" & Cicafajta(I) & "</OPTION>")  
  Next  
</SCRIPT>  
</SELECT>
```



Bár a párbeszédablak bezárásához felhasználhatnánk az *onchange* eseményt, a lehetőségek bemutatásához egy OK és egy Mégse gombot is elhelyezünk az ablakban. Az OK *onclick* eseménykezelőjében visszaadjuk az aktuális listaelemet, és bezárjuk az ablakot:

```
window.returnValue = Lista(Lista.selectedIndex).text  
window.close()
```

A Mégse választása esetén töröljük a visszatérési értéket, és bezárjuk az ablakot:

```
window.returnValue = ""  
window.close()
```

A párbeszédablakot megjelenítő dokumentumban a visszatérési érték alapján írjuk ki a választást:

```
If Cica <> "" Then  
    CicaKi.innerText = "A választott fajta: " & Cica  
Else  
    CicaKi.innerText = "Nem választott fajtát!"  
End If
```

Több érték visszaadásához általában tömböt tartalmazó változót használunk.

### Nem modális párbeszédablak készítése

A **8–75. példában** cicaneveket gyűjtünk. A neveket egy párbeszédablakba lehet beírni, amely a Hozzáad gombra való kattintással nyílik meg. A párbeszédablak Felvesz gombja az ablak bezárása nélkül bővíti a listát az új névvel, az OK gombra való kattintás esetén be is csukódik az ablak. Megjelenítünk egy Mégse gombot is, amely a lista bővítése nélkül zárja be az ablakot.

Emlékeztetünk arra, hogy a nem modális párbeszédablaknak a *window*-objektumra való hivatkozást adunk át. Ennek a segítségével az új dokumentumban hivatkozni tudunk az eredeti dokumentum globális változóira és alprogramjaira. A *showModelessDialog* visszatérési értéke viszont az új ablakra mutató objektum-hivatkozás, így az eredeti dokumentumból is tudunk hivatkozni az új dokumentum változóira és alprogramjaira.

*A weblap kódja*

Bár a nem modális párbeszédablakokból egyszerre többet megnyithatunk, ezt a lehetőséget most letiltjuk. Ehhez az új párbeszédablaknak fenntartott *Ablak*-objektum-hivatkozást először *Nothing*-ra állítjuk (kezdetben még nincs nyitva az új ablak):

```
Set Ablak = Nothing
```

A weblap Hozzáad parancsgombjának *onclick* eseménykezelőjében megvizsgáljuk az *Ablak* értékét, és csak *Nothing* esetén nyitjuk meg az új párbeszédablakot a *showModelessDialog* metódussal:

```
If Ablak Is Nothing Then  
    Set Ablak = window.showModelessDialog( _  
        "Dialógus\Cicanév.htm", window, forma)
```

Mint említettük, paraméterként weblapunk *window*-objektumát adjuk át. Az új ablak megnyitása után a modális párbeszédablakkal ellentétben folytatódik a szkript végrehajtása. Az *Ablak* változóval hivatkozhatunk az új ablak változóira és alprogramjaira.

Ha már nyitva van a párbeszédablak, akkor csak ráállítjuk a fókuszt a *Cicanev* szövegmezőre, és kijelöljük annak tartalmát:

```
Else
    Ablak.Cicanev.focus()
    Ablak.Cicanev.select()
End If
```

Figyeljük meg, hogy ezekkel az utasításokkal az eredeti dokumentum szkriptjében hivatkozunk az új ablak változóira (objektumaira)!

A szövegmezőbe írt cicanévet az új dokumentum szkriptjében átmásoljuk az eredeti dokumentum *Nev* változójába, majd a *Beir* eljárással hozzáfűzzük a nevek listáját tartalmazó bekezdéshez. Ha a felhasználó bejelölte a kedvenc név jelölőnégyzetet, akkor a szöveget félkövér betűkkel írjuk ki:

```
Sub Beir()
    If Nev <> "" Then
        If Kedvenc.checked Then
            Nev = "<B>" & Nev & "</B>"
        End If
        Lista.innerHTML = Lista.innerHTML & Nev & "; "
    End If
End Sub
```

A *Beir* eljárást az új ablakból hívjuk!

*A párbeszédablak kódja*

A párbeszédablakot a *Cicanév.htm* fájl definiálja, melyet a Dialógus mappába tesszünk. A *dialogArguments* tulajdonság segítségével hozzárendeljük az eredeti ablakot a *WebLap* objektumváltozóhoz:

```
Set WebLap = window.dialogArguments
```

Így a *WebLap* minősítés segítségével az eredeti weblap minden globális változójához és alprogramjához hozzáférhetünk.

A párbeszédablak három parancsgombot tartalmaz. A *Felvesz* gombra való kattintáskor a szövegmezőbe beírt cicanévet átírjuk a *Nev* változóba, és meghívjuk az eredeti dokumentum *Beir* szubrutinját:

```
WebLap.Nev = Cicanev.value
WebLap.Beir()
```

Az OK gombra való kattintáskor ugyanezeket a feladatokat a *Felvesz* gomb előbbi eseménykezelőjének meghívásával végezzük el, majd bezárjuk az ablakot:

```
Felvesz_onclick
window.close()
```

A Mégse gombra való kattintáskor töröljük a *Nev* változó értékét, és bezárjuk az ablakot:

```
WebLap.Nev = ""  
window.close()
```

Gondoskodnunk kell még az *Ablak* változó értékének törléséről, ha a felhasználó becsukja a párbeszédablakot. Ehhez a *window*-objektum *onunload* eseményét használjuk fel, ami az ablak bezárásakor következik be:

```
Sub window_onunload  
    Set WebLap.Ablak = Nothing  
End Sub
```

A példa kipróbálásánál figyeljük meg, hogy a jelölőnégyzet állapotát akkor is változtathatjuk, ha nyitva van a párbeszédablak! Ezt egy modális párbeszédablak alkalmazása esetén nem tehetnénk meg.

Ha elhagyjuk a kódból a *Nothing* vizsgálatát, akkor a Hozzáad gombra történő ismételt kattintással több párbeszédablak is megnyitható, melyek egymástól függetlenül működnek.

### Párbeszédablak folyamatjelzővel

A nem modális párbeszédablakot felhasználhatjuk folyamatjelző készítésére is. Ezzel elkerülhetjük olyan ActiveX objektumok alkalmazását, melyek esetleg nincsenek regisztrálva a felhasználó számítógépén.

Folyamatjelzőként egy meghatározott magasságú és pozíciójú DIV-objektumot alkalmazunk, melynek kódját a CD-melléklet Fejezet08\Dialogus mappájában található Folyamat.htm fájl tartalmazza. Az objektum hátterét kékre színezzük, kezdeti szélességének pedig 0-t adunk meg:

```
<DIV id = "Jelzo" style = "width: 0; height: 20;  
                           background-color: blue;  
                           position: absolute; top: 20; left: 10">  
</DIV>
```

A folyamatjelző ablakot a **8–76. példa** szkriptjében nyitjuk meg:

```
Set Ablak = window.showModelessDialog( _  
    "Dialogus\Folyamat.htm", _  
    window, _  
    "dialogWidth: 230px; dialogHeight: 60px; status: no")
```

A példában a folyamatjelzőt egy ciklus futásának jelzésére használjuk. Az egyszerűség kedvéért a ciklus most nem végez semmilyen tevékenységet, csak a folyamatjelzőt módosítja. A hivatkozás egyszerűsítéséhez (és a futás gyorsításához) a folyamatjelző ablak DIV-objektumának *style* tulajdonságát hozzárendeljük egy objektumváltozóhoz:

```
Set Teglalap = Ablak.Jelzo.style
```

A ciklusban a *Teglalap width* elemét módosítjuk a ciklusváltozó értékének megfelelően. Ha a maximális szélesség 200 pixel, a ciklus pedig 1-től 10000-ig fut, akkor:

```
For I = 1 to 10000
    Teglalap.width = I / 10000 * 200
Next
```

A ciklus befejezése után bezárjuk az ablakot, és töröljük az objektumot:

```
Ablak.close()
Set Ablak = Nothing
```

A Folyamat.htm fájlt hasonló módon más programokban is felhasználhatjuk.

## Menü készítése

A grafikus felületet használó programozási nyelvek segítségével az ablakokat a szokásos kellékekkel láthatjuk el, így módot nyújtanak menüsor készítésére is. Mivel a VBScript egy böngésző ablakában fut, közvetlenül nincs lehetőségünk a menüsor módosítására. Ennek a hiányosságnak a kiküszöbölésére számos trükk létezik. Itt egy viszonylag egyszerű, de hasznos módszert mutatunk be.

A menüponthoz tartozó listát a *clip* (kivágás) stíluselemmel rejtjük el. A *clip* meghatározza, hogy mekkora rész maradjon látható az objektumot magában foglaló képzeletbeli téglalap (*rect*, rectangle) felső, jobb oldali, alsó és bal oldali szélétől számítva:

```
style = "clip: rect(felül jobbra alul balra)"
```

Ha valamelyik oldalon nem akarunk levágást alkalmazni, akkor azt az értéket állítsuk *auto*-ra. A *clip* csak abszolút pozicionálás esetén alkalmazható.

A **8–77. példában** az egyes menüpontokat DIV-objektumok tartalmazzák. A DIV-en belül SPAN-objektumok sorolják fel a menüpontokhoz tartozó parancsokat.

Mindegyik DIV-objektumnál megadtuk a pozíciót, a háttérszínt, a szélességet és az alsó kivágás értékét úgy, hogy csak egyetlen sor legyen látható. Az *onmouseover* eseménynél az alsó kivágást is automatikusra állítjuk, így megjelenik a teljes lista. Mivel a böngésző a szöveg fölött megváltoztja az egérkurzor alakját, ezért azt visszaírjuk nyíllra (*default*):

```
Obj.style.clip = "rect(auto auto auto auto)"
Obj.style.cursor = "default"
```

Emlékezzünk vissza arra, hogy a *Me* kulcsszó azt az objektumot jelenti, amelyik meghívta az eseménykezelő eljárást. A hivatkozáshoz ezt adjuk át az eljárásnak az *Obj* paraméter segítségével.

Az *onmouseout* eseménynél az alsó levágást ismét 16 pixelre állítjuk, a választott betűméretnél ez felel meg egy sornak. Az egérmutató alakját pedig rábízuk a böngészőre (*auto*):

```
Obj.style.clip = "rect(auto auto 16 auto)"
Obj.style.cursor = "auto"
```

Az egyes SPAN-objektumok esetén az *onmouseover* eseményre megváltoztatjuk, az *onmouseout*-ra pedig visszaírjuk a háttér és a betűk színét. Figyeljük meg, hogy a

listák első eleméhez, illetve a Fájl menüben az elválasztóvonalhoz nem rendeltünk eseménykezelést, így azokat nem lehet kiválasztani!

#### *A menüparancsok hozzárendelése*

A menüparancsok az *onclick* eseményre lépnek működésbe. Ezt a fenti példában csak a Beállítások menüben a Háttérszín és az „Automatikus mentés”, illetve a Fájl/Kilépés parancsokhoz rendeltük hozzá. A háttérszínt a már többször alkalmazott módon, véletlenszerűen változtatjuk. Az „Automatikus mentés”-nél csak jelezzük a beállítást, illetve megszüntetést. Ehhez bevezetjük az *Automatikus* logikai változót. Ha ennek értéke hamis, a menüpontot kiegészítjük egy pipával, ha igaz, akkor elhagyjuk a jelölést. A logikai változó értékét is módosítjuk a választásnak megfelelően:

```
If Automatikus Then
  Obj.innerHTML = "Automatikus mentés"
  Automatikus = False
Else
  Obj.innerHTML = "Automatikus mentés &radic;"
  Automatikus = True
End If
```

Egy igazi menü esetén természetesen el kell készíteni az összes menüpont *onclick* eseménykezelőjét. Magukat a DIV- és a SPAN-objektumokat ciklussal hozhatjuk létre, egyben elvégezhetjük az eseménykezelő eljárások hozzárendelését. Ezzel nagymértékben egyszerűsödik a forráskód.

Mint említettük, sokféleképpen készíthetünk menüt a weblapra. Megjegyezzük, hogy a *PopupMenu* ActiveX vezérlőelem is ezt a célt szolgálja.

### **A felhasználó tájékoztatása**

#### *A title tulajdonság*

A legtöbb DHTML-objektum rendelkezik a *title* tulajdonsággal, melynek szövege egy kis sárga téglalapban (tooltip) megjelenik a képernyőn, ha az egeret az objektum fölé visszük.

A **8–78. példában** a *title* segítségével írjuk ki a képeken látható virágok nevét. Ha a tulajdonság értékét egy szkriptben adjuk meg, akkor némi tördelési lehetőség is a rendelkezésünkre áll. Ezt a weblapra helyezett parancsgombnál mutatjuk be. A *title* értékek megváltoztatásához felhasználtuk az *images* kollekciót, amely a dokumentumba illesztett képeket tartalmazza.

Megjegyezzük, hogy a *behavior* (viselkedés) stíluselem segítségével lehetőségünk van a teljes HTML-eszközkészlet felhasználására a *title* megadásánál. Erről részletebben a Microsoft Development Network weblapján tájékozódhatunk.

#### *Felbukkanó ablakok megjelenítése*

Az egyszerű tooltipnél sokkal látványosabb magyarázatra ad lehetőséget a felbukkanó (popup) ablak megjelenítése. A popup-objektumot a *window createPopup* módszerével lehet létrehozni, és a *Set* utasítással rendelhetjük egy objektumváltozóhoz (**8–79. példa**):

```
Set Herics = window.createPopup()
```

Az objektum a weblapokhoz hasonló szerkezetű, a HTML-kódját *document*-objektumának *body* elemével hozhatjuk létre:

```
Set HericsBody = Herics.document.body
HericsBody.text = "white"
HericsBody.bgColor = "blue"
HericsBody.innerHTML = "<B>Tavaszi héric</B>"
```

A példa kódjában a stíluselemek segítségével még több formátumot is megadtunk.

A megjelenítést a kép *onmouseover* eseménykezelőjében a popupobjektum *show* metódusával végezzük el. A *show* szintaxisa:

```
ObjektumAzonosító.show(x, y, szélesség, magasság, viszonyítás)
```

Az *x* és *y* paraméterek pixelben mérve megadják az ablak helyzetét a *viszonyítás* objektumhoz képest. A popup ablak szélességét és magasságát szintén pixelben mérjük. Ha nem adjuk meg a viszonyítási objektumot, akkor a böngésző a képernyő koordináta-rendszerét használja.

A tavaszi héric esetén a képhez viszonyítjuk a pozíciót:

```
Call Herics.show(50, 50, 130, 20, HericsKep)
```

Az ablakot a *hide* metódussal lehet elrejteni:

```
Herics.hide()
```

A példában ezt az *onmouseout* eseménykezelőben tettük meg, de megjegyezzük, hogy egy popup ablak akkor is eltűnik, ha bárhová kattintunk az egérrel. Így a *hide* metódust nem kell közvetlenül meghívni.

Ha a popup ablak hosszabb szöveget tartalmaz, akkor célszerű egy külön DIV-objektumban összeállítani a kódot, majd a tartalmát átmásolni az ablakba. Ennek módját a **8–80. példa** mutatja be. A DIV-objektumot a *visibility* stíluselem segítségével lehet elrejteni, hogy ne jelenjen meg a weblapon. A példában a *visibility* helyett a *display* stíluselemet használtuk, mert így a böngésző nem hagy ki helyet az objektumnak. A *filter* stíluselemre nem térünk ki részletesen, ez hozza létre a háttér áttűnését sötétkékből feketébe.

A példában a popup ablak megjelenítését végző eljárásnak átadtuk a képjelző objektum indexét. Ennek segítségével a megfelelő DIV tartalmát átmásoljuk az ablakba. A DIV-objektumokat ugyanazzal a *Kod* azonosítóval láttuk el, így kollektívként kezelhetjük őket:

```
ViragBody.innerHTML = Kod(Index).innerHTML
```

Az index segítségével adjuk meg a képjelző objektumot is az ablak pozicionálásához:

```
Call Virag.show(80, 130, 160, 190, document.images(Index))
```

Megjegyezzük, hogy a popup-objektumokat globálisan kell létrehozni. Mivel nem kaphatják meg a fókuszt, a nem modális ablakokhoz hasonlóan viselkednek. Ha nem tüntetjük el őket, akkor is folytathatjuk a munkát a weblapon, például adatot írhatunk egy szövegmezőbe.

## 9. ÖSSZETETT ADATTÍPUSOK

A 6. fejezet elején két csoportba soroltuk a változókat. Az egyszerű típusok csak egyetlen adatot (számot, karakterláncot, logikai értéket) tárolnak, az összetett típusok pedig többet. Az összetett típusok közül eddig a tömbökkel ismerkedtünk meg. A kollekciókat speciális tömböknek tekinthetjük. A továbbiakban bonyolultabb adattípusokat tárgyalunk. Bemutatjuk ezek megvalósítását tömbök és objektumok segítségével.

### 9.1. Tömbök és listák

#### Adatszerkezetek

Az informatikában a valós világ elemeit a program tervezése során modellezzük, az általuk hordozott információ tárolásához adatmodelleket alakítunk ki. Az adatmodell a számunkra lényeges információkat kiemeli, a többit elhagyja. Ugyanazt a rendszert többféleképpen is lehet modellezni. Egy osztály tanulóinál vizsgálhatjuk a tanulmányi eredményeket, vagy azt, hogy melyik csoportban tanulnak egy adott idegen nyelvet.

Az adatmodell nem csak adatokat tartalmaz, elemei között kapcsolatok állnak fenn. A tanulmányi eredmények alapján sorrendet lehet megállapítani a tanulók között. A nyelvtanulás szempontjából pedig csoportosítani tudjuk őket. Az adatokat és a közöttük fennálló kapcsolatokat adatszerkezetnek nevezzük. A tanulmányi eredmények alapján sorozatot készíthetünk, a nyelvi csoportok pedig halmazokat alkotnak.

Az elméleti adatszerkezeteket a programozás során konkrét eszközökkel kell megvalósítani. Az adatok tárolását a memória és a háttértár végzi. A kapcsolatokat a használt programnyelv segítségével alakítjuk ki.

Két város távolságát például egy fizikai mennyiséggel, a hosszúsággal írjuk le. Ezt az adatmodellt programunkban egy egyszerű, numerikus változó valósítja meg. Magyarország különböző városainak távolságát az autóatlaszokban látható táblázatos formában adjuk meg. Ezt az adatszerkezetet kétdimenziós tömbben tárolhatjuk. De ha két város között a legrövidebb útra vagyunk kíváncsiak, akkor már egy bonyolultabb modellt, a térképet kell felhasználnunk. Nem csak a távolságok számítanak, hanem az is, hogy egy városból közvetlenül mely városokba vezet út. A térképnek megfelelő adatmodell egy úgynevezett gráf, melyet hálós adatszerkezettel írhatunk le.

A tömbök nagyon sokrétű adattárolási lehetőséget biztosítanak, sokféle adatszerkezet valósítható meg a segítségükkel. Ha például a városok távolságát megadó táblázatban csak azokat a cellákat töltjük ki, melyek két szomszédos városhoz tartoznak, akkor ebből a kétdimenziós tömbből a megfelelő algoritmus segítségével meg tudjuk határozni bármely két város között a legrövidebb utat. Ez a tömb tehát egy gráfot, egy hálós adatszerkezetet realizál.

Az adatszerkezetek részletes ismertetése jóval túlmutat könyvünk keretein. Az érdeklődő Olvasónak a témával kapcsolatos szakkönyvek tanulmányozását ajánljuk. A továbbiakban néhány speciális eset tárgyalásával bemutatjuk az adatszerkezetek használatát. Felhívjuk a figyelmet arra, hogy nagy mennyiségű, összetett adatok, adatbázi-

sok kezelésére nem célszerű egyedi programokat készíteni. Ehhez már egy adatbázis-kezelő rendszer alkalmazására van szükség.

## A tömbök elemei

Eddig általában olyan tömböket használtunk, melyek elemei egyszerű típusú adatok voltak. Tömbök azonban tartalmazhatnak újabb tömböket, illetve objektumokat is. Ez utóbbi lehetőséggel már találkoztunk a kollekciók bemutatásánál.

Ha a tömb egy eleme szintén tömb, akkor az eredeti tömbnévnek és indexének megadásával hivatkozunk rá. Ezután kell kiírni az elemtömb indexét. Így az *Adat* nevű tömb 3-as indexű elemeként szereplő tömb 4-es indexű eleme:

```
Adat (3) (4)
```

Bár a szerkezet megfelel egy kétdimenziós tömbnek, a hivatkozás szintaxisa eltér attól. Ez azonban nem csak formai változást jelent. Ha például fel akarjuk cserélni az *Adat* tömb 4-es és 5-ös indexű elemét alkotó tömböket, akkor ezt a következő utasításokkal tehetjük meg:

```
Temp = Adat (4)
Adat (4) = Adat (5)
Adat (5) = Temp
```

Vegyük észre, hogy kétdimenziós tömb esetén ezt csak ciklussal végezhetjük volna el, nem szólva arról a jóval bonyolultabb esetről, amikor a két felcserélt tömb nem is egyforma elem- vagy dimenziószámú.

Tömböt nem csak tömbelemek, hanem egyszerű változók is tartalmazhatnak. A már ismert *Array* függvény éppen ezt az esetet valósítja meg, egy változónak tömböt ad értékül:

```
Változónév = Array(tömbelemek felsorolása)
```

A **9–1. példában** a lottósorsolások heti húzásait tartalmazó tömböket egy dinamikus tömbben tároljuk. A feladatok között bemutatott bővítésnél elegendő ezt az egydimenziós tömböt újradimenzionálni. Nem jelent gondot az a megkötés, hogy a dinamikus tömböknél csak az utolsó dimenzió mérete változtatható.

Megjegyezzük, hogy egy tömb elemei tetszőleges típusú változók lehetnek. Egyik eleme tartalmazhat egy numerikus értéket, másik eleme kétdimenziós tömböt, harmadik eleme egy objektumot, negyedik eleme hatdimenziós tömböt és így tovább. A tömbelemként szereplő tömb elemeinél ugyanilyen lehetőségekkel rendelkezünk.

## Dinamikus tömbelemek

Ha egy tömb elemeiként dinamikus tömböket akarunk alkalmazni, akkor a szintaxis kötöttsége miatt segédváltozót használunk. A módszert a **9–2. példában** mutatjuk be, amelyben diákok jegyeit tároljuk. A jegyeket dinamikus tömbökbe írjuk, új jegy esetén bővítjük a tömböt. A dinamikus tömböket a diákok sorszáma szerint egy újabb dinamikus tömbben tároljuk. A weblapon megjelenítünk két szövegmezőt a diák sorszámanak és jegyének beírására, ami a Felvesz gombra való kattintás után történik meg.



A globális szkriptben deklaráljuk a *Diak* és a *Temp* dinamikus tömböt. Példánkban öt diák jegyeit tároljuk, a *Temp*-et pedig újradimenzionáljuk egyetlen elemmel (a maximális index 0):

```
Dim Diak(), Temp(), I
ReDim Diak(4)
ReDim Temp(0)
```

Ciklussal beírjuk a *Temp* tömböt a *Diak* tömb minden elemébe. Ezzel a közvetett módszerrel érjük el, hogy az elemek dinamikus tömbök legyenek:

```
For I = 1 To 4
    Diak(I) = Temp
Next
```

A dinamikus tömbök 0 indexű elemét nem használjuk jegy tárolására, ide írhatjuk például a diák nevét. Így nem kell állandóan megvizsgálni, hogy van-e már eleme a tömbnek.

Egy jegy beírása esetén némi hibaellenőrzést végzünk, ezt a példa forráskódja tartalmazza. Ha megfelelő adatok kerültek a szövegmezőkbe, akkor bővítenünk kell a megadott diák jegyeit tartalmazó dinamikus tömböt, amit szintén egy segédváltozó útján érünk el:

```
Temp = Diak(Sorszam)
MaxIndex = UBound(Temp) + 1
ReDim Preserve Temp(MaxIndex)
```

Ezt követően tároljuk az új jegyet, majd a segédváltozót visszaírjuk az eredeti tömbbe:

```
Temp(MaxIndex) = Jegy
Diak(Sorszam) = Temp
```

A példa forráskódja tartalmazza még a listázást végző eljárást, ami a szokásos módon jeleníti meg az adatokat.

## Listák

A listák olyan adatszerkezetet valósítanak meg, amelyben minden egyes elem tartalmaz egy mutatót a logikai szempontból rákövetkező elemre. A listaelem tehát két részből áll, az adatból és a mutatóból. Ennek megfelelően a VBScriptben listát olyan tömbből készíthetünk, melynek elemei kételemű tömbök. A *Lista* tömb I-edik eleme:

```
Lista(I)(0) = adat : Lista(I)(1) = a_következő_elem_indexe
```

Áttekinthetőbb kódot kapunk, ha a második index helyett konstansokat alkalmazunk:

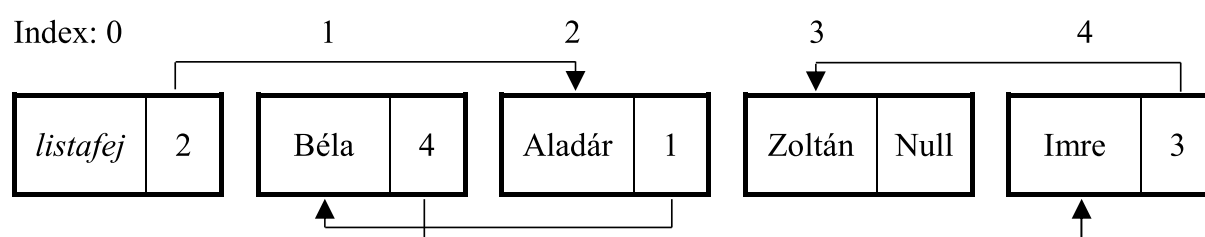
```
Const Adat = 0, Mutato = 1
...
Lista(I)(Adat) = adat : Lista(I)(Mutato) = a_következő_elem_indexe
```

Az *adat* az eddigieknek megfelelően bonyolultabb típussal is rendelkezhet. A lista adatszerkezetet ne tévesszük össze a SELECT/OPTION-objektumokkal!

A tömb 0 indexű elemét arra használjuk, hogy a lista első elemére mutasson (így az adatrészét üresen hagyjuk). A lista utolsó eleménél viszont a következő elem indexének helyére *Null*-t írunk:

*Index:      Adat:      Mutató (a következő elem indexe):*

0	-	2
1	Béla	4
2	Aladár	1
3	Zoltán	Null
4	Imre	3



9–1. ábra. A tömb elemei a lista adataival és a mutatókkal

A lista kezdetét a 0 indexű elem mutatója jelzi, így a lista első eleme Aladár. Aladár mutatója Bélára hivatkozik, Béla Imrere, Imre Zoltánra. Zoltán mutatójának értéke *Null*, így befejeződött a lista. A 0 indexű elemet, amely nem tartalmaz adatot, szokás listafejnek is nevezni.

A **9–3. példa** bemutatja az elemek megjelenítésének algoritmusát, egyelőre a globális szkriptben megadott lista esetén. Az első elem indexét a listafej mutatója jelzi:

```
Index = Nev(0) (Mutato)
```

Egy ciklus segítségével hozzáfűzzük a kiírásra kerülő sztringhez a nevet, majd átírjuk az index értékét az aktuális listaelem mutatójának értékére. A ciklust addig futtatjuk, amíg a mutató értéke *Null* nem lesz:

```
Do Until IsNull(Index)
    Lista = Lista & Nev(Index) (Adat) & "<BR>"
    Index = Nev(Index) (Mutato)
Loop
```

## Keresés a listában

A keresés algoritmusát rendezett lista esetén a **9–4. példában** mutatjuk be. A későbbiekben szükségünk lesz a keresett elemet megelőző elem indexére, ezért ezt is feljegyezzük.

Kezdetben az előző elem indexe 0 (listafej), az első elem indexét pedig a listafej mutatója adja meg:

```
Elozo = 0
Index = Nev(0) (Mutato)
```

Bevezetünk egy logikai változót, amely jelzi, ha befejezhetjük az összehasonlításokat, mert megtaláltuk a keresett elemet, vagy már nála nagyobb elemek következnek (rendezett a lista):

```
Folytat = True
```

A keresést addig végezzük, amíg el nem érünk a lista végére, illetve le nem állítjuk a folytatást. A lista végét a mutató *Null* értéke jelzi:

```
Do While Not IsNull(Index) And Folytat
```

Ha az aktuális elem kisebb, mint a keresett érték, akkor átállítjuk rá az előző elem indexét, a következő elemet pedig a mutató jelöli ki:

```
If Nev(Index) (Adat) < KeresettNev Then
    Elozo = Index
    Index = Nev(Index) (Mutato)
```

Ha már túljutottunk a sorban a keresett elem helyén, akkor leállítjuk a keresést:

```
Else
    Folytat = False
End If
Loop
```

A keresett elemet akkor találtuk meg, ha a ciklusból való kilépés után az *Index* értéke nem *Null*, és

```
Nev(Index) (Adat) = KeresettNev
```

A 9–4. példában az *Elozo* és az *Index* változók segítségével azt is meghatároztuk, hogy a lista mely két eleme közé kellene beilleszteni az ott nem szereplő nevet. Ha a magyar ábécé szerinti sorrendet akarjuk kialakítani, akkor az elemek összehasonlítását az *StrComp* függvénnyel kell elvégeznünk.

Rendezetlen listában történő keresés esetén az összes elemet végig kell nézni, amíg meg nem találjuk a keresett értéket.

## A lista rendezett feltöltése

Listával könnyen megvalósíthatjuk egy tömb elemeinek rendezett beolvasását, az elemek állandó áthelyezése nélkül. A módszert a **9–5. példa** mutatja be. A tároláshoz dinamikus tömböt használunk. Kezdetben a listafej mutatóját *Null*-ra állítjuk (még nincs eleme a listának):

```
Dim Nev
ReDim Nev(0)
Nev(0) = Array("", Null)
```

Új elem beolvasásánál egy sorral bővítjük a tömböt, és a végére beírjuk az új nevet. A mutatónak egyelőre *Empty* értéket adunk (bármilyen más is írhattunk volna):

```
UjNev = NevBe.value
MaxIndex = UBound(Nev) + 1
ReDim Preserve Nev(MaxIndex)
Nev(MaxIndex) = Array(UjNev, Empty)
```

Az előző példában bemutatott algoritmussal megkeressük az új név helyét. Az előző elem mutatóját az új elemre, azaz *MaxIndex*-re állítjuk (akkor is, ha az előző elem a listafej), az új elem mutatóját pedig a következő elemre, amit a keresés által megadott *Index* változó jelez:

```
Nev(Elozo) (Mutato) = MaxIndex  
Nev(MaxIndex) (Mutato) = Index
```

Ezután már csak a listázás van hátra, amit az ismert algoritmussal végzünk el.

### Törlés a listából

Egy elem törlésének az algoritmusát a **9–6. példa** mutatja be. Először a szokásos módon megkeressük a beírt nevet a listában. Ha nem találtuk meg, akkor hibaüzenetet adunk, és kilépünk a törlést végző eljárásból.

Ha megtaláltuk, akkor könnyen elvégezhetjük a törlést. Az előző elem mutatóját rá kell állítani a törlésre kerülő elem által mutatott elemre. A törlésre kerülő elemet keresési eljárásunk *Index* változója jelzi:

```
Nev(Elozo) (Mutato) = Nev(Index) (Mutato)
```

A **9–7. példában** az előző két példát összevonva bővíthetjük a listát, és törölhetünk is belőle elemeket. Ehhez csak a törlést végző szubrutint kell átmásolni a feltöltést végző dokumentumba.

A bemutatott törlési eljárással a listából kikerül az elem, magában a tömbben azonban benne marad. Sajnos a dinamikus tömbnek csak a végén elhelyezkedő elemeit tudjuk valójában törölni. Így átrendeázés nélkül a tömb mérete nem módosítható.

A teljes lista törlése már fizikailag is elvégezhető. A tömböt újradimenzionáljuk 0 maximális indexszel, és a megmaradó listafej mutatóját *Null*-ra állítjuk:

```
Redim Nev(0)  
Nev(0) = Array("", Null)
```

A listákkal nagyon változatos adatmodelleket lehet megvalósítani. Itt csak a leggyorsabb formájukat tárgyaltuk. Az előző elemre mutató index használatával készíthető olyan lista, amelyben visszafelé is tudunk lépkedni. A cirkuláris listában az utolsó elem az elsőre mutat, így körbejárható. Multilisták esetén pedig bármely listaelem újabb lista kiindulópontja lehet.

## 9.2. Szótárak és halmazok

### Asszociatív tömbök

Az eddig megismert tömbök elemeire indexük segítségével hivatkozunk. Indexként nem negatív egész számokat alkalmazunk. Az asszociatív tömbök elemeit név–adat párok alkotják, az adatokra a nevekkel hivatkozunk. Az ilyen jellegű hivatkozás esetén asszociatív tárolási módról beszélünk. Az asszociatív adattárolás fontos szerepet játszik az adatbázisok kezelésében, ahol a nevet gyakran kulcsnak hívják.

A kulcs valamilyen numerikus érték vagy sztring, de egy tömbön belül nem lehet két különböző elemnél azonos az értéke. A tömb adatait tetszőleges típusú változók alkotják.

Az asszociatív tömböt a *Dictionary* (szótár) objektum valósítja meg. Létrehozása a már ismert *CreateObject* metódushívással történik úgy, hogy egyben hozzá is rendeljük egy objektumváltozóhoz. A kiszolgáló itt egy úgynevezett Script Runtime osztálykönyvtár (*Scripting*), a típus pedig *Dictionary*:

```
Set TömbNév = CreateObject("Scripting.Dictionary")
```

A cicák adatait tartalmazó szótár létrehozása például az alábbi utasítással történhet:

```
Set Cica = CreateObject("Scripting.Dictionary")
```

Elemeket a szótárobjektum *Add* metódusával adhatunk az asszociatív tömbhöz. Minden egyes elem esetén megadjuk a kulcs és az adat értékét:

```
Call TömbNév(kulcs, adat)
```

A 9–8. példában kulcsként cicáink nevét használjuk, adatként pedig a fajtájukat tároljuk:

```
Call Cica.Add("Kormi", "egyiptomi")
Call Cica.Add("Cirmi", "angóra")
```

A szótárban az elemek a bevitel sorrendjében helyezkednek el. A bővítéshez bármikor meghívhatjuk az *Add* metódust.

### A szótár elemeinek elérése

A szótárobjektum elemeit az *Item* (elem) tulajdonsággal érhetjük el, melynek paramétere a keresett adat kulcsa:

```
TömbNév.Item(kulcs)
```

A tulajdonság értéke a kulcshoz tartozó adat. A *Cica.Item("Kormi")* értéke például "egyiptomi".

A 9–8. példában az *Item* segítségével kilistázzuk a cicák nevét és fajtáját. A listázáshoz igen kényelmesen használható a *For Each* utasítás, amely sorra átadja a halmaz kulcsait az utána álló változónak:

```
For Each Nev In Cica
    Lista = Lista & Nev & " fájtája: " & Cica.Item(Nev) & "<BR>"
Next
```

Az *Item* által kijelölt elemet meg is változtathatjuk:

```
Cica.Item("Kormi") = "abesszin"
```

Ha a kulcs nem található meg a szótárban, akkor az értékadó utasítás bal oldalán szereplő kulcs a jobb oldalon lévő értékkel bekerül a tömb elemei közé! Ez tehát az *Add* metódussal egyenértékű eredményre vezet. Az *Add* metódussal azonban nem módosíthatunk egy már meglévő elemet, mert hibát jelez, ha létezik a kulcs a szótárban!

A *Key* tulajdonsággal egy kulcsot változtathatunk meg:

```
TömbNév.Key(régi_kulcs) = új_kulcs
```

Ha a régi kulcs nem, vagy az új kulcs már szerepel a szótárban, akkor hibaüzenetet kapunk.

A kulcsok vizsgálatához a *CompareMode* tulajdonsággal határozhatjuk meg az összehasonlítás módját:

```
TömbNév.CompareMode = mód
```

A *mód* megfelel az *StrComp* függvénynél ismertetett paraméternek. Ha értéke 0, akkor megkülönbözteti, 1 esetén pedig nem különbözteti meg a kis- és a nagybetűket egymástól.

Az *Item* használata elhagyható, így a szótár elemeit a kulcsokkal indexelhetjük, például:

```
Cica("Cirmi") = "sziámi"
```

Ez az utasítás megváltoztatja a „Cirmi” kulcshoz tartozó értéket, ha már létezik, és felveszi a „Cirmi”–„sziámi” párt, ha még nem létezik. A továbbiakban alkalmazni fogjuk ezt az egyszerűsítési lehetőséget.

Megjegyezzük, hogy ha az *Item* egy kifejezésben szerepel, de paramétere nincs benne a szótárban, akkor létrejön egy új elem az adott kulccsal és üres (*Empty*) értékkel. A

```
Temp = Cica.Item("Cirmi")
```

utasítás például felveszi a szótárba a „Cirmi” kulcsot, de a hozzá tartozó érték üres lesz. Ugyanerre az eredményre vezet a

```
Temp = Cica("Cirmi")
```

utasítás is.

## Keresés a szótárban

Mint láttuk, az *Item* tulajdonság használata esetén egy nem létező kulcsra való hivatkozásnál sem kapunk hibaüzenetet. Ha nem akarjuk, hogy egy rosszul begépett adat miatt bővüljön a szótár, akkor az *Exists* (létezik-e) metódussal ellenőriznünk kell a paraméterként megadott kulcs létezését:

```
TömbNév.Exists(kulcs)
```

A függvény visszatérési értéke *True* vagy *False*.

A **9–9. példa** lehetővé teszi a cicák fajtájának megváltoztatását. Először ellenőrizzük a beírt név létezését:

```
If Not Cica.Exists(NevBe.value) Then  
    window.alert("Nincs ilyen nevű cica.")
```

Ha találtunk ilyen nevet, akkor megváltoztatjuk a fajtáját:

```
Else  
    Cica(NevBe.value) = FajtaBe.value  
End If
```

A CD-mellékleten található példa még a listát is frissíti.

## Törlés a szótárból

A szótár egy elemét (a kulcs–adat párt) a *Remove* (töröl) metódussal törölhetjük, melynek paramétereként a kulcs értékét kell megadni:

```
TömbNév.Remove(kulcs)
```

A **9–10. példában** lehetővé tesszük az elemek törlését. A *Remove* metódus hibaüzenetet ad, ha nem létező kulcsra hivatkozunk, ezért a törlés előtt az *Exists* metódussal megvizsgáljuk, hogy szerepel-e a megadott kulcs a szótárban:

```
If Cica.Exists(NevBe.value) Then  
    Cica.Remove(NevBe.value)  
Else  
    window.alert("Nincs ilyen nevű cica.")  
End If
```

A példát kiegészítettük egy Felvesz/Módosít gombbal is, amelynek eseménykezelő eljárása egyetlen lényeges utasítást tartalmaz:

```
Cica(NevBe.value) = FajtaBe.value
```

Ha már létezik a megadott név a szótárban, akkor ez az utasítás módosítja a cica fajtáját. Ha még nem létezik a név, akkor a szótár bővül a név–fajta párral.

Megemlíttük, hogy a *RemoveAll* metódus törli a szótár összes elemét:

```
TömbNév.RemoveAll()
```

Maga az objektum megmarad, újra elkezdhetjük például a szótár bővítését.

Ez a példa már jól mutatja a szótárobjektum óriási előnyét a tömbökhöz képest. Metódusai mentesítnek minket a keresés, a bővítés és a törlés algoritmusának megírása alól.

## A szótár elemeinek rendezése

Mint említettük, az elemek a felvétel sorrendjében kerülnek be a szótárba. A *For Each* utasítás is ebben a sorrendben éri el őket. Nincsen módunk meghatározni egy új elem helyét, ezért a rendezést csak utólag végezhetjük el.

A rendezés lépései:

1. Átmásoljuk a szótár kulcs–érték párait egy kétdimenziós tömbbe.
2. Rendezzük a tömböt a kulcsok vagy az értékek szerint.
3. Töröljük a szótár elemeit.
4. Visszamásoljuk az adatokat a tömbből a szótárba.
5. Szükség esetén töröljük a kétdimenziós tömböt.

A szótárobjektum *Keys* és *Items* metódusának segítségével az első lépés egyszerűen végrehajtható. Mindkét metódus egy-egy dinamikus tömböt készít a szótárban előforduló kulcsokból, illetve értékekből. Ezért a kétdimenziós tömb helyett két egydimenziós tömböt fogunk használni.

A **9–11. példában** a fenti algoritmust a cicákkal mutatjuk be. Először feltöltjük a tömböket:

```
Nev = Cica.Keys()  
Fajta = Cica.Items()
```

A rendezés pontosan az 5.1. fejezetben bemutatott algoritmus szerint végezzük, de nem csak a *Nev*, hanem a *Fajta* tömb elemeire is végrehajtjuk a szükséges cseréket. Itt a későbbiekben ismertetésre kerülő, hatékonyabb rendezési algoritmust is használhatunk. A rendezéshez szükségünk van a tömbök méretére, amit az *UBound* függvénnyel határozhatunk meg. A példában az *UBound* helyett a szótár *Count* tulajdonságát használtuk, amely megadja a szótár elemeinek (kulcs–adat értékpárjainak) a számát. Mivel a tömb indexelése 0-val kezdődik, a maximális index eggyel kisebb, mint az elemek száma:

```
MaxIndex = Cica.Count - 1
```

A következő lépés a szótár elemeinek törlése, és a tömbök elemeinek visszaírása:

```
Cica.RemoveAll()  
For I = 0 To MaxIndex  
    Cica(Nev(I)) = Fajta(I)  
Next
```

Végül töröljük a *Keys* és *Items* metódusok által létrehozott dinamikus tömböket:

```
Erase Nev  
Erase Fajta
```

Mint említettük, a szótár elemei összetett típusú adatok, például tömbök is lehetnek. A **9–12. példában** diákok adatait tartjuk nyilván egy szótárobjektumban. A diákok nevét használjuk kulcsnak, de gondoskodnunk kell arról, hogy egyforma értékek ne forduljanak elő. A kulcsokhoz tartozó értékeket egy-egy tömb tárolja, melynek elemei megadják a diák születési helyét, idejét és lakcímét. Az algoritmus pontosan meg-



felel az előző példában használnak, csak az adatbeírást és a listázást kell a tömbeknek megfelelően módosítani. A táblázatos megjelenítést az Olvasóra bízunk.

## Halmazok szimulálása

A VBScript közvetlenül nem rendelkezik egy fontos adattípussal, a halmazzal. A halmaz esetén csak arra vagyunk kíváncsiak, hogy egy meghatározott elemet tartalmaz-e vagy sem. Bármely elem csak egyszer fordulhat elő egy halmazban, és az elemek sorrendje tetszőleges (általában nem hajtunk végre rendezést).

A halmazt olyan szótárobjektummal szimulálhatjuk, melynél a kulcsok jelentik az elemeket, a hozzájuk tartozó értékeket pedig üresen hagyjuk. A halmazokkal végzett műveletek algoritmusát egy konkrét példán mutatjuk be.

A **9–13. példában** megadunk két mondatot. Készítsünk két halmazt (szótárobjektumot), melyek az egyes mondatok karaktereit tartalmazzák, de mindegyiket csak egyszer! A betűket kulcsként kezeljük, ha már bekerült egy betű a halmazba, akkor többet nem vesszük fel.

A halmazok feltöltéséhez a *Mid* függvényt használjuk. Ciklussal sorra vesszük a mondatok karaktereit, és hozzáadjuk őket a halmazokhoz. A szótárobjektumoknál említettük, hogy ha egy kifejezésben egy nem létező kulcsra hivatkozunk, akkor az új kulcs üres adattal bekerül a szótárba. A kifejezés felírásához felhasználjuk a *Temp* segédváltozót:

```
For I = 1 To Len(Mondat)
    Karakter = Mid(Mondat, I, 1) ' a mondat I-edik karaktere
    Temp = Halmaz(Karakter)
    ' a karakter szükség esetén bekerül a halmazba
Next
```

Ha már egy létező kulcsra hivatkozunk, akkor az nem kerül újra be a szótárba. A *Karakter* változót csak az áttekinthetőség kedvéért alkalmaztuk, a *Mid* függvény hívását megadhattuk volna a *Halmaz* paramétereként.

A CD-n lévő példa meg is jeleníti a halmazokat. A listázáshoz a szótárobjektumoknál már megismert módszert használjuk.

Megjegyezzük, hogy a halmaz elemeinek számát a szótárobjektum *Count* tulajdonsága adja meg.

## Halmazműveletek

A halmazok között végzett leggyakoribb műveleteket a 9–13. példa halmazaival mutatjuk be. Más halmazműveleteket hasonló módon valósíthatunk meg.

### Halmazok uniója

A halmazok uniója (egyesítése) azokat az elemeket tartalmazza, amelyek legalább az egyik halmaznak az elemei. Ehhez először átmásoljuk az egyik halmaz elemeit az unióba:

```
For Each Karakter in Elso
    Temp = Unio(Karakter)
Next
```

majd pontosan ugyanezzel az algoritmussal hozzávesszük a másik halmaz elemeit:

```
For Each Karakter in Masodik
    Temp = Unio(Karakter)
Next
```

Ismét felhívjuk a figyelmet a feltöltésnél tett megjegyzésünkre, hogy a már létező elemek nem kerülnek újra be a szótárba. Ha nem kell változatlanul hagyni az eredeti halmazokat, akkor az első ciklus elmaradhat. Elegendő a második halmaz elemeit hozzávenni az első halmazhoz.

### *Halmazok metszete*

Két halmaz metszete (közös része) azokból az elemekből áll, melyek mindkét halmaznak elemei. Most két egymásba ágyazott ciklust használunk. Az első halmaz elemei közül azokat vesszük fel a metszetbe, melyek a másodikban is benne vannak:

```
For Each Karakter in Elso
    If Masodik.Exists(Karakter) Then
        Temp = Metszet(Karakter)
    End If
Next
```

### *Halmazok különbsége*

Két halmaz különbsége azt a halmazt jelenti, melynek elemei az egyik halmazban megtalálhatók, de a másikban nem. A különbség az eddigi halmazműveletekkel szemben nem kommutatív, vagyis a két halmaz sorrendje nem cserélhető fel.

A műveletet úgy végezzük el, hogy az első halmaz elemei közül azokat vesszük fel a különbségalmazba, amelyek nem szerepelnek a másodikban (az első halmazból vonjuk ki a másodikat):

```
For Each Karakter in Elso
    If Not Masodik.Exists(Karakter) Then
        Temp = Kulonbseg(Karakter)
    End If
Next
```

Megtehetjük azt is, hogy először az első halmaz összes elemét felvesszük a különbségalmazba, majd töröljük belőle azokat, amelyek a második halmazban szintén benne vannak:

```
For Each Karakter in Elso
    Temp = Kulonbseg(Karakter)
Next

For Each Karakter in Masodik
    If Elso.Exists(Karakter) Then
        Kulonbseg.Remove(Karakter)
    End If
Next
```

Mint a következő példában látni fogjuk, ez a módszer főleg akkor hatékony, ha több halmaz különbségét kell képezni. Ekkor ciklusba foglalhatjuk az egyes halmazoknál az elemek törlését.

## Halmazok alkalmazása

Halmazok használatával lényegesen egyszerűsödnek azok az algoritmusok, ahol nem engedhetjük meg az elemek ismétlődését. A **9–14. példában** bemutatjuk az ötös-lottó számainak véletlenszerű választását. Ha egy tömbben tároljuk a már kiválasztott számokat, akkor legalább két egymásba ágyazott ciklussal tudjuk csak megvizsgálni és elkerülni az ismétlődéseket. Halmaz használatával azonban erre nincs szükség. Addig folytatjuk a választást, amíg az elemek száma el nem éri az 5-öt:

```
Do
    Temp = Tipp(Int(90 * Rnd() + 1))
Loop While Tipp.Count < 5
```

A **9–15. példa** egy játékot szimulál. Három kalóz mindegyike véletlenszerűen választ a saját kártyacsomagjából 9 lapot. Az nyer, aki több olyan lapot húz ki, amelyet mások nem választottak. A kártyalapokat sorszámozzuk, így csak a húzott lapot sorszámainak egyezését kell vizsgálnunk. Minden egyes kalóznál el kell venni a 9 lapból azokat, amelyek szerepelnek a többi kalóz által választott lapok között (halmazok különbsége). A maradék adja meg a csak általa választott lapokat. A különbségképzés, de a húzás is jóval bonyolultabb algoritmust igényelne, ha nem használnánk halmazokat.

## 9.3. Objektumok

A korszerű programozási nyelvek objektumokat kezelnek. A VBScriptben is objektumokkal végeztük a vezérlést, az adatok beolvasását és megjelenítését. Az eddig megismert objektumok legtöbbje a dokumentum objektummodellhez tartozott. Néhány objektumot a VBScript szolgáltatott számunkra, illetve külső forrásból ágyaztuk be a dokumentumba. A továbbiakban saját objektumokat hozunk létre.

Megjegyezzük, hogy saját objektumok definiálása a VBScript 5-ös változatától kezdve lehetséges, így a fejezet példáinak futtatásához az Internet Explorer 5.0-ös vagy későbbi változatára van szükség.

### Az objektumok szerepe

A programozási nyelvek objektumai nem csak a felhasználó és a számítógép közötti kommunikációt szolgálják. Ugyanilyen fontos feladatuk az adatok tárolása, feldolgozása. Mint látni fogjuk, adataikat (változóikat) elrejtik más objektumok elől, csak metódusaikon keresztül lehet hozzájuk férni. Az objektumok elrejtik az adatok feldolgozását is. Alprogramjaik közül csak azokat lehet kívülről meghívni, melyek használatát engedélyeztük. Ezzel nagymértékben csökkenthető a programozás során elkövetett hibák száma.

Az objektumok tehát egységbe foglalják, összezárlják a változókat és a hozzájuk tartozó alprogramokat. Az objektum alkalmazásánál csak azt kell tudnunk, hogy mi a szerepe, azzal nem kell foglalkoznunk, hogyan végzi el a feladatát. A bezárás és az adatok elrejtése az objektum-orientált programozás fontos elve.

Saját objektumaink ugyanúgy osztályokat alkotnak, mint a DHTML-objektumok. Az objektumok létrehozása előtt definiálnunk kell az osztályt. A definícióban az ob-

jektumok tagjai szerepelnek, melyek adatok (azaz változók) és alprogramok (függvények, eljárások) lehetnek. Az objektumok adatait gyakran adattagoknak, az alprogramokat pedig tagfüggvényeknek, illetve eljárástagoknak hívják. Egyes programozási nyelvekben az elemváltozó és az elemfüggvény elnevezés használatos.

Ha definiáltunk egy osztályt, akkor nagyon sok objektumot hozhatunk létre a segítségével. Egyiküket sem kell újból beprogramozni, mindegyikük rendelkezni fog az osztály definíciójában szereplő változókkal és alprogramokkal.

Az objektum-orientált programozási nyelvekben egy osztályból kiindulva újabb osztályokat definiálhatunk. Ezekben meghagyhatjuk az eredeti változókat és alprogramokat, de meg is változtathatjuk őket, vagy bővíthetjük a számukat. Ezt a folyamatot öröklődésnek nevezzük. Az új osztály automatikusan örökli az eredeti osztályhoz tartozó változókat és alprogramokat. A definícióban csak az eltéréseket kell megadni. Az öröklődés segítségével egyre bonyolultabb osztályokat hozhatunk létre.

Az osztályok definiálása, a változók és alprogramok bezárása, az öröklődés olyan eszközökkel bővítette a programozási nyelveket, melyek nélkül nem lehetne megírni a mai igényeket kielégítő, bonyolult programokat.<sup>28</sup>

A VBScript nem teszi lehetővé az öröklődést. Ezért objektum-orientált helyett objektumalapú programnyelvnek szokás nevezni.

### Az objektumosztály definíciója

A saját objektumosztály definícióját a globális szkriptben kell elhelyezni. A változókat és az alprogramokat a *Class* (osztály) ... *End Class* kulcsszavak közé írjuk:

```
Class Osztálynév
    változók (adatok) deklarálása
    alprogramok definiálása
End Class
```

A VBScript objektumai csak külön beállítás esetén rejtik el a változókat. Ehhez a deklarációban a *Private* (saját) kulcsszót alkalmazzuk a *Dim* helyett:

```
Private Változónév_1, Változónév_2, ...
```

Ha nem akarunk elrejtetni egy változót, akkor a deklarációban a *Public* (nyilvános) kulcsszót használjuk:

```
Public Változónév_1, Változónév_2, ...
```

A *Dim* utasítással deklarált változók szintén nyilvános (*Public*) elérésűek.

A nyilvános elérésű változók alkotják az osztály objektumainak tulajdonságait, ezeket az osztály definícióján kívül is használhatjuk. A tulajdonságok hatókörét az objektum létrehozásának helye szabja meg (globális vagy lokális). A tulajdonságokra a DHTML-objektumoknál megszokott formában, az objektum nevével minősítve hivatkozunk:

```
Objektumnév.Változónév
```

---

<sup>28</sup> Az öröklődés hiánya miatt nem tárgyaljuk a polimorfizmust.

A definícióban az osztály alprogramjait is a *Public* vagy a *Private* csoportba sorolhatjuk:

```
Public Sub Eljárásnév, illetve Public Function Függvéynév
Private Sub Eljárásnév, illetve Private Function Függvéynév
```

A *Private* hozzáférésű alprogramokat a változókhoz hasonlóan csak az osztály definícióján belül lehet használni. Külső hívás esetén hibaüzenetet kapunk.

A nyilvános (*Public*) besorolású alprogramok alkotják az osztályhoz tartozó objektumok metódusait. A metódusokra az objektum nevével minősítve hivatkozunk:

```
Objektumnév.Metódusnév
```

A metódusok tehát függvények és eljárások egyaránt lehetnek, melyek – a szokásos módon – paraméterekkel is rendelkezhetnek.

Megjelölés hiányában az osztályban definiált alprogramok nyilvános hozzáférésűek. Mivel szeretnénk hangsúlyozni a besorolást, sem a változóknál, sem pedig az alprogramoknál nem hagyjuk el a *Public* kulcsszót.

### Saját objektumosztály létrehozása

Az objektumok létrehozásához először definiálni kell az osztályt. A **9-16. példában** megadunk egy *Tarolo* nevű objektumosztályt, amit egy nem negatív szám tárolására használunk. A tárolást a *Beir* nevű metódus végzi, amely ellenőrzi a beírt karakter sorozatot és a szám előjelét. Nem megfelelő érték esetén hibaüzenetet jelenít meg:

```
Class Tarolo
Private Szam
Public Sub Beir(SzamBe)
    If Not IsNumeric(SzamBe) Then
        window.alert("Nem számot adott meg!")
    ElseIf SzamBe < 0 Then
        window.alert("Negatív számot nem tárolok!")
    Else
        Szam = SzamBe
    End If
End Sub
```

A szám kiolvasásához a *Kiolvas* metódust alkalmazzuk, amely hibaüzenetet küld a felhasználónak, ha még egyetlen számot sem tároltunk:

```
Public Function Kiolvas()
    If IsEmpty(Szam) Then
        window.alert("Még nem tároltam egyetlen számot sem!")
        Kiolvas = 0
    Else
        Kiolvas = Szam
    End If
End Function
End Class
```

Felhívjuk a figyelmet arra, hogy a *Tarol* metódus eljárás, amelynek hívása önálló utasítást képez. A *Kiolvas* viszont függvény, így hívása valamilyen kifejezésben (akár egyszerű értékadó utasításban) történhet. Ennek megfelelően az utasításai között szerepelnie kell a visszatérési értéket meghatározó értékadásnak.

A példa – bár nagyon egyszerű, – jól mutatja az adatok elrejtésének fontos következményét. Objektumaink nem „hajlandók” negatív számokat tárolni. Akárhány példányt hozunk belőlük létre, mindegyikük rendelkezni fog a hibaellenőrző metódussal, nem tudunk megfeledkezni róla. A tárolt érték elérése is csak a *Kiolvas* metóduson keresztül történhet. A metódus figyelmeztet, ha az első tárolás előtt hívjuk.

### Objektumpéldányok létrehozása

Az osztály definiálása után a *New* operátorral hozhatunk létre objektumokat. A létrehozásnál hozzá kell rendelnünk az objektumot egy objektumváltozóhoz:<sup>29</sup>

```
Dim ObjektumváltozóNév  
Set ObjektumváltozóNév = New Osztálynév
```

Az objektumváltozó neve lesz az objektum azonosítója, melynek a segítségével hivatkozhatunk rá.

A 9-16. példában három objektumot hozunk létre:

```
Dim ElsoSzam, MasodikSzam, HarmadikSzam  
Set ElsoSzam = New Tarolo  
Set MasodikSzam = New Tarolo  
Set HarmadikSzam = New Tarolo
```

Mint említettük, mindegyikük rendelkezik az osztály-definícióban szereplő adattaggal (a *Szam* változóval) és metódusokkal (*Beir*, *Kiolvas*). A hivatkozás az objektumváltozó nevével minősítve történik:

```
ElsoSzam.Beir(32)  
window.alert("A tárolt szám: " & ElsoSzam.Kiolvas())
```

A példában a *MasodikSzam* értékét úgy próbáljuk megjeleníteni, hogy nem tároltuk előtte semmit, a *HarmadikSzam* pedig negatív értéket kapna. Figyeljük meg a hibaüzeneteket! Mivel a *Kiolvas* metódus függvény, ebben az esetben is gondoskodunk visszatérési értékről, de az előtte megjelenő hibaüzenet jelzi, hogy ez nem a tárolt számnak felel meg. Mindegyik objektum ismeri a feladatát, amit külön kód írása nélkül, pontosan végrehajt.

Egy nagyobb programban nem célszerű az objektum adattagjainak írását és olvasását végző alprogramok hibaüzeneteit üzenetablakban megjeleníteni. Helyette egy speciális értékkel vagy hiba generálásával jelezhetjük a nem megfelelő adatokat.

---

<sup>29</sup> A VBScriptben csak késői kötést használhatunk. Mivel nem létezik korai kötés, a fogalom magyarázatára nem térünk ki.

## Objektumok tárolása tömbökben

Egyedi azonosítóval ellátva csak viszonylag kevés objektumot hozhatunk létre. Több objektum kezeléséhez tömböt alkalmazunk. A tömb elemei is lehetnek objektum-hivatkozások.

A **9-17. példában** a *Tarolo* osztály objektumait egy tömb elemeihez rendeljük, és a *Beir* metódussal véletlenszámokat tárolunk bennük:

```
For I = 0 To 19
    Set Adat(I) = New Tarolo
    VeletlenSzam = Int(99 * Rnd() + 1)
    Adat(I).Beir(VeletlenSzam)
Next
```

A véletlenszámok eléréséhez most is a *Kiolvas* metódust használjuk:

```
For I = 1 To 19
    Lista = Lista & Adat(I).Kiolvas() & "<BR>"
Next
```

Figyeljünk arra, hogy a ciklusban ne egyetlen létező objektumhoz rendeljünk hozzá újabb objektumváltozót! A

```
Set Temp = New Tarolo
For I = 0 To 19
    Set Adat(I) = Temp
Next
```

utasítássorozat egyetlen objektumot hoz létre. A tömb összes eleme erre az objektumra hivatkozik.

Nem csak tömbök tartalmazhatnak objektumokat, egy objektum bármely adattagja lehet újabb objektum-hivatkozás, illetve tömb is.

## Az objektumok metódusai

Az objektumok metódusai az adatok beírásán és kiolvasásán kívül más tevékenységet is végezhetnek. A **9-18. példában** *Tarolo* osztályunkat kiegészítjük a beírt szám négyzetgyökét meghatározó függvénnyel:

```
Public Function Gyok()
    Gyok = Sqr(Szam)
End Function
```

Mivel a függvényt kívülről is szeretnénk felhasználni, nyilvánosként deklaráltuk. Figyeljük meg, hogy a gyökvonás előtt nem ellenőriztük az előjelet, hiszen objektumaink nem „hajlandók” negatív számokat tárolni!

A példában a kiírást végző ciklust kiegészítettük a négyzetgyök megjelenítésével:

```
Lista = Lista & Adat(I).Gyok() & "<BR>"
```

A gyökvonást csak az egyszerűség kedvéért alkalmaztuk. Objektumaink metódusai bármilyen bonyolult utasítássorozatot tartalmazhatnak.

## Az adattagok hatóköre

Az előző példában a gyökvonást végző függvényen belül hivatkoztunk a *Szam* változóra, amit a függvényen kívül, az osztályblokk elején deklaráltunk. Az osztályblokkban (de nem az alprogramjaiban) deklarált változók az osztályon belül globálisnak számítanak. Az itt definiált alprogramokból elérhetők, akár a *Public*, akár a *Private* csoportba tartoznak. Mint tudjuk, a nyilvános változókat az objektumok nevével minősítve az objektumon kívül is felhasználhatjuk.

Egy osztály alprogramjaiban deklarált változók továbbra is lokálisak, csak az adott alprogramban hivatkozhatunk rájuk.

## Interfészek definiálása

A nyilvános elérésű adattagok használata ellentmond az adatrejtés elvének, hiszen közvetlenül írhatjuk vagy olvashatjuk őket. Ezért a 9–16. példában a *Szam* változó értékét csak a *Beir* és *Kiolvas* metódusok alkalmazásával tettük elérhetővé. A *Private* adattagok (általában ellenőrzéssel történő) beírását, illetve kiolvasását végző metódusokat gyakran interfészeknek nevezik. Az interfészeket természetesen *Public*-ként kell definiálni, hiszen kívülről hívjuk őket, amikor az objektumot „megkérjük” egy adat tárolására vagy kiolvasására.

Interfész: egy objektum valamely adattagjának írását vagy olvasását végző alprogram.

A VBScriptben az

```
ElsoSzam.Beir(32) vagy az ElsoSzam.Kiolvas()
```

jellegű utasítások azonban nem hasonlítanak azokhoz az értékadásokhoz, melyekkel például a DHTML-objektumok tulajdonságait írtuk és olvastuk. Ezt a hiányosságot a *Property* (tulajdonság) típusú alprogramokkal küszöbölhetjük ki. A *Property* interfészek segítségével elrejtjük a beolvasást és a kiírást végző alprogramokat. Az alprogram nevét úgy kezelhetjük, mintha az objektum egy tulajdonságának az azonosítója lenne. Gyakran csak az interfészekeken keresztül elérhető adattagokat tekintik az objektumok tulajdonságainak.

Interfészt úgy készítünk, hogy egy *Property Let* és egy *Property Get* metódust hozunk létre az osztály-definícióban belül. A *Property Let* eljárás írni, a *Property Get* függvény olvasni fogja az adattag értékét. A két metódusnak azonos nevet kell adni, amely kívülről a tulajdonság nevének a szerepét játssza:

```
Public Property Let Tulajdonságnév(Paraméter)
    utasítások
End Property

Public Property Get Tulajdonságnév()
    utasítások
End Property
```



A *Property Let* paraméterének segítségével a tulajdonságnak megadott értéket az objektum megfelelő *Private* változójában tároljuk. A *Property Get* ezt az értéket adja vissza, így nincs paramétere.

Az interfészek létrehozása után a tulajdonságra már a megszokott módon hivatkozhatunk:

```
Objektumnév.Tulajdonságnév
```

Ez a kifejezés szerepelhet a tulajdonság értékének beírásánál és kiolvasásánál is.

Ha egy *Private* elérésű adattaghoz nem hozunk létre *Property Get*-et, akkor a tulajdonság csak írható, a *Property Let* hiányában pedig csak olvasható lesz.

Megjegyezzük, hogy ha egy tulajdonság értéke objektum-hivatkozás (azaz újabb objektumra mutat), akkor a *Property Let* helyett a *Property Set*-et kell alkalmazni.

### Az interfészek használata

A **9-19. példában** úgy alakítjuk át a 9–18. példa kódját, hogy az *Property* interfészeket tartalmazzon.

A *Szam* tulajdonság elnevezését megtartjuk, ez lesz tehát a *Property Let* és *Property Get* alprogramok neve. Ezért megváltoztatjuk a *Class* osztályon belül a változónevet, például kiegészítjük a típust jelző *sng* előtaggal:

```
Private sngSzam
```

A *Beir* alprogram definíciójában *Sub* helyett *Property Let*-et írunk:

```
Public Property Let Szam(SzamBe)
```

az *End Sub* helyett pedig *End Property*-t. A hibaellenőrzések változatlanok maradnak, egyedül a *Private* változó értékadását módosítjuk az új változónévnek megfelelően:

```
sngSzam = SzamBe
```

Ezzel a beírást végző eljárást elkészítettük.

A kiolvasáshoz módosítanunk kell a *Kiolvas* függvény definícióját. Az első és utolsó sor az előzőeknek megfelelően:

```
Public Property Get Szam(), illetve End Property
```

A visszatérési értéket két helyen adtuk meg. Az új függvénynév miatt ezeket az utasításokat is meg kell változtatni. Ha a tulajdonság még nem kapott értéket, akkor:

```
Szam = 0
```

egyébként pedig:

```
Szam = sngSzam
```

Ezzel az osztály-definíció szükséges módosításait elvégeztük.

Az interfészek definiálása után a *Szam* tulajdonság értékének írása és olvasása a DHTML-objektumoknál megszokott módon történik:

```
Adat(I).Szam = VeletlenSzam, illetve Lista = Lista & Adat(I).Szam
```

Vegyük észre, hogy az első utasításnál az interpreter egy eljárást (*Property Let*), a másodiknál pedig egy függvényt (*Property Get*) hív meg eléggé szokatlan módon.

A *Szam* azonosítót az osztály definícióján belül is használhatjuk. A *Gyok* metódusban például a `Gyok = Sqr(sngSzam)` helyett marad a `Gyok = Sqr(Szam)`, ami ismét a *Property Get*-en keresztül éri el a változót.

Felhívjuk a figyelmet arra, hogy az *sngSzam* változó csak az adat tárolását szolgálja, a hivatkozás a tulajdonságnévvel történik mind az objektumon kívül, mind az alprogramokon belül. Magát a változót pedig csak a *Property Get*-ben, illetve a *Property Let*-ben használjuk.

### Az alapértelmezett metódus

A hivatkozásokat még tovább egyszerűsíthetjük az alapértelmezés segítségével. Egyetlen metódusnál, a leggyakrabban a *Property Get*-nél használhatjuk a *Default* (alapértelmezett) kulcsszót. A *Default*-tal kijelölt metódus nevét nem kell az objektum neve után írni. Az objektumnév kiírása egy utasításban egyenértékű az alapértelmezett metódus hívásával.

A **9-20. példában** a 9–19. példa *Property Get* metódusát alapértelmezetté tesszük:

```
Public Default Property Get Szam()
```

Ekkor a kiírásnál nem kell megadnunk a tulajdonság nevét, elegendő az objektum nevét leírni:

```
Lista = Lista & Adat(I)
```

Vegyük észre, hogy itt az objektumra való hivatkozás szintén egy metódus hívását eredményezi.

### Az objektumok inicializálása

Sokszor előfordul, hogy egy objektum létrehozásakor bizonyos utasításokat végre kell hajtani. Ezt a folyamatot az objektum inicializálásának nevezzük.

Az inicializálást végző utasításokat a *Class\_Initialize* eljárásban helyezhetjük el:

```
Private Sub Class_Initialize  
    utasítások  
End Sub
```

A *Class\_Initialize* az objektum létrehozásakor bekövetkező *Initialize* eseményt kezelő eljárás. A többi eseménykezelőhöz hasonlóan egy esemény (*Initialize*) bekövetkezése okozza a meghívását.

A **9-21. példában** a *Pelda* osztály objektumainak létrehozását üzenetablakkal jelezzük. Ehhez a *Class\_Initialize* metódusba beírjuk a

```
window.alert("Létrejött az objektum.")
```

utasítást.

A *Class\_Initialize* eljárást legtöbbször *Private* típusúként deklaráljuk, hiszen csak az objektum létrehozásakor szeretnénk végrehajtani. Közvetlen hívására általában nem adunk módot.

## Az objektumok megszüntetése

Objektumot csak úgy tudunk létrehozni, ha egyben hozzárendeljük egy objektumváltozóhoz. Ha az objektumváltozó megszűnik (például kilépünk az őt deklaráló alprogramból), akkor megszűnik az objektum is.

Az előző példában a *Temp* objektumváltozót lokálisan hoztuk létre, így az eseménykezelő eljárás végrehajtása után a változóval együtt maga az objektum szintén megszűnt.

Közvetlenül szüntethetünk meg egy objektumot, ha az objektumváltozóhoz a *Nothing* értéket rendeljük:

```
Objektumváltozó = Nothing
```

Az 5.2. fejezetben ismertettük, hogy egy objektumra több objektumváltozó hivatkozhat:

```
Set ElsoSzam = New Adat  
Set MasodikSzam = ElsoSzam
```

Ebben az esetben a *MasodikSzam* ugyanarra az objektumra hivatkozik, mint az *ElsoSzam*. Az interpreter jegyzi az adott objektumhoz tartozó hivatkozások számát. Az objektumot csak akkor szünteti meg, ha már egy hivatkozás sem mutat rá.

A fenti példában az

```
ElsoSzam = Nothing
```

utasítás még nem törli az objektumot, így a *MasodikSzam* segítségével elérhetjük a tulajdonságait és metódusait. A

```
MasodikSzam = Nothing
```

utasítás végrehajtása után már nem maradt egyetlen változó sem, amely az objektumra hivatkozik, így maga az objektum is megszűnik.

## A Terminate esemény

A fent elmondottakat a *Terminate* esemény segítségével szemléltetjük. A *Terminate* esemény közvetlenül az objektum megszűnése előtt jön létre. A *Class\_Terminate* eljárás tartalmazza a megszűnéskor végrehajtott utasításokat:

```
Private Sub Class_Terminate  
    utasítások  
End Sub
```

A *Terminate* eseménykezelő eljárását az *Initialize*-hez hasonlóan általában *Private* eléréssel definiáljuk.

A **9-22. példa** az objektum megszűnését üzenetablakkal jelzi. Ehhez a *Class\_Terminate* eljárásba a

```
window.alert("Megszűnt az objektum.")
```

utasítást írtuk. Figyeljük meg, hogy a parancsgomb eseménykezelőjének végrehajtása után megjelenik az üzenet, mert a lokális változók az eljárásból való kilépés után megszűnnek! Így megszűnik az az objektum is, amelyre hivatkoznak.

A **9-23. példában** a *Temp* változót globálisként deklaráltuk. Megszüntetéséhez külön parancsgombra van szükség, melynek *onclick* eseménykezelője a *Nothing* értéket rendeli az objektumváltozóhoz:

```
Set Temp = Nothing
```

A **9-24. példa** bemutatja az objektum megszűnését több objektum-hivatkozás használata esetén. Csak az utolsó hivatkozás törlésekor hívódik meg a *Class\_Terminate* eljárás, mert ekkor szűnik meg az objektum. A példa megértéséhez gondosan tanulmányozzuk a forráskódot!

### Objektumok használata

A **9-25. példában** olyan weblapot készítünk, amely az egérrel való kattintás helyén megjelenít egy véletlenszerűen változó kártyalapot. Ha a kártyalapra kattintunk, akkor az eltűnik a képernyőről. Az ablakban tetszőleges számú kártyalapot lehet létrehozni.

A kártyalapok kezeléséhez különböző tevékenységeket kell végezni (megjelenítés, változtatás, törlés), és több adatot is nyilván kell tartani (hely, a kép és az időzítő azonosítója stb.). Bízunk ezeket a tevékenységeket és adatokat egy-egy objektumra!

Az objektumokat az eddigiekhez hasonlóan egy tömb elemeihez rendelhetnénk hozzá. Nem tudjuk azonban törölni a tömb tetszőleges elemét, így inkább egy szótár-objektumot használunk, melynek a *Kep* azonosítót adjuk. Kulcsként a kép sorszámát alkalmazzuk. Az eddig létrehozott képek számát a *MaxIndex* változóban tároljuk:

```
MaxIndex = 0  
Set Kep = CreateObject("Scripting.Dictionary")
```

Az objektumok létrehozásához definiáljuk a *Kartya* objektumosztályt, amely

- létrehozza az IMG-objektumot,
- véletlenszerűen megválasztja a megjelenített lapot (*src* tulajdonság),
- megjeleníti a kártyalapot a kattintás helyén,
- elindítja a lap véletlenszerű változtatását (időzítő),
- szükség esetén leállítja az időzítőt, és törli a lapot a képernyőről.

#### *Az IMG-objektum létrehozása, és a lap kiválasztása*

A *Kartya* osztály inicializáló eljárásában a *createElement* metódussal hozzuk létre az IMG-objektumot. Az *src* tulajdonságot a *Valaszt* eljárásban adjuk meg, mert szükség lesz rá a lap változtatásánál is. Itt csak meghívjuk ezt az eljárást:

```
Private Sub Class_Initialize  
    Set Elem = document.createElement("IMG")  
    Valaszt()  
End Sub
```

Mivel az időzítő kívülről hívja a *Valaszt* eljárást, ezért nyilvánossá tesszük. A lapokat a *Kartya* nevű mappában tároljuk. A fájlnev egy sorszám, így a lap kiválasztásához véletlenszerűen kell választani egy számot:

```
Public Sub Valaszt()  
    Dim Lap  
    Lap = Int(52 * Rnd() + 1)  
    Elem.src = "Kártya\" & Lap & ".gif"  
End Sub
```

### *A kép megjelenítése és változtatása*

A kép megjelenítését és az időzítő indítását egyetlen eljárásban végezzük. Ezt kívülről fogjuk meghívni, ezért nyilvánossá tesszük. Az eljárás paraméterként megkapja az egérekattintás koordinátáit és a kártyalap sorszámát, amire a törlésnél lesz szükség. A sorszámot az IMG-objektum *index* tulajdonságában tároljuk. Mivel ilyen nevű tulajdonság eredetileg nem létezik, a *setAttribute* eljárással végezzük el az értékadást.

Az *Indit* eljárás először meghatározza a kép pozícióját. A 71x96 pixel méretű képet úgy jelenítjük meg, hogy az egérekattintás helyén legyen a közepe:

```
Public Sub Indit(X, Y, Sorszam)  
    With Elem.style  
        .position = "absolute"  
        .left = X - 35  
        .top = Y - 48  
    End With
```

A megjelenítéshez az elemet az *insertAdjacentElement* metódussal be kell illeszteni a dokumentumba. A BODY-objektumnak a *Torzs* azonosítót adtuk:

```
Call Torzs.insertAdjacentElement("beforeEnd", Elem)
```

Mivel hivatkozunk a törzsobjektumra, az osztály-definíciót tartalmazó szkriptet a BODY-ba tesszük.

Végül értéket adunk a képobjektum *index* tulajdonságának, és elindítjuk az időzítőt, amely a *Valaszt* eljárás hívásával 2 másodpercenként megváltoztatja a megjelenített kártyalapot:

```
Call Elem.setAttribute("Index", Sorszam)  
Temp = window.setInterval("Kep(" & Sorszam & ").Valaszt()", 2000)  
End Sub
```

Figyeljük meg, hogy az időzítő azonosítását végző *Temp* változót az osztályon belül globálisként deklaráltuk, egyébként nem tudnánk törölni a folyamatot!

### *Az időzítő leállítása és a kép törlése*

Az osztály-definícióban már csak a *Terminate* eseménykezelő megírása van hátra. Ebben leállítjuk az időzítőt, és a *removeNode* metódussal töröljük az IMG-objektumot:

```
Private Sub Class_Terminate  
    window.clearInterval(Temp)  
    Elem.removeNode()  
End Sub
```

Ezzel az osztály definícióját elkészítettük. Az osztály minden objektuma képes lesz a saját kártyalapjának létrehozására, változtatására, illetve törlésére. További felügyeletet és adminisztrációt nem igényelnek.

#### *Az objektumok létrehozása és megszüntetése*

Az egérekattintások kezelését a *document*-objektumra bízjuk. Ha képre kattintunk, akkor egyszerűen töröljük a szótárból az *index* által kijelölt elemet:

```
Sub document_onclick
  Set Elem = window.event.srcElement
  If Elem.tagName = "IMG" Then
    Kep.Remove(Elem.Index)
```

Mivel megszűnt a kártyaobjektumra mutató hivatkozás, az interpreter megszünteti magát az objektumot is. Bekövetkezik az objektum *Terminate* eseménye, amelynek eseménykezelője leállítja a megfelelő időzítőt, és törli az IMG-objektumot. Ezzel eltűnik a képernyőről a kártya képe.

Ha nem képre kattintottunk, akkor megnöveljük a kártyalapok számát jelző változó értékét, majd a szótár új elemeként létrehozunk egy új objektumot, és meghívjuk annak *Indit* metódusát. A metódusnak átadjuk az egérekattintás koordinátáit, illetve a kép sorszámát:

```
Else
  MaxIndex = MaxIndex + 1
  Set Kep(MaxIndex) = New Kartya
  Call Kep(MaxIndex).Indit(window.event.x, window.event.y, _
    MaxIndex)
End If
End Sub
```

Vegyük észre, hogy a létrehozott képek számán kívül semmilyen adatot nem kell nyilvántartanunk a globális szkriptben! Minden egyes objektum maga végzi az IMG és az időzítő azonosítójának nyilvántartását, a képek tulajdonságainak megadását, a képek megjelenítését, változtatását, törlését, az időzítő indítását és leállítását. Javasoljuk az Olvasónak, hogy próbálja meg objektumok nélkül megírni ugyanezt a programot!

Megoldásunk szépséghibája, hogy a *MaxIndex* folyamatosan nő. Egy kártyalap törlésénél nem csökkenthetjük az értékét, mert a következő objektum létrehozásánál egy, már létező indexet használnánk fel. Nem valószínű azonban, hogy a felhasználó eljutna a kattintgatásokkal az egész típusú változók értékének felső határáig.

### **Az objektumok konstruktora**

Az előző példában nem bízhattuk a *Class\_Initialize* eseménykezelőre az *Indit* eljárás meghívását (vagy utasításainak végrehajtását), mert ahhoz meg kell adnunk az egérekattintás helyének koordinátáit és a lap sorszámát. Sajnos a *Class\_Initialize* eljárás nem rendelkezik paraméterekkel. Ezért először létre kell hozni az objektumot, aztán megadni a tulajdonságok értékét, majd meghívni a szükséges metódusokat. (Az előző példában a tulajdonságok helyett az *Indit* metódus paramétereit használtuk).

Az objektum-orientált programnyelvek egy speciális alprogramot, az úgynevezett konstruktort használják az objektumok inicializálására.

Konstruktor: az objektumok létrehozásakor automatikusan meghívásra kerülő alprogram, amely paramétereinek segítségével értéket adhat az objektum adattagjainak. A konstruktor utasításai meghívhatják az objektum létrehozásakor végrehajtandó alprogramokat is.

A konstruktor neve gyakran megegyezik az objektumosztály nevével.

A VBScriptben a *Class\_Initialize* egy erősen korlátozott működésű konstruktornak tekinthető. Mint említettük, nem használhatunk paramétereket. Paraméterek hiányában viszont nem adhatjuk meg az adattagok kezdőértékét.

Megjegyezzük, hogy a konstruktor mintájára az objektum törlésekor végrehajtásra kerülő alprogramot destruktornak hívjuk. A VBScriptben a *Class\_Terminate* a destruktorhoz hasonló szerepet játszik.

#### *Konstruktor készítése a VBScriptben*

A konstruktor hiányát egy kis ügyeskedéssel pótolhatjuk. Az osztály-definíción kívül megadunk egy függvényt, amely létrehozza, és egy lokális változóhoz rendeli az objektumot. Paramétereinek segítségével értéket ad a tulajdonságoknak, majd végrehajtja a szükséges utasításokat és metódushívásokat. A függvény visszatérési értéke az új objektumra mutató hivatkozás lesz. A függvényhívással rendeljük hozzá az objektumot a végleges objektumváltozóhoz. A függvény végrehajtása után a lokális változó eltűnik, így egyetlen hivatkozás marad az objektumra.

Mivel a függvényt a *New* operátor helyett használjuk, válasszuk függvénynévnek a *New\_Osztálynév* karakterláncot! Ezzel hangsúlyozzuk az objektum-osztállyal való kapcsolatát és a konstruktor-függvényt helyettesítő szerepét:

```
Function New_Osztálynév(Paraméter_1, Paraméter_2, ...)
    Dim Temp
    Set Temp = New Osztálynév
    Temp.Tulajdonság_1 = Paraméter_1
    Temp.Tulajdonság_2 = Paraméter_2
    ...
    az objektumok létrehozásánál végrehajtásra kerülő
    egyéb utasítások és metódushívások
    Set New_Osztálynév = Temp
End Function
```

A függvény definícióját a forráskódban az osztály-definíció után tesszük, hogy még jobban kiemeljük az összetartozásukat. Ne feledkezzünk meg az utolsó utasításról, amely a létrehozott objektumra állítja a visszatérési értéket!

A fenti definíció után egy új objektumot a következő utasítással hozhatunk létre:

```
Set Objektumváltozó = New_Osztálynév(Paraméter_1, Paraméter_2, ...)
```

Ezzel a módszerrel teljes egészében elválasztottuk az objektum létrehozását és a létrehozáskor végrehajtásra kerülő utasításokat a program többi részétől. Mivel az így

definiált konstruktor-függvény nem tagja az osztálynak, benne csak a nyilvános elérhető tulajdonságokat, illetve metódusokat használhatjuk. Ezért lehetőség szerint hagyjuk meg a *Class\_Initialize* eljárást is, és amit csak lehet, ebben végezzünk el.

#### *A konstruktor-függvény használata*

A **9-26. példában** a kártyakavalkád kódjára alkalmazzuk a fent leírt módszert. A konstruktor-függvényben meg kell adni az egér pozícióját és a kártya sorszámát, illetve meg kell hívni az *Indit* metódust. A pozíció és a sorszám már az objektum tulajdonságai lesznek, így az *Indit* metódusnak nincs szüksége paraméterekre. A változókat az egyszerűség kedvéért nyilvánossá tettük, de használhattuk volna a *Property* alprogramokat is. Az új objektumok létrehozása a *New\_Kartya* függvény hívásából áll, és nem kell foglalkoznunk az *Indit* eljárás hívásával.

## 9.4. Rekordok

Mint már többször említettük, hatékony adatkezelést csak adatbázis-kiszolgálókkal együttműködve lehet végrehajtani. A VBScript sok eszköze segíti ezt a feladatot. Ezek tárgyalására a terjedelem korlátai miatt nincs módunk. A rekordok létrehozását és kezelését azonban fontosságuk miatt röviden bemutatjuk.

### A rekord

A tömbök ismertetésénél láttunk olyan példákat, amelyekben egy többdimenziós tömb egyes dimenziói különböző tulajdonságú adatokat tároltak. Kétdimenziós tömböt alkalmaztunk például személyek nevének, születési évének és helyének nyilvántartására. Ez a megoldás akkor kényelmetlen, ha mozgatni (rendezni, léptetni) akarjuk az elemeket. Vigyáznunk kell arra, hogy az egyes személyek adatai ne keveredjenek össze.

Ezt a hibalehetőséget kiküszöbölhetjük, ha a személyeket jelképező egydimenziós tömb elemeihez a három adat tárolására háromelemű tömböket rendelünk. Programjaink azonban továbbra sem lesznek áttekinthetőek. Az *Adat(3)(2)* hivatkozásból nem derül ki, hogy éppen a születési helyről van szó. Konstansok használatával azonban beszédesebbé tehetjük a hivatkozásokat:

```
Const SzulHely = 2
...
Adat(3)(SzulHely) = "Debrecen"
```

Az összetartozó adatok tárolására tömbök helyett rekordokat is használhatunk.

**Rekord:** olyan összetett adatszerkezet, amely összetartozó adatokat tároló változókat tartalmaz. A változókat a rekord mezőinek nevezzük.

Egy rekord tetszőleges számú mezőt tartalmazhat. A mezőkre a rekordnévvel minősítve hivatkozunk:

```
Rekordnév.Mezőnév
```

A rekordokat gyakran struktúráknak is nevezzük.



A VBScript nem rendelkezik ilyen adatszerkezettel, mert az objektumok teljes mértékben helyettesítik a rekordokat. Tulajdonságaik képviselik a rekordok mezőit. Az objektum sokkal többre képes a rekordnál, mert alprogramjaival fel is dolgozhatja az adatokat. Az értékadásnál például használhatjuk az interfészeket, melyekkel megvizsgálhatjuk az adatok érvényességét.

A továbbiakban az objektumok segítségével bemutatjuk a rekordok kezelésének legfontosabb lépéseit. A rekordként alkalmazott objektumok adattagjait nyilvánossá tesszük. Mivel ez az alapértelmezett besorolás, nem írjuk ki a *Public* kulcsszót. A tulajdonság helyett a továbbiakban a mező kifejezést használjuk.

### Rekordok létrehozása és értékadása

Mivel objektumokat használunk a rekordok megvalósítására, a létrehozás, a hivatkozás és az értékadás pontosan megfelel az objektumoknál alkalmazott módszereknek.

Hozzunk létre olyan rekordokat, melyek személyek nevét, születési évét és helyét tárolják! Az osztály-definíció:

```
Class Szemely
    Dim Nev, SzulEv, SzulHely
End Class
```

Egy rekord létrehozása az objektumpéldány létrehozásával történik:

```
ElsoSzemely = New Szemely
```

A mezők értékadásánál a rekord (objektum) nevét használjuk minősítőként:

```
ElsoSzemely.SzulEv = 1962
```

A *With* utasítással itt is egyszerűsíthetjük a hivatkozást:

```
With ElsoSzemely
    .Nev = "Dékány Endre"
    .SzulEv = 1951
    .SzulHely = "Debrecen"
End With
```

A **9–27. példában** az 5–29. példa táblázatának adatait rekordokban tároljuk, melyeket hozzárendelünk egy tömb elemeihez. Ne felejtsük el, hogy a rekordok objektumok, tehát a hozzárendelést a *Set* utasítással végezzük:

```
Set A(I) = New Szemely
```

Szükség esetén a mezők kezdeti értékét a *Class\_Initialize* eseménykezelőben adhatjuk meg.

### A rekordok mozgatása

A rekordok objektumok, melyekre objektumváltozókkal hivatkozunk. Emlékezzünk vissza, hogy egy új hozzárendelés esetén az objektumból nem készül új példány. Az így létrehozott hivatkozások ugyanarra az objektumra mutatnak. Amikor pedig egy objektumváltozót egy másik objektumra irányítunk, akkor további hivatkozás hiányá-

ban az általa mutatott objektum megszűnik. Ez főleg a tömbelemek átrendezésénél fordulhat elő.

Ha például az *Adatok* tömb elemei objektumváltozók, és a *Hely* indexű helyre szeretnénk beszúrni azt az objektumot, amelyre az utolsó elem hivatkozik, akkor először egy ideiglenes változóhoz rendeljük a *MaxIndex* által kijelölt objektumot:

```
Set Temp = Adatok(MaxIndex)
```

Ezután az objektumokat a tömbelemek átrendezéséhez hasonlóan az eggyel feljebb lévő elemhez rendeljük:

```
For I = MaxIndex - 1 To Hely Step -1  
    Set Adatok(I + 1) = Adatok(I)  
Next
```

Végül a felszabaduló tömbelemet ráállítjuk a *Temp* által mutatott objektumra:

```
Set Adatok(Hely) = Temp
```

Figyeljünk az utasítások sorrendjére, mert ha megszüntetünk egy objektumra vonatkozó minden hivatkozást, akkor az objektum eltűnik a memóriából!

## Rekordok használata

A létrehozástól és a hivatkozásoktól eltekintve a rekordokat ugyanúgy használjuk, mint a tömbök elemeit. A **9–28. példában** bemutatjuk az adattárolás és az adattörlés módszerét. Hasonlítsuk össze a forráskódot az 5–48. példáéval.

Az adatokat kétdimenziós tömb helyett rekordokban tároljuk, melyeket a

```
Class Szemely  
    Dim Nev, SzulEv  
End Class
```

objektumosztállyal definiálunk. A hivatkozásoknál

```
Adatok(0, I) helyett Adatok(I).Nev  
Adatok(1, I) helyett Adatok(I).SzulEv
```

szerepel.

Új személy felvételénél a rekordokat tároló tömb újradimenzionálása után az utolsó helyen létrehozunk egy rekordot, és beírjuk az adatokat (ütköző):

```
Set Adatok(MaxIndex) = New Szemely  
Adatok(MaxIndex).Nev = Nev  
Adatok(MaxIndex).SzulEv = SzuletesiEvBe.value
```

Miután megkerestük az új elem helyét, a fent vázolt módszerrel átállítjuk a hivatkozásokat. Vegyük észre, hogy a tömbelemek cseréjénél nem az objektumokat mozgattuk, hanem a hivatkozásokat írtuk át.

A törlést végző eljárás – a hivatkozások formájától eltekintve – teljesen megegyezik a tömböknél alkalmazott alprogrammal. A törlésre kerülő rekord automatikusan eltűnik, amint megszüntetjük a rá mutató hivatkozást.

## Tömböt tartalmazó mezők

A rekordok mezői – mint az objektumok tulajdonságai – tetszőleges típusú változók lehetnek. Hivatkozhatnak újabb objektumokra, vagy tartalmazhatnak tömböket is. Ezzel meglehetősen összetett adatszerkezeteket lehet létrehozni.

Ha a mező értéke tömb, akkor így hivatkozunk a tömb egy elemére:

```
Rekordnév.Mezőnév(Index)
```

A **9–29. példa** diákok jegyeit tartja nyilván a különböző tantárgyakból. Az egyszerűség kedvéért csak 4 diákot és 3 tantárgyat kezelünk. A diákokat a *Diak* statikus tömb elemei jelentik. A tantárgyi jegyeket *Jegyek* típusú rekordban tároljuk, melynek minden mezője egy-egy dinamikus tömböt tartalmaz. Így könnyebben tudjuk az adatokat eljárással kezelni, mintha maga a mező lenne a tömb. Az osztály-definíció *Class\_Initialize* eljárása rendeli hozzá a rekordokhoz a dinamikus tömböt:

```
Class Jegyek
  Dim Magyar, Matematika, Enek
  Dim Tomb()
  Private Sub Class_Initialize
    Magyar = Tomb : Matematika = Tomb : Enek = Tomb
  End Sub
End Class
```

A jegyeket az *Osztalysz* eljárás írja be a tömbökbe. A beolvasás helyett véletlenszerűen választott osztályzatokat használunk (egyest most nem adunk). A szubrutin a tantárgyi jegyek számát is véletlenszerűen választja meg. Ennek megfelelően újradimenzionálja a dinamikus tömböt. Így tantárgyanként eltérő számú jegyet kezelhetünk.

Az eljárásnak a rekord egy-egy mezőjét kell átadni. Mivel objektumokat használunk, az objektumváltozók nem tartalmazzák magát a mezőt, csak az objektumra mutató hivatkozást. Ezért a mezőt közvetlenül nem tudjuk átadni az eljárásnak. A hívás előtt a mező értékét beírjuk egy ideiglenes változóba, és ezt lesz a eljárás aktuális paramétere. A referencia szerinti paraméterátadás miatt ki kell írunk a *Call* kulcsszót:

```
With Diak(I)
  Temp = .Magyar
  Call Osztalysz(Temp)
```

A végrehajtás után a változó értékét visszaadjuk a mezőnek:

```
.Magyar = Temp
```

Ezt az értékadást nem tudtuk volna végrehajtani, ha a mező maga lett volna a tömb.

A CD-n található példa tartalmazza még az *Osztalysz* eljárás, illetve az osztályzatok megjelenítését végző alprogram kódját. A megjelenítést a szokásos módon hajtjuk végre.

A fenti megoldás helyett megtehettük volna, hogy az egész objektumot (rekordot) átadjuk az eljárásnak.

## Objektumot tartalmazó mezők

A hivatkozás formája, ha a mező egy újabb rekord (objektum):

```
Rekordnév.EgyikMezőnév.MásikMezőnév
```

A **9–30. példában** a személyek adatainak a tárolásánál a születési dátumot felbontjuk év, hónap, nap részekre. Ehhez definiálunk egy *SzulDatum* objektumosztályt:

```
Class SzulDatum  
    Dim Ev, Honap, Nap  
End Class
```

A nevet és a születési helyet a már ismert módon adjuk meg:

```
Set A(0) = New Szemely  
A(0).Nev = "Dékány Endre"  
A(0).SzulHely = "Debrecen"
```

A születési dátum tárolásához először létre kell hozni az újabb objektumot:

```
Set A(0).SzulDatum = New SzulDatum
```

majd a megfelelő minősítéssel ellátva adhatunk értéket a mezőknek:

```
A(0).SzulDatum.Ev = 1951  
A(0).SzulDatum.Honap = 12  
A(0).SzulDatum.Nap = 5
```

A *With* utasítás nagymértékben lerövidíti a kódot:

```
Set A(1) = New Szemely  
With A(1)  
    .Nev = "Kiss István"  
    Set .SzulDatum = New SzulDatum  
    With .SzulDatum  
        .Ev = 1963 : .Honap = 4 : .Nap = 15  
    End With  
    .SzulHely = "Budapest"  
End With
```

Az értékadásnál általában ciklust alkalmazunk.

A fentiekben arra is példát mutattunk, hogyan tartalmazhat egy objektum egy másik objektumot.

Megjegyezzük, hogy a modális párbeszédablakoknál a *dialogArguments* vagy a *returnValue* értéke szintén lehet rekord (objektum). Így sokkal kényelmesebb és áttekinthetőbb módon adhatunk át egyszerre több paramétert, mint tömbök segítségével.

A klasszikus rekord adatszerkezetet a VBScriptben objektumok helyett tömböket tartalmazó változókkal is létrehozhatjuk. Ebben az esetben a mezőket ciklussal tudjuk kezelni. Ha az indexek numerikus értéke helyett konstansokat használunk, akkor csak a szintaxis fog eltérni a rekordokétól. Az igényeket és lehetőségeket elemezve kell eldöntetünk, hogy melyik módszert választjuk.

## 10. FÁJLKEZELÉS

Az eddig bemutatott példákban az adatok a böngésző ablakának bezárása után eltűntek a memóriából. A VBScript lehetővé teszi, hogy a szükséges értékeket fájlokban tároljuk. Az elmentett adatokat természetesen be is lehet olvasni.

A weblapok tudunk nélkül nem használhatják a háttértárat. A böngésző rákérdez a fájlműveletek engedélyezésére. Ezt úgy kerülhetjük el, hogy az állományokat .htm helyett .hta kiterjesztéssel látjuk el (HTML-alkalmazások). A szkriptek fájlkezelését a fejlettebb vírusirtó programok is figyelik. A figyelmeztető üzenettel együtt megjelenő menü segítségével engedélyezhetjük a szkript végrehajtását. Az üzenet csak a vírus lehetőségére utal. A könyv példáiban szereplő szkriptek természetesen semmilyen vírust nem tartalmaznak, és nem okoznak kárt a háttértáron.

### 10.1. A fájlrendszer objektumai

#### A fájlrendszerobjektum

A VBScript a mappák és fájlok kezeléséhez a Script Runtime osztálykönyvtár File System Object (fájlrendszerobjektum) modelljét használja. A modell szerint minden meghajtó, mappa és fájl egy-egy objektum, melynek tulajdonságait az objektumoknál megszokott módon érjük el. A fájlrendszer objektumain a metódusok segítségével hajthatunk végre műveleteket.

A meghajtó-, mappa- vagy fájlobjektumok eléréséhez először létre kell hozni egy fájlrendszerobjektumot:

```
Set FSO = CreateObject("Scripting.FileSystemObject")
```

A programban egyetlen ilyen objektum jön létre akkor is, ha többször megismételjük a fenti utasítást. Az objektum neve bármilyen deklarált változónév lehet. A továbbiakban a szokásoknak megfelelő FSO elnevezést használjuk.

Az FSO-objektum segítségével létrehozható objektumok és kollekciók osztályai:

<i>Drive:</i>	meghajtó, beleértve a CD-olvasókat és a hálózati meghajtókat is
<i>Drives:</i>	a számítógéphez csatlakozó meghajtók kollekciója
<i>File:</i>	fájl
<i>Files:</i>	a mappa által tárolt fájlok kollekciója
<i>Folder:</i>	mappa
<i>Folders:</i>	a mappa almappáinak kollekciója
<i>TextStream:</i>	szöveges fájl

Megjegyezzük, hogy ha már nem használunk egy objektumot, akkor célszerű a hivatkozást a *Nothing* segítségével törölni.

## A meghajtóobjektumok és kollekciójuk

A meghajtóobjektumot az *FSO* *GetDrive* metódusával rendelhetjük hozzá az objektumváltozóhoz:

```
Set Objektumváltozó = FSO.GetDrive(név)
```

A név lehet a meghajtó betűjele (kettősponttal vagy anélkül), illetve egy hálózati megosztott mappa elérési útja (\\számítógépnév\megosztásnév).

A *Drive*-objektumok legfontosabb tulajdonságai:

<i>DriveLetter</i> :	a meghajtó betűjele kettőspont nélkül
<i>DriveType</i> :	a meghajtó típusa
<i>FreeSpace</i> :	a szabad terület mérete bájtban
<i>IsReady</i> :	a háttértár készenléti állapotát jelző logikai érték
<i>Path</i> :	a meghajtó betűjele kettősponttal (elérési út)
<i>RootFolder</i> :	a gyökérkönyvtár elérési útja
<i>TotalSize</i> :	a háttértár teljes kapacitása bájtban
<i>VolumeName</i> :	kötetcímke

A fentiek közül egyedül a kötetcímket lehet a szkriptekből megváltoztatni, a többi tulajdonság csak olvasható. A háttértár tulajdonságaira (kötetcímke, kapacitás stb.) csak akkor hivatkozhatunk, ha az objektum *IsReady* tulajdonságának értéke *True*, különben hibaüzenetet kapunk.

Az *FSO*-objektum létrehozásakor elkészül a *Drives* kollekció is, amely a számítógéphez csatlakozó meghajtókat tartalmazza. Az objektumok számát a kollekció *Length* tulajdonsága adja meg.

Az objektumok használatát a **10–1. példa** mutatja be. A forráskód *Select Case* utasításában láthatjuk a különböző típusok kódját.

## A mappaobjektumok és kollekcióik

Tulajdonságainak olvasásához a mappát az *FSO*-objektum *GetFolder* metódusával hozzá kell rendelni egy objektumváltozóhoz:

```
Set Objektumváltozó = FSO.GetFolder("elérési_út")
```

Megadhatunk abszolút és relatív elérési utat is.

A mappaobjektum legfontosabb tulajdonságai:

<i>IsRootFolder</i> :	logikai érték, a gyökérkönyvtár esetén <i>True</i> , egyébként <i>False</i>
<i>Name</i> :	a mappa neve (írható)
<i>ParentFolder</i> :	a szülő mappaobjektum (!)
<i>Path</i> :	a mappa teljes elérési útja
<i>Size</i> :	a mappa mérete az almappákkal együtt

A mappa nevét a szkriptekből meg is tudjuk változtatni.

A mappaobjektum két kollekcióval rendelkezik. A *Files* a mappa összes állományát tartalmazza, a *SubFolders* pedig az almappáit. A kollekciók elemeinek a számát *Count* tulajdonságaik értéke adja meg.

A tulajdonságok használatát a **10–2. példa** mutatja be.

Új mappát az *FSO*-objektum *CreateFolder* metódusával hozhatunk létre, melynek paramétere az új mappa elérési útja. A metódus visszatérési értéke a mappához rendelt objektumváltozó:

```
Set Objektumváltozó = FSO.CreateFolder("elérési_út_és_mappanév")
```

A **10–3. példában** egy VBScript nevű mappát hozunk létre a C: meghajtó gyökérkönyvtárában. Ha már létezik ilyen mappa (például többször futtatjuk a szkriptet), akkor hibaüzenetet kapunk. Ezért a létrehozás előtt az *FSO*-objektum *FolderExists* metódusával megvizsgáljuk a mappa létezését:

```
If FSO.FolderExists("C:\VBScript") Then  
    window.alert("Már létezik ilyen mappa.")  
Else  
    Set Mappa = FSO.CreateFolder("C:\VBScript")  
End If
```

A CD-n található példában kiíratjuk a mappa tulajdonságait is.

Egy mappát az *FSO DeleteFolder* metódusával törölhetünk, melynek paramétere az elérési út a mappa nevével. A metódus az állományokkal és almappákkal együtt törli a mappát. Ha a mappában csak olvashatóként megjelölt fájlok, illetve mappák is találhatóak, akkor második paraméterként adjuk meg a *True* logikai értéket, különben hibaüzenetet kapunk.

A **10–4. példában** a megfelelő gombra kattintva létrehozhatunk, illetve törölhetünk egy mappát. Az ellenőrzést a fájl űrlapmező segítségével végezhetjük el, amit most csak a mappák listájának megjelenítésére használunk, tehát a Mégse gombbal lépünk ki belőle.

Megemlítjük, hogy mappát létrehozni és törölni egy mappákból álló kollekció *Add*, illetve a mappa *Delete* metódusával is lehet. Az *FSO*-objektum *CopyFolder* metódusa az első paraméterként megadott mappát a második paraméter által kijelölt helyre másolja, a *MoveFolder* metódus pedig a mappa mozgatását végzi.

## A fájlobjektumok

Tulajdonságainak eléréséhez az állományt az *FSO GetFile* metódusával hozzá kell rendelni egy objektumváltozóhoz:

```
Set Objektumváltozó = FSO.GetFile("elérési_út")
```

A fájl létezését az *FSO FileExists* metódusával vizsgálhatjuk meg, amit a *FolderExists*-hez hasonlóan használunk.

A fájlobjektumok legfontosabb tulajdonságai:

<i>DateCreated:</i>	a létrehozás ideje
<i>Name:</i>	a fájl neve (írható)
<i>ParentFolder:</i>	a fájl mappája
<i>Path:</i>	a fájl teljes elérési útja
<i>Size:</i>	a fájl mérete bájtokban
<i>Type:</i>	a fájl típusának megnevezése, ami megfelel az Intéző Típus oszlopában feltüntetett sztringnek

A fájl nevét meg is változtathatjuk.

A **10–5. példában** a mappa *Files* kollekciójának a segítségével kilistázzuk a C:\Windows mappa fájljainak tulajdonságait.

A fájlok kezeléséhez tartozik az *FSO CopyFile* metódusa, amellyel másolni, *MoveFile* metódusa, amellyel mozgatni és a *DeleteFile* metódusa, amellyel törölni lehet a paraméterként megadott állományt. A metódusok használatát a függelékben található feladatokban mutatjuk be.

### Az aktuális mappa megadása

A fájlrendszer használata esetén kényelmetlen lehet minden egyes esetben megadni a fájl teljes elérési útját. Erre nincsen szükség, ha kijelöljük az aktuális mappát. Sajnos a fájlrendszer-objektum nem rendelkezik ilyen lehetőséggel. Ezért az aktuális mappa megadásához a Windows Script Host (WSH) szolgáltatást vesszük igénybe.

A WSH lehetővé teszi az operációs rendszer szkriptekkel történő vezérlését, az adminisztráció egyszerűsítését, a feladatok automatikus végrehajtását. A WSH ismertetése külön kötetet igényelne. Az Olvasó figyelmébe ajánljuk a Windows Script Technologies dokumentációját, amelyben a VBScript leírását szintén megtalálja. A WSH-t VBScript nyelven is programozhatjuk. Érdemes megismerni vele!

Az aktuális mappát a *WshShell* (WSH környezet) objektum *CurrentDirectory* (aktuális mappa) tulajdonsága adja meg. Olvasásához először létre kell hozni egy objektum-hivatkozást:

```
Set WshShell = CreateObject("WScript.Shell")
AktualisMappa = WshShell.CurrentDirectory
```

A *CurrentDirectory* értékét a szkriptben meg is változtathatjuk. A **10–6. példa** megjeleníti a betöltéskor aktuális mappát, majd megváltoztatja az előzőekben létrehozott C:\VBScript-re. (Ha ez nem létezik, akkor hibaüzenetet kapunk.)

A **10–7. példában** átmásoljuk a Windows mappa Notepad.exe állományát a VBScript mappába, és megváltoztatjuk a fájlnevet Jegyzetömbre. A másolás előtt a C:\VBScript-et jelöljük ki aktuális mappának, így a cél esetén nem kell megadni az elérési utat. A WSH *Run* metódusával el is indíthatjuk az alkalmazást.



## 10.2. Szövegfájlok

A VBScript közvetlenül csak szövegfájlokat tud létrehozni, írni, olvasni. Adatainkat karaktersorozatokként tárolhatjuk ezekben az állományokban. Az *Asc* és *Chr* függvények segítségével azonban bináris kódolású fájlokat is kezelhetünk.

### Szövegfájlok kezelése

A szövegfájlok ANSI- vagy Unicode-kódolású karaktersorozatokat tartalmaznak. Ezeket az állományokat egyszerű szövegszerkesztőkkel, például a Jegyzettömbbel olvashatjuk és módosíthatjuk. A HTML-állományok is szövegfájlok. A szövegfájl-objektumot a VBScriptben gyakran *TextStream*-nek (szövegfolyamnak) nevezik.

A szövegfájlok sorokból állnak. Az egyes sorokat karakterek alkotják, a sor végét az *EndOfLine* (sorvége) kód jelzi, amely függ az operációs rendszertől. Az állomány végét az *EndOfStream* (adatfolyam vége) speciális kód mutatja.

A szövegfájlokat tartalmuk eléréséhez meg kell nyitni. Egyszerre több állományt is megnyithatunk. A megnyitás során a fájlt hozzárendeljük egy objektumváltozóhoz, és megadjuk a használat módját. A fájlt megnyithatjuk csak olvasásra, ekkor az elejétől kezdve egymás után beolvashatjuk a karaktereit vagy a sorait. Ha írásra nyitjuk meg, akkor az elejétől kezdve karaktereket vagy sorokat írhatunk bele. Ebben az esetben az állomány előző tartalma elvész. Hozzáfűzés esetén a kiírt karakterek vagy sorok a fájl végéhez kapcsolódnak.

Használat után az állományt le kell zárni. Lezárás nélkül adatokat veszthetünk, mert az operációs rendszer nem végzi el azonnal a mentést.

A fájlt egyszerre csak egyféle művelet elvégzéséhez nyithatjuk meg. Ha olvasás után írni akarunk bele, akkor először le kell zárni, majd újra megnyitni.

A szövegfájlok úgynevezett soros (szekvenciális) elérésű állományok. Az adatokhoz a tárolás sorrendjében férünk hozzá. Egy adat eléréséhez be kell olvasni az összes előtte lévő értéket. Nem tudunk közvetlenül ráállni egy meghatározott sorszámú karakterre vagy sorra.

A közvetlen hozzáférésű fájloknál megadhatjuk a beolvasásra kerülő adat helyét, azonban ezzel a lehetőséggel a VBScript nem rendelkezik.

### Szövegfájlok létrehozása, megnyitása és lezárása

Szövegfájlt az *FSO.OpenTextFile* metódusával nyithatunk meg, illetve hozhatunk létre:

```
Set Objektumváltozó = FSO.OpenTextFile(név, mód, létrehoz, kód)
```

Az egyes paraméterek jelentése:

<i>név:</i>	a fájl elérési útja és neve
<i>mód:</i>	megnyitás olvasásra (1), írásra (2) vagy hozzáfűzésre (8)
<i>létrehoz:</i>	<i>True</i> , ha nem létező név esetén hozza létre az állományt
<i>kód:</i>	Unicode esetén <i>True</i> , ANSI-kód esetén <i>False</i>

A paraméterek közül csak a nevet kötelező megadni. Alapértelmezés szerint a megnyitás olvasásra történik, az utolsó két paraméter értéke pedig *False*. Ha a *létrehoz* paraméter értéke *False*, és nem létező fájlt akarunk megnyitni, akkor hibaüzenetet kapunk. *True* esetén az interpreter létrehozza, és megnyitja az állományt.

A fájlt az objektum *Close* metódusával zárjuk le:

```
Objektumváltozó.Close()
```

Megjegyezzük, hogy állományokat az *FSO* vagy egy mappaobjektum *CreateTextFile* metódusával is létrehozhatunk. A megnyitást a fájlobjektum *OpenAsTextStream* metódusával végezzük. A paraméterezést a Scripting Runtime osztálykönyvtár dokumentációjában olvashatjuk. A továbbiakban a létrehozáshoz és a megnyitáshoz egységesen a fent ismertetett *OpenTextFile* metódust használjuk.

A **10–8. példában** létrehozuk a C:\Gyakorlás mappát, és elhelyezünk benne egy Szöveg.txt nevű állományt. A hibaüzenetek elkerülése érdekében mind a mappa, mind pedig a fájl esetén megvizsgáljuk, hogy már létezik-e. Ha a fájl létezik, akkor olvasásra nyitjuk meg. A szkript végén szabályosan le is zárjuk az állományt.

A folyamatot üzenetablakok kísérik, melyek szövege egyben magyarázza a forráskódot. Ha ellenőrizzük az Intézővel a fájlt, akkor láthatjuk, hogy egyelőre üres.

## Szövegfájlok írása

A fájlba meghatározott számú karaktert vagy egyszerre egy teljes sort írhatunk. Karakterek írása esetén is célszerű időnként lezárni a sorokat. Így könnyebben tudjuk beolvasni az adatokat.

Karaktereket a fájlobjektum *Write* metódusával írhatunk az állományba:

```
Objektumváltozó.Write(sztring)
```

A *WriteLine* metódus a sztring után egy sorvége (EndOfLine) jelet helyez a fájlba, amit felhasználhatunk a beolvasásnál:

```
Objektumváltozó.WriteLine(sztring)
```

Ha csak sorvége jelet akarunk kiírni, akkor paraméter nélkül alkalmazzuk a *WriteLine* metódust. A sorvége jel kivételével a kiírt karakterláncok közé semmilyen elválasztójel sem kerül.

A **10–9. példa** néhány karakterláncot ír a Szöveg.txt fájlba. A metódusok használatának bemutatásához az egyes sorokat egymástól eltérő módon írtuk ki. A szkript futtatásához szükség van az előző példa által létrehozott mappára és állományra.

Futtassuk többször is a programot (frissítsük a dokumentumot). Mivel a fájlt írásra nyitottuk meg, a szöveg mindig az elejétől kezdve kerül bele.

Ha az állományt hozzáfűzésre nyitjuk meg, akkor az előzőleg beleírt szöveg megmarad, az új szöveg a fájl végére kerül. A **10–10. példa** újabb versszakot fűz az előzőleg elkészített fájlhoz. Ha többször futtatjuk, akkor minden egyes alkalommal hozzáfűzi a második versszakot.

## Szövegfájlok beolvasása

A szövegfájl tartalmát a megnyitás után a fájlobjektum következő metódusaival olvashatjuk:

*ReadLine()*: beolvas egy sort  
*ReadAll()*: beolvassa a fájl hátralévő részét  
 (ha még nem kezdtük el a beolvasást, akkor a teljes tartalmát)  
*Read(n)*: beolvas n darab karaktert

A metódusok visszatérési értéke a beolvasásnak megfelelő sztring.

A beolvasást a fájlobjektum *AtEndOfLine* és *AtEndOfStream* tulajdonságával vezérelhetjük, melyek értéke *True*, ha elértük egy sor, illetve az állomány végét.

A **10–11. példában** különböző módszerekkel beolvassuk az előző feladatban készített szövegfájlt. Ne felejtjük el, hogy a megjelenítésnél a böngésző csak akkor kezd új sort, ha BR-objektumot helyezünk el a dokumentumban!

A karakterenkénti beolvasásnál ügyeljünk arra, hogy a Windowsban a sorvége jel két karakternek felel meg. Ezért ezt a két karaktert ki kell hagynunk. Ehhez a *Skip* metódust használjuk. A *Skip* átlépi a paramétereként megadott számú karaktert:

```
If Fajl.AtEndOfLine Then
    Fajl.Skip(2)
    Szoveg = Szoveg & "<BR>"
Else
    Szoveg = Szoveg & Fajl.Read(1)
End If
```

A *SkipLine()* metódussal egy egész sort hagyhatunk ki. A *Skip(2)* helyett a *ReadLine()* metódust is használhattuk volna, amely az *AtEndOfLine* jellel együtt a sor végéig beolvassa a hátralévő részt. Ekkor nem szükséges tudnunk, hogy a sorvége jel hány karakterből áll.

## Változók értékének kiírása és beolvasása

A szövegfájlokat a szkriptek változóinak mentésére is használhatjuk. Célszerű minden változót a *WriteLine* metódussal kiírni, így a *ReadLine* metódussal könnyen beolvashatjuk az értéküket. Vigyázni kell arra, hogy a beolvasás a kiírásnak megfelelő sorrendben történjen.

Ha egy sorba több változót írunk, akkor gondoskodjunk valamilyen elválasztójel (szóköz, speciális karakter stb.) alkalmazásáról. Ebben az esetben a beolvasás után a *Split* függvényvel tudjuk a karaktersorozatot szétszedni.

A **10–12. példában** a szövegmezőkbe írt értékeket mentjük el a Változók.txt fájlba, amit a 10–8. példában létrehozott C:\Gyakorlás mappába helyezünk. Az állományt a

```
Set Fajl = FSO.OpenTextFile(Ut, Iras, True)
```

utasítással nyitjuk meg. A harmadik paraméter miatt ez azt utasítás létre is hozza a fájlt, ha még nem létezik. Mivel írásra nyitjuk meg, az előzőleg kiírt adatok elvesznek.

Ha felhasználó a fájl létrehozása előtt a Beolvasás gombra kattint, akkor hibaüzenetet jelenítünk meg, és kilépünk az eseménykezelő eljárásból.

## Hibakezelés a fájlműveletek során

A fájlműveleteknél könnyen előfordulhatnak futási hibák. Egy mappa vagy állomány létrehozása, illetve törlése előtt mindig ellenőrizni kell, hogy létezik-e az adott objektum, különben hibaüzenettel megszakad a programunk végrehajtása. Az írást meggátolhatja, ha egy másik program, például a Microsoft Word már használja a szövegfájl. A hibás adathordozó is megszakíthatja a szkriptet. Ezeket a hibalehetőségeket nem tudjuk kiküszöbölni előzetes vizsgálatokkal, mert csak a metódusok végrehajtása közben derülnek ki.

A futási hibák miatt bekövetkező leállást az *On Error Resume Next* utasítással kerülhetjük el. Az utasítás kikapcsolja az interpreter hibakezelését. Az utasítás működését az *Err*-objektummal együtt a 7.4. fejezetben ismertettük.

A **10–13. példában** bemutatjuk a hibakezelés módját. A fájlrendszer használata előtt elhelyezzük a szkriptben az *On Error Resume Next* utasítást, majd minden egyes fájlművelet után megvizsgáljuk az *Err*-objektum *Number* tulajdonságát. Ha nem nulla, akkor nem lehetett végrehajtani az utasítást. Ebben az esetben hibaüzenetet jelenítünk meg, felszabadítjuk az objektumváltozókat, és kilépünk a programból. Ezeket az utasításokat hibakezelő eljárásba is helyezhetjük. Ha nem lépünk ki az eljárásból, akkor ne feledkezzünk meg az *Err.Number* törléséről!

A Windows különböző változatai nem egyformán reagálnak a fájlkezelés hibáira. A Windows 98 például a rendszerhiba kék képernyőjével leáll, ha nem éri el a szükséges háttértárat. A késleltetett mentése miatt az is előfordulhat, hogy íráskor nem kapunk hibajelzést, az adatok mégsem kerülnek a háttértárra. Ekkor az operációs rendszer értesíti a felhasználót az előforduló problémákról.

## Szövegfájlok használata

Fájlkezelésre nagyon gyakran szükségünk van a programok futtatásánál. Nagy mennyiségű adatot csak állományokban tudunk tárolni. A feldolgozáshoz fájlokból olvassuk be az adatokat, és fájlokba mentjük az eredményeket.

A fájlkezelést gondosan tervezzük meg, majd alaposan próbáljuk ki. Az állományokat minél hamarabb zárjuk le. A beolvasás és a mentés külön egységet képezzen. Az adatok feldolgozása közben lehetőleg ne legyenek megnyitva a fájlok.

A **10–14. példában** bemutatjuk az adatok beírásának, mentésének és a háttértárról történő beolvasásának módszerét. Az 5–48. példa programját bővítjük ki a fájlkezeléssel. Ebben a példában személyek nevét és születési évét lehetett táblázatosan megjeleníteni. Most a begépelt értékek nem vesznek el az ablak bezárásakor.

Az *Adatok.txt* szövegfájl a C:\Gyakorlás mappába tesszük. A Mentés parancsgomb *onclick* eseménykezelőjében létrehozuk a fájlrendszerobjektumot, majd ellenőrizzük, hogy létezik-e a mappa. Ha nem, akkor elkészítjük. A következő lépés az állomány megnyitása. Ezeket a feladatokat az eddigieknek megfelelő módon végezzük.

A hibavizsgálatok során egy eljárást hívunk, melynek paraméterként átadjuk a hibaüzeneteket tároló *Uzenet* tömb megfelelő indexét. Az alprogram megjeleníti a hibaüzenetet, majd felszabadítja a fájlrendszer-objektumait. A visszatérés után kilépünk a

mentést végző eljárásból. A rövidebb kód érdekében az *If ... Then* utasítások tömörebb formáját használjuk.

Ha sikerült a fájl megnyitása, akkor egy ciklussal kimentjük az *Adatok* tömb elemeit. Ezt követően lezárjuk az állományt, felszabadítjuk az objektumokat, és értesítjük a felhasználót a művelet sikeres befejezéséről.

A Mentés parancsgomb használatát a dokumentum betöltésekor letiltjuk. Csak akkor engedélyezzük, ha már beolvastunk vagy betöltöttünk adatokat. Így az eredeti szkriptet kiegészítjük a Mentés gomb engedélyezését és letiltását végző utasításokkal.

A *window onload* eseménykezelőjében megvizsgáljuk, hogy létezik-e az *Adatok.txt* fájl. Ha igen, akkor a program az előzőleg mentett adatok betöltésével indul. Az adatokat először a *Nev* és a *SzulEv* változókba olvassuk be, majd ellenőrizzük a fájlművelet közben keletkező hibákat. Az értékeket csak hibátlan beolvasás után adjuk át az *Adatok* tömbnek.

Az állomány lezárását követően megjelenítjük a táblázatot a képernyőn. A programban összevonhattuk volna a *window\_onload* és a *Beolvas\_onclick* eljárások azonos funkciójú részeit, de nem akartuk megváltoztatni az eredeti szkript alprogramjait. Így kerülhetjük el a legbiztosabban egy létező program bővítése során fellépő hibákat.

Ha sikerült adatokat beolvasni a háttértárról, akkor engedélyezzük a Törlés és a Mentés gombok használatát.

Egy tisztességes program a kilépés előtt ellenőrzi, hogy elvégezték-e az adatok mentését. Ezt a *Mentett* logikai változó bevezetésével érjük el, melynek először igaz értéket adunk (nincsenek adatok). Hamisra kell állítani az értékét minden beolvasásnál és törlésnél. Ismét igaz lesz az értéke a mentés után.

Ha a felhasználó bezárja az ablakot, akkor bekövetkezik a *window*-objektum *onunload* eseménye. Ennek eseménykezelőjében megvizsgáljuk a *Mentett* változó értékét. Ha ez hamis, akkor az *MsgBox* függvény segítségével lehetőséget adunk a felhasználónak, hogy elvégezze a mentést.

## Automatikusan működő alkalmazás készítése

A HTML-alkalmazások segítségével készíthetünk olyan szkriptet, amely minden beavatkozás nélkül lefut. Az adatokat a háttértárról olvassa, az eredményeket pedig a háttértárra menti. Ha kis méretben indítjuk el, és letiltjuk a tálcán való megjelenést, akkor szinte észre sem vesszük a működését.

A **10–15. példa** a C:\Gyakorlás mappába készít egy *Lottószámok.txt* állományt, melyet minden futtatásnál újabb tippekkel bővít.

A szkriptben először véletlenszerűen választunk öt lottószámot, majd ellenőrizzük, hogy létezik-e már a fájl. Ha nem, akkor létrehozuk, és beleírunk egy magyarázó szöveget. Ezután lezárjuk, mert olvasásra fogjuk megnyitni.

A létező fájlban megszámloljuk a sorokat. A sorok száma alapján megállapítjuk a következő tipp sorszámát. Ezt követően lezárjuk az olvasásra megnyitott állományt, majd az újabb tipp hozzáfűzéséhez ismét megnyitjuk.

Miután elvégeztük a feladatot, bezárjuk az ablakot. A végrehajtásról úgy győződhetünk meg, hogy minden futtatás után megnézzük a Lottószámok.txt fájl tartalmát. A Gyakorlás mappának léteznie kell, különben hibaüzenettel leáll a program.

### HTML-állomány készítése

A szövegfájlokban nem csak adatokat tárolhatunk. Készíthetünk megformázott táblázatokat közvetlenül a nyomtatáshoz, vagy egy szövegszerkesztővel történő további feldolgozáshoz. A HTML-állományok is szövegfájlok, így akár teljes weblapokat összeállíthatunk szkriptjeink segítségével.

A **10–16. példában** olyan weblapot hozunk létre, amely egy verset jelenít meg. A szöveget közrefogó HTML-kódot a Keret.htm fájl tartalmazza, melynek három csillaggal jelölt helyére beszurjuk a Szoveg.txt állomány sorait. (A vers helyét más karakterorozattal is jelölhetnénk.)

A fájlok kiválasztását egy *file* típusú INPUT-objektummal végezzük, amelyet a 8.3. fejezetben ismertettünk. Így a felhasználó tetszőleges állományt megadhat. A megnyitás előtt ellenőrizzük a fájlok létezését. Az elkészített weblap a C:\Gyakorlás mappába kerül Weblap.htm néven, de a programot módosíthatjuk úgy, hogy ezt is a felhasználó határozza meg.

A szkriptben beolvassuk a Keret.htm fájl sorait, és egyből kiírjuk a Weblap.htm-be, amíg csak el nem érünk a három csillaghoz. Mivel ezt a sort nem akarjuk kiírni, előltesztelő ciklust használunk. A ciklus ismétlési feltételéhez már szükségünk van egy sorra, ezért az első sort még a ciklus előtt beolvassuk:

```
Temp = Keret.ReadLine()  
Do Until Temp = "***"  
    WebLap.WriteLine(Temp)  
    Temp = Keret.ReadLine()  
Loop
```

Ezt az úgynevezett előolvasási technikát gyakran alkalmazzuk a fájlok kezelésénél.

Ha elérkeztünk a három csillaghoz, akkor beillesztjük a Szoveg.txt állomány sorait. Minden sor után egy soremelés-objektumot helyezünk el:

```
Do While Not Szoveg.AtEndOfStream  
    Temp = Szoveg.ReadLine()  
    WebLap.WriteLine(Temp & "<BR>")  
Loop
```

Végül hozzáírjuk a Keret.htm fájl maradék részét:

```
Do While Not Keret.AtEndOfStream  
    Temp = Keret.ReadLine()  
    WebLap.WriteLine(Temp)  
Loop
```

A szkriptben már csak a fájlok lezárása, az objektumok felszabadítása és a felhasználó tájékoztatása van hátra. A Szoveg.txt állomány változtatásával programunk tetszőleges szöveget megjelenítő weblapot összeállíthat.

## Bináris fájlok feldolgozása

Az *OpenTextFile* metódussal megnyitott állományoknak nem kell feltétlenül szöveget tartalmazniuk, bármilyen bináris kódot feldolgozhatunk. Ha az Olvasó nem érdeklődik a bináris fájlok kezelése iránt, akkor nyugodtan kihagyhatja ezt a részt.

A bájtok beolvasását egyesével végezzük:

```
Fajl.Read(1)
```

Vegyük figyelembe, hogy a VBScript a beolvasott értéket karakterkódként értelmezi, és a megfelelő karaktert adja vissza. Ha módosítani akarjuk a bájtot, akkor az *Asc* függvénnyel váltsuk át a kód numerikus értékére:

```
Temp = Asc(Fajl.Read(1))
```

A *Temp* egy 0 és 255 közötti egész szám lesz, decimálisan ez a számtartomány tárolható egyetlen bájtban.

A módosítás után a kód helyett a karaktert kell megadnunk a *Write* metódusnak, amit a *Chr* függvénnyel határozzunk meg:

```
Fajl.Write(Chr(Temp))
```

Így bármilyen bináris fájlt feldolgozhatunk, melynek tudjuk értelmezni a kódját.

A **10–17. példában** egy képfájlt módosítunk. Elkészítjük a kép sötétebb, világosabb és negatív változatát. A szkript a legegyszerűbb kódolású, BMP kiterjesztésű, 24 bites színmélységű fájlokat kezeli. Nem akarjuk az Olvasót terhelni a képfájlok kódolásának részleteivel, ezért az algoritmust csak vázlatosan ismertetjük.

A 24-bites színmélységű BMP fájlok minden egyes képpont színét 3 bájtban tárolják a kék, a zöld és a vörös komponenseknek megfelelően. A fájl elején egy úgynevezett fejléccet találunk. Az első két bájtot a BM (bitmap) karaktereket tartalmazza. Programunk ellenőrzi ezt a két bájtot, más fájl típus esetén hibaüzenettel leáll.

A fájl 11.–14. bájta hexadecimálisan megadja a fejléc hosszát. Ennyi bájtot után kezdődik a képpontok kódja. Kiszámítjuk, és az *Offset* változóban tároljuk ezt az értéket. A 24-bites színmélységű képek esetén a fájl 29. bájtyának értéke 24, a következő bájtot pedig 0. Ezt is ellenőrizzük, mert a programunk csak ekkor működik jól.

Mivel az állományok feldolgozása viszonylag lassú, folyamatjelzőt használunk. A léptetés gyakoriságának meghatározásához megállapítjuk a képfájl méretét. A folyamatjelzőt nem minden bájtra léptetjük, mert változtatása lassítja a program futását.

Ezzel az előkészítést elvégeztük. A módosított fájlokba át kell másolni az eredeti állomány fejlécét. Ezért lezárjuk, majd újra megnyitjuk a fájlt. Így állunk vissza az elejére. Ezután az új fájlokba átmásolunk *Offset* számú bájtot.

A képpontok RGB-kódját egyesével olvassuk be, és módosítva írjuk ki. Mivel a legfényesebb érték 255, a világosításhoz például megnöveljük a kódot a 255-ig hátralévő rész felével:

```
Temp = Asc(Be.Read(1))
Vilagos.Write(Chr((255 + Temp) \ 2))
```

A sötétítéshez megfelezzük a kód értékét:

```
Sotet.Write(Chr(Temp \ 2))
```

A negatív képet úgy kapjuk meg, hogy 255-ből kivonjuk az eredeti kódokat:

```
Negativ.Write(Chr(255 - Temp))
```

A programban ezután módosítjuk a folyamatjelzőt, majd az összes bájt feldolgozása után lezárjuk a fájlokat. A szkript kipróbálásához a CD-melléklet Fejezet10 mappájában található Nőszírom.bmp fájlt használhatjuk.

A bemutatott egyszerű transzformációk helyett bármilyen más műveletet is alkalmazhatunk, csak arra kell figyelni, hogy az eredmény egy 0 és 255 közé eső szám legyen.

### 10.3. Nyomtatás

Bár a nyomtatás nem tartozik közvetlenül a fájlkezeléshez, mégis ebben a fejezetben tárgyaljuk. A dokumentumok közvetlen nyomtatása helyett ugyanis gyakran készítünk szöveges állományokat, melyeket más programokkal is kinyomtathatunk.

#### Nyomtatás a szkriptekből

A weblapok nyomtatását általában a böngésző segítségével végezzük. Az Internet Explorer 5.5-ös változatától kezdve a szkriptekben meghívhatjuk a *window*-objektum *print* metódusát, amely egyenértékű a böngésző Fájl/Nyomtatás parancsával:

```
window.print()
```

A metódus végrehajtásának következtében megjelenik a képernyőn a Nyomtatás ablak, amelyben megadhatjuk a nyomtatási tartományt és a példányszámot, illetve elvégezhetjük a nyomtató kiválasztását, majd a nyomtatást. A *print* használatát a **10–18. példa** mutatja be.

A nyomtatáshoz az *onbeforeprint* (a nyomtatás előtt) és az *onafterprint* (a nyomtatás után) esemény kapcsolódik. Az *onbeforeprint* akkor következik be, amikor meghívódik a *print* metódus. Az *onafterprint* esemény pedig akkor jön létre, amikor az operációs rendszer átvette a szkripttől a nyomtatandó dokumentumot. Felhívjuk a figyelmet arra, hogy a Windows esetén a dokumentumok egy úgynevezett nyomtatási sorba kerülnek, – a nyomtató kezelését már az operációs rendszer végzi. Akár a nyomtatási sorban, akár a nyomtatónál keletkezik hiba (például kifogy a papír), az üzenetet már nem a programunk, hanem az operációs rendszer fogadja, és jelzi a felhasználónak. Így az *onafterprint* esemény bekövetkezése nem a nyomtatás tényleges végrehajtását jelenti!

A nyomtatás folyamata a következő lépésekből áll:

1. Meghívjuk a *print* metódust.
2. Végrehajtódik az *onbeforeprint* eseménykezelője.
3. Megkezdődik a *print* metódus végrehajtása.
4. A *print* metódus elküldi az operációs rendszernek a nyomtatási parancsot.



5. Befejeződik a *print* metódus végrehajtása.
6. Lefut az *onafterprint* eseménykezelője.
7. Az operációs rendszer megjeleníti a Nyomtatás ablakot.
8. A beállítások elvégzése és az OK gombra való kattintás után az operációs rendszer végrehajtja a nyomtatást.

Felhívjuk a figyelmet arra, hogy a Nyomtatás ablak csak az *onafterprint* esemény után jelenik meg a képernyőn. A végrehajtást és a bekövetkező eseményeket a **10–19. példa** segítségével követhetjük nyomon.

A nyomtatással kapcsolatos eseményeket főleg arra használjuk, hogy megváltoztassuk a lap tulajdonságait a nyomtatásnál. A **10–20. példában** az *onbeforeprint* eseménykezelőjében a nyomtatás idejére átírjuk a weblap címét, és letiltjuk a háttérkép, illetve a parancsgomb megjelenítését. Az *onafterprint* eseménykezelőjében visszaállítjuk az eredeti értékeket.

Megjegyezzük, hogy sem az *onbeforeprint*, sem az *onafterprint* nem vesz részt az eseménybuborék terjedésében, és nem szakítható meg a végrehajtásuk. Az események akkor is bekövetkeznek, ha a felhasználó a menüből adja ki a nyomtatási parancsot. Ha a dokumentum rendelkezik a megfelelő eseménykezelő eljárásokkal, akkor ezek szintén végrehajthatók.

## A lapdobás beállítása

A weblapok gyakran más tördelést igényelnek a nyomtatásnál, mint a képernyőn történő megjelenítésnél. A felhasználó megváltoztathatja a böngészőben a betűméretet (Nézet/Szövegméret), így teljesen elronthatja az objektumok elrendezését. Ennek elkerülése érdekében általában megadjuk a *font-size* stíluselem értékét (betűméret pixelben vagy tipográfiai pontban, pt mértékegységgel).

Az is megeshet, hogy egy kép egyik fele a lap aljára, másik fele már a következő lap tetejére kerül, vagy máshol következik be helytelen tördelés. A lapdobást a *page-break-before* (lapdobás előtte) illetve a *page-break-after* (lapdobás utána) stíluselemekkel állíthatjuk be, melyek *always* értéke esetén az adott objektum előtt, illetve mögött új lap kezdődik a nyomtatásnál.

A stíluselemek használatát a **10–21. példa** mutatja be. A dokumentum által megjelenített versszakokat bekezdésobjektumokba tettük. A nyitó tagokban lapdobást írtunk elő az egyes versszakok után:

```
<P style = "page-break-after: always">
```

A tulajdonság értékét a szkriptekben is megadhatjuk. Figyeljünk az eltérő szintaxisra (**10–22. példa**):

```
Sub window_onload
  Dim Bekezdés
  For Each Bekezdés In document.all.tags("P")
    Bekezdés.style.pageBreakAfter = "always"
  Next
End Sub
```

## A nyomtatási formátum megadása

A weblapok nyomtatásánál gyakran okoz gondot, hogy a sötét háttér sok festéket használ el, és megnehezíti a szöveg olvasását csakúgy, mint az erőteljes mintázat.

Az Internet Explorer Eszközök/Internetbeállítások/Speciális menüjében a Nyomtatás csoportban lehetőségünk van a háttérszínek és háttérképek nyomtatásának letiltására. Ha nem akarjuk a felhasználót terhelni a beállítások megváltoztatásával (egyébként is általában csak a nyomtatás után jut eszébe), akkor a *media* stíluselem segítségével külön formátumot írhatunk elő a képernyőn történő megjelenítéshez, és külön formátumot a nyomtatáshoz. A globális stílusok részletes tárgyalására nincs módunk, így ezt a lehetőséget csak egy példán keresztül mutatjuk be. A **10–23. példa** weblapja a háttér letiltása nélkül nyomtatva nagyon rosszul olvasható, és rengeteg festéket elhasznál. A **10–24. példában** a HEAD-ben elhelyeztük a következő kódrészletet:

```
<STYLE media = "screen">
  BODY {background-color: blue; color: red}
</STYLE>
<STYLE media = "print">
  BODY {background-color: white; color: black}
  INPUT {visibility: hidden}
</STYLE>
```

Ezzel a képernyőn (*screen*) kék hátteret és vörös betűszínt, a nyomtatásnál (*print*) viszont fehér hátteret és fekete betűszínt írtunk elő. Az elrejtésével letiltottuk a parancsgomb nyomtatását is. Más objektumok megjelenítését hasonló módon szabályozhatjuk. A globális stílusok használatáról a Microsoft Development Network weblapján találunk részletes tájékoztatást.

## Táblázatok nyomtatása

A táblázatok nyomtatásánál figyelniünk kell arra, hogy a fejléc minden oldalon megjelenjen. Célszerű külön állományt készíteni akár HTML, akár szövegfájl formátumban. A HTML-fájlban a *page-break-before* stíluselemet, szövegfájl esetén pedig a *Chr(12)* karaktert<sup>30</sup> használjuk a lapdobás létrehozására. Szövegfájloknál a formázást a 3.4. fejezetben ismertetett *Formaz* függvénnyel végezhetjük el. A kitöltő karakterként alkalmazott nem törhető szóköz (*&nbsp;*) helyett azonban egyszerű szóközt használunk. A módosított függvényt a *FormazSpace.vbs* állomány tartalmazza, amit a CD-melléklet Fejezet10-es mappájában találunk. Ezt kell csatolni a HTML-kódhoz.

A **10–25. példa** egy hosszú táblázat nyomtatásra alkalmas formában történő elkészítését szemlélteti. A táblázattal négy függvény értékeit állítjuk elő a megadott tartományban és lépésközzel. A weblapon választhatunk a HTML- illetve a szövegfájl között. A program az elkészítés után meg is nyitja a megfelelő állományt, amit a Fájl menü Nyomtatás parancsával nyomtathatunk ki.

Az algoritmust csak vázlatosan ismertetjük. Az Olvasó a forráskódban elhelyezett megjegyzések alapján nyomonkövetheti a módszert.

---

<sup>30</sup> A kód függ a nyomtatótól, de a legtöbb esetben megfelelő ez az érték.

A globális szkript elején deklaráljuk a sorok számát egy lapon a HTML-, illetve szövegfájl formátumnak megfelelően. Szükség esetén így elegendő csak itt elvégezni a forráskód módosítását. Ugyanez vonatkozik a lapdobás kódjának értékére is.

Az *Elkeszit* eljárásban némi hibaellenőrzést végzünk (a négyzetgyök és a logaritmus miatt csak pozitív kezdőértéket engedünk meg). A bejelölt típusnak megfelelően összeállítjuk a fájl elérési útját és kiterjesztését, majd megszámloljuk, hogy hány függvény értékeit kell megjelenítenünk. Erre a kivitel formázásánál lesz szükség.

Ezután megnyitjuk az állományt, és meghívjuk a formátumnak megfelelő eljárást. Az eljárások készítik el a szükséges táblázatot. A végén lezárjuk a fájlt, majd felszabadítjuk a változókat.

A weblapot a *Weblap* eljárás hozza létre. Miután összeállította a HTML-kód fejrészét, és megnyitotta a BODY-t, a *HtmlFejlec* eljárás hívásával elkészíti az első táblázat fejlécét. Ekkor még nem kell új lapot kezdeni, ezért az *UjOldal* sztring üres. Tartalmát csak az első fejléc kiírása után módosítjuk a *page-break-before* stíluselem értékadásával. A kiírt sorok számát a *Sor* változó tartja nyilván. Értékétől függően kezdünk új táblázatot a lapdobás után.

A táblázatot készítő ciklusban ellenőrizzük a sorok számát, majd elkészítünk egy sort. A függvények nevét a kezdő zárójellel együtt egy tömbben tároljuk, így az *Execute* utasítás segítségével ciklusban végezzük el a függvényértékek kiszámítását. Az *Execute* számára összeállítjuk a függvényhívást végző utasítást tartalmazó sztringet. Mivel az interpreter a sztringek összefűzésénél a függvény argumentumában a tizedespontot tizedesvesszővel helyettesítené, ezért az *Execute* végrehajtása előtt a *Replace* függvénnyel kicseréljük az esetleges tizedesvesszőt tizedespontra.

A *Sor* változó értékét minden sor kiírása után eggyel megnöveljük. Miután elkészült a táblázat, lezárjuk a HTML-kódot, és betöltjük a weblapot a böngészőbe.

A szövegfájlt hasonló algoritmussal készítjük el. A formázást a már említett *Formaz* függvénnyel végezzük. Az állományt a Microsoft Office Word programjával nyitjuk meg. Vegyük észre, hogy a Word felismeri a lapdobást (oldaltörést)! A Word hiányában használhatjuk a Jegyzettömböt (Notepad.exe) is, de a lapdobást nem minden nyomtató hajtja végre.

Az elkészült táblázatok még lehetne szépíteni, a programot bővíteni. Gondoskodhatunk a lapok számozásáról, az adatbevitel alaposabb ellenőrzéséről. Ezeket a kiegészítéseket az Olvasóra bízuk.



# 11. ALGORITMUSOK

Az előző fejezetekben főleg a dokumentum objektummodell objektumaival és a VBScript utasításaival ismerkedtünk meg. Példáinkban ezeknek az eszközöknek a használatát mutattuk be. A könyv utolsó fejezetében áttekintést adunk a programozás során előforduló leggyakoribb algoritmusokról. A kezdő programozó ezt a részt egyfajta lexikonnak tekintheti, amelyben megkeresheti a számára szükséges eljárást. Kitérünk néhány nehezebb feladat megoldására is, melynek algoritmusát a tapasztaltabb programozók hasznosíthatják.

Az algoritmus fogalmával az 1. fejezetben már megismerkedtünk.

Algoritmus: egy feladat megoldásához vezető utasítások sorozata.

Az algoritmustól elvárjuk, hogy véges számú lépésből álljon. Általában az algoritmust végrehajtó program futása is befejeződik véges számú utasítás végrehajtása után. Vannak azonban olyan esetek, amikor a működés a cél, nem pedig a befejezés. Egy számítógép operációs rendszere a bekapcsolástól a kikapcsolásig fut. A Pioneer-11 űrszondát vezérlő algoritmus 30 éven át funkcionált, amíg az űreszköz bejárta a Naprendszer távoli bolygóit.

Az algoritmusnak minden egyes lépése egyértelműen végrehajtható. Ki kell tudnunk választani a következő lépést. Szigorúbb értelemben elvárjuk az algoritmustól, hogy bizonyíthatóan elvezessen a feladat megoldásához. A programhelyesség bizonyításával könyvünkben nem foglalkozunk.

Nagyon sokféle algoritmus létezik. Minden számítógépes program egy-egy összetett algoritmusnak tekinthető. Ezek az összetett algoritmusok felbonthatók elemi tevékenységek sorozatára. Ebben fejezetben azokat az elemi algoritmusokat gyűjtöttük össze, melyek sorozatokhoz rendelnek valamilyen értéket vagy másik sorozatot.

A sorozatok elemei tömbökben tárolhatók. Az algoritmusokban egydimenziós tömböket használunk, de tetszőleges típusú adatra alkalmazhatóak. Mivel általánosan szeretnénk megadni a feladatok megoldásának a menetét, a tömbök indexelését 1-gyel kezdjük, és nem használjuk ki a VBScriptben rendelkezésünkre álló 0 indexű elemet. A tömb elemeinek a számát, tehát a legnagyobb index értékét általában N-nel jelöljük.

Az algoritmusokat mondatszerű leírással ismertetjük, utasításaikat eljárásokba foglaljuk. Az eljárásokat tetszőleges programnyelven kódolhatjuk.

Az értékadó utasításokban szereplő egyenlőségjel helyett a pszeudokódban szokásos `:=` (legyen egyenlő) definiáló egyenlőségjelet használjuk:

*Változónév := kifejezés*

A VBScripttel szemben több programnyelv ezt a jelölést alkalmazza.

Az algoritmusokat az Algoritmusok.htm fájl foglalja össze, amely a CD-melléklet Dokumentumok mappájában található. Minden egyes algoritmust VBScript példával is szemléltetünk. Ezekben a tömböket általában véletlen egész számokkal töltjük fel. A függelék feladataiban további alkalmazásokat találunk.

## 11.1. Érték hozzárendelése sorozathoz

A bemutatásra kerülő az algoritmusokban egy sorozat elemeit vizsgáljuk. A vizsgálat eredménye egy szám, például az elemek összege, vagy valamilyen index, például a keresett elem indexe lesz. A sorozathoz tehát egy értéket rendelünk.

### Összegezés

Az összegezés algoritmusát már az. 5. fejezetben felhasználtuk. Most általánosan is megfogalmazzuk. Adott egy N-elemű tömb, határozzuk meg az elemek összegét!

Az eljárásban az *Összeg* nevű változót nullázzuk, majd egy ciklusban hozzáadjuk a tömb összes elemét:

```
Eljárás
  Összeg := 0
  Ciklus I := 1-től N-ig
    Összeg := Összeg + Tömb(I)
  Ciklus vége
  Ki: Összeg
Eljárás vége
```

Az algoritmust a **11–1. példa** mutatja be.

### Eldöntési algoritmus

A programokban gyakran meg kell vizsgálnunk egy tömb elemeinek tulajdonságait. Ezzel kapcsolatban több kérdést felvethetünk. Az eldöntési algoritmus azt határozza meg, hogy van-e legalább egy adott tulajdonságú elem a tömbben. Az elemszámot itt is N-nel jelöljük.

A tömb elemeit egy ciklussal addig vizsgáljuk, amíg nem találunk megfelelő tulajdonságú elemet, vagy végig nem érünk rajtuk. A kilépés után ellenőrizzük, hogy miért ért véget a ciklus. Ha az utoljára megvizsgált elem rendelkezik a keresett tulajdonsággal, akkor a *Van* logikai változó értékét igazra állítjuk. Egyébként végigértünk a tömbön, de nem találtunk megfelelő elemet:

```
Eljárás
  I := 1
  Ciklus amíg Tömb(I) nem adott tulajdonságú És I < N
    I := I + 1
  Ciklus vége
  Van := Hamis
  Ha Tömb(I) adott tulajdonságú Akkor
    Van := Igaz
  Elágazás vége
  Ki: Van
Eljárás vége
```

Ezt az algoritmust az 5. fejezetben már használtuk. A **11–2. példa** a tömbelemek 7-tel való oszthatóságának vizsgálatánál mutatja be az alkalmazását.

## Lineáris keresés

A keresési feladatokban nem csak arra vagyunk kíváncsiak, hogy létezik-e az adott tulajdonságú tömbelem, hanem az indexét is szeretnénk meghatározni.

Vegyük észre, hogy ha találtunk adott tulajdonságú elemet az előző algoritmusban, akkor az I értéke megadja annak indexét:

```
Van = Hamis
Ha Tömb(I) adott tulajdonságú Akkor
    Van := Igaz
    KeresettIndex = I
Elágazás vége
```

Ha több ilyen elem létezik, akkor algoritmusunk az elsőt keresi meg. Mivel itt sorra egymás után vizsgáljuk a tömb elemeit, a módszert lineáris keresésnek hívjuk. Bemutatásához a 11–2. példában kiírjuk az elem sorszámát is.

Ha biztosan tudjuk, hogy a tömbnek van a vizsgált tulajdonsággal rendelkező eleme, akkor egyszerűsíthetjük a ciklus ismétlési feltételét, hiszen nem kell figyelnünk, hogy végigértünk-e a tömbön:

```
Ciklus amíg Tömb(I) nem adott tulajdonságú
    I := I + 1
Ciklus vége
```

A kilépés után az I megadja a keresett elem indexét.

Az 5. fejezetben bemutattuk az ütköző alkalmazását a feltétel egyszerűsítéséhez. Ha van rá lehetőség, akkor a tömb végéhez hozzáfűzünk egy keresett tulajdonságú elemet. Ha a keresés során ezt az elemet találjuk meg, akkor az eredeti tömbelemek nem rendelkeznek az adott tulajdonsággal:

```
Van := Igaz
Ha I = a legutolsó elem (ütköző) indexe Akkor
    Van := Hamis
Elágazás vége
```

**A 11–3. példában** ütköző segítségével keresünk 10-zel osztható tömbelemet. Ezért a tömb végéhez illesztünk egy újabb elemet, melynek értéke 10. Erre biztosan rátalálunk, így a ciklusfejben csak egy feltételt kell megvizsgálni. Ez csökkenti a futási időt, és egyszerűbbé teszi a programot.

## Megszámlálás

A megszámlálási algoritmus a keresés mellett azt is meghatározza, hogy hány tömbelem rendelkezik a vizsgált tulajdonsággal.

A számláláshoz bevezetünk egy *Darab* nevű változót, melynek értékét minden egyes találat esetén megnöveljük:

```
Eljárás
  Darab := 0
  Ciklus I := 1-től N-ig
    Ha Tömb(I) adott tulajdonságú Akkor
      Darab := Darab + 1
  Elágazás vége
Ciklus vége
Ki: Darab
Eljárás vége
```

A módszer alkalmazását a **11–4. példa** mutatja be.

## Maximum kiválasztása

A tömb elemeit most valamilyen reláció szerint össze lehet hasonlítani egymással. Keressük a legnagyobb elemet és annak sorszámát a tömbben.

Az eljárás során először az első elemet vesszük legnagyobbnak, majd egy ciklus segítségével összehasonlítjuk a többi elemmel. Ha találtunk nagyobbat, akkor ezzel helyettesítjük a legnagyobb elemet és annak helyét is:

```
Eljárás
  LegnagyobbElem := Tömb(1)
  Hely := 1
  Ciklus I := 2-től N-ig
    Ha Tömb(I) > LegnagyobbElem Akkor
      LegnagyobbElem := Tömb(I)
      Hely := I
  Elágazás vége
Ciklus vége
Ki: LegnagyobbElem, Hely
Eljárás vége
```

A **11–5. példa** az algoritmus közvetlen alkalmazásával meghatározza egy tömb legnagyobb elemét és az elem sorszámát.

Vegyük észre, hogy a legkisebb elemet ugyanilyen módon kereshetjük meg, csak a szelekció feltételét kell módosítani:

```
Ha Tömb(I) < LegkisebbElem Akkor
  LegKisebbElem := Tömb(I)
```



## Maximum-kiválasztás néhány változó esetén

A tárgyalt algoritmusoknál kihasználtuk, hogy a tömbök elemeit ciklusokkal kezelhetjük. Ha a változók nem tömbökben helyezkednek el, akkor nem tudunk ciklusváltozót alkalmazni az indexeléshez. A ciklussal történő elemkiválasztás elve azonban ekkor is felhasználható. Ciklus helyett az összes változóra ki kell írni az összehasonlítást és a cserét.

Válasszuk ki például három változó közül a legnagyobbat. Először az első változó értékét tekintjük maximálisnak:

```
LegnagyobbElem := ElsőVáltozó
```

majd a *LegnagyobbElem*-et egymás után összehasonlítjuk a többi változó értékével. Ha nagyobbat találunk, akkor elvégezzük a cserét:

```
Ha LegnagyobbElem < MásodikVáltozó Akkor
    LegnagyobbElem := MásodikVáltozó
Elágazás vége
Ha LegnagyobbElem < HarmadikVáltozó Akkor
    LegnagyobbElem := HarmadikVáltozó
Elágazás vége
```

Több változó esetén mindenképpen érdemes tömböt használni.

## Tömbelem keresése rendezett tömbben

A keresési algoritmusokban gyakran egy adott értékű tömbelemet keresünk. Ha az elemek egymással összehasonlíthatóak, és nagyság szerint rendezve helyezkednek el, akkor egy tömbelem keresését gyorsítani tudjuk. A továbbiakban feltételezzük, hogy az elemek növekvő sorrendben következnek egymás után.

A lineáris keresés során elegendő addig elmenni, amíg a keresett elemnél kisebb elemeket találunk. Ha már nagyobb elem következik, akkor nem szerepel a keresett elem a tömbben:

```
Hely := 1
Ciklus amíg Tömb(Hely) < keresett érték És Hely < N
    Hely := Hely + 1
Ciklus vége
```

Ezt az elvet használtuk fel az 5–40. példában. Ütköző alkalmazásával ez a módszer is gyorsítható.

## Bináris keresés

A lineárisnál lényegesen hatékonyabb algoritmus az úgynevezett bináris keresés. Ez először a tömb középső elemét hasonlítja össze a keresett értékkel (ha páros számú elem van, akkor a maximális index felét vesszük). Ha a középső elem megegyezik a keresett értékkel, akkor befejeztük a keresést. Ha a középső elem nagyobb, mint a keresett érték, akkor a tömb első felében folytatjuk a keresést, egyébként pedig a második felében. A maradékra ugyanezt az eljárást ismételtetjük, így a tartomány mérete egyre csökken. Ha már csak egyetlen elem maradt, akkor befejeződik az eljárás. A megmaradt elem vizsgálatával eldönthető, hogy megtaláltuk-e a keresett értéket.

Az algoritmus végrehajtásához feljegyezzük a vizsgált tartomány első és utolsó elemét. Ezek kezdetben megegyeznek a tömb két szélső elemével. A találat figyeléséhez egy logikai változót alkalmazunk:

```
Eljárás
  Első := 1, Utolsó := N
  Talált := Hamis
```

Ezután egy ciklus kezdődik, melyben kiszámítjuk a tartomány középső elemének indexét, majd ezt az elemet összehasonlítjuk a keresett értékkel. Ha megegyeznek, akkor rátaláltunk a keresett elemre:

```
Ciklus
  Középső := Int((Első + Utolsó) / 2)
  Ha Tömb(Középső) = KeresettElem Akkor
    Talált := Igaz
```

Ha a középső elem kisebb, akkor a tartomány második felében kell folytatni a keresést, a középső után következő elemet írjuk át elsőre (a középsőt már megvizsgáltuk):

```
Egyébként ha Tömb(Középső) < KeresettElem Akkor
  Első := Középső + 1
```

Ha a középső elem nagyobb, akkor a keresett elem a tartomány első felében helyezkedik el, a középső előtt lévő elemet írjuk át utolsóra:

```
Egyébként ha Tömb(Középső) > KeresettElem Akkor
  Utolsó := Középső - 1
Elágazás vége
```

A ciklust addig kell ismételni, amíg nem találtunk rá a keresett elemre, és nem ért össze a tartomány két határa:

```
amíg Talált = hamis És Első <= Utolsó
Ciklus vége
```

A *Talált* logikai változó megmutatja, hogy rátaláltunk-e a keresett elemre:

```
Ki: Talált
```

Ha igen, akkor a helyét éppen a középső index utoljára beállított értéke adja meg:

```
Ha Talált Akkor
  Hely := Középső
  Ki: Hely
Elágazás vége
Eljárás vége
```

**A 11–6. példában** bináris kereséssel határozzuk meg egy tömbelem helyét.

A bináris keresés elnevezésnek az a magyarázata, hogy egy-egy lépésben feleződik a tartomány mérete. Így 100 elem esetén legfeljebb 7 lépésben megtaláljuk a keresett elemet. Ha ez az utolsó elem, akkor a lineáris keresésnél 100 iterációra van szükség! A bináris keresést gyakran logaritmikus keresésnek is nevezik, mert az ismétlések száma az elemek számának logaritmusával arányos.

## 11.2. Rendezési algoritmusok

A programozás során sokszor van szükség az elemek rendezésére. A rendezési algoritmusokkal külön szakirodalom foglalkozik, melyben tárgyalják az egyes módszerek hatékonyságát, gyorsaságát, memóriaigényét. Ezek az igények gyakran ellentmondanak egymásnak.

A végrehajtási idő sem adható meg egyértelműen. Más algoritmus bizonyul gyorsnak, ha az elemek teljesen véletlenszerűen helyezkednek el, és megint más, ha csak kismértékben rendezetlenek. Ha tudjuk például, hogy rövidebb-hosszabb sorozatokat már rendezve találunk, akkor ezt kihasználhatjuk az algoritmus hatékonyságának a növelésénél.

Az elemek rendezését elvégezhetjük beolvasás közben, vagy egy másik tömbbe való áthelyezéssel is. A továbbiakban egy már létező tömböt rendezünk, másik tömb felhasználása nélkül. Bár az elemek összehasonlításához sokféle reláció alkalmazható, példáinkban egész számokat fogunk rendezni növekvő sorrendbe. Szükség esetén a relációk megfordításával csökkenő sorrendet kapunk.

Megjegyezzük, hogy a nagyméretű fájlokban elhelyezkedő adatok rendezése speciális algoritmusok alkalmazását igényli.

### Minimumkiválasztásos rendezés

A legegyszerűbb rendezésnél a tömb elemeit sorra összehasonlítjuk a legelső elemmel. Ha kisebbet találunk, akkor felcseréljük a két értéket. Mire végigérünk a tömbön, az első helyre a legkisebb elem kerül. A folytatásban az egyre magasabb indexű elemek kerülnek a helyükre. Ez a módszer azonban nagyon sok fölösleges cserével jár.

Nagymértékben csökken a cserék száma, ha először megkeressük a legkisebb elemet, és csak utána tesszük az első helyre. Ennek a minimumkiválasztásos rendezésnek az algoritmusát az 5–7. példában már bemutattuk. Itt a teljesség kedvéért ismét visszatérünk rá. Az 5. fejezetben használt algoritmust annyiban módosítjuk, hogy a tömb *I*-edik elemét félretesszük a *LegkisebbElem* változóba. Az interpreter egy változó értékét gyorsabban elő tudja venni a memóriából, mint egy tömbelemét:

```
Eljárás
  Ciklus I := 1-től N-1-ig
    LegkisebbElem := Tömb(I)
    Hely := I
    Ciklus J := I+1-től N-ig
      Ha Tömb(J) < LegkisebbElem Akkor
        LegkisebbElem := Tömb(J)
        Hely := J
    Elágazás vége
  Ciklus vége
  Tömb(Hely) := Tömb(I)
  Tömb(I) := LegkisebbElem
Ciklus vége
Eljárás vége
```

A módosított eljárást a **11–7. példa** szemlélteti.

## Buborékos rendezés

A buborékos rendezés során egymás után összehasonlítjuk a tömb szomszédos elemeit, és ha a nagyobb volt elől, akkor fölcseréljük a két elemet. Ezzel a legnagyobb elem a tömb végére kerül, felszáll, mint egy buborék a vízben. Az eljárást ismételve rendezett tömböt kapunk:

```
Eljárás
  Ciklus I := 1-től N-1-ig
    Ciklus J := 2-től N-ig
      Ha Tömb(J-1) > Tömb(J) Akkor
        Tömb(J-1) és Tömb(J) cseréje
      Elágazás vége
    Ciklus vége
  Ciklus vége
Eljárás vége
```

Az algoritmus alkalmazását a **11–8. példa** mutatja be.

Az ismétlések számát csökkenthetjük, ha kihasználjuk, hogy az első lépésben a legnagyobb elem a tömb végére kerül. Így a következő ismétlés során elegendő eggyel kevesebb elemmel foglalkozni. Tehát minden egyes ciklusban csökkenthetjük az összehasonlításra kerülő elemek számát.

Az eljárást addig ismételjük, amíg van közben csere. Ha már nem hajtottunk végre cserét, akkor rendezett a tömb. A cseréket egy logikai változóval figyeljük. Értékét a külső ciklus elején hamisnak választjuk, és csere esetén átírjuk igazra. Ha a ciklus végrehajtása után hamis marad, akkor az eljárás befejeződött:

```
Eljárás
  I := N
  Ciklus
    VoltCsere := hamis
    Ciklus J := 2-től I-ig
      Ha Tömb(J-1) > Tömb(J) Akkor
        Tömb(J-1) és Tömb(J) cseréje
        VoltCsere := igaz
      Elágazás vége
    Ciklus vége
    I := I - 1
  amíg VoltCsere = igaz
  Ciklus vége
Eljárás vége
```

Ha minden ciklusban volt csere, és az *I* lecsökken 1-re, akkor az előtesztelő belső ciklus nem fut le, tehát a *VoltCsere* hamis marad. Ezért nem szükséges minden lépésben megvizsgálnunk, hogy az *I* értéke nem lépett-e ki a megengedett tartományból.

Az algoritmust a **11–9. példa** valósítja meg. A **11–10. példa** a belső ciklusban is kiírja a tömb elemeit, a felcserélt változókat és a *VoltCsere* változó értékét. A tömbelemek számát lecsökkentettük, hogy ne legyen túl hosszadalmas a rendezés.

## Rendezés beillesztéssel

A beillesztéses módszer a kezünkben tartott kártyalapok rendezéséhez hasonlít. A második lapot összehasonlítjuk az elsővel, és szükség esetén berakjuk eléje. A harmadik lapot szükség esetén berakjuk a második vagy az első elé és így tovább.

Az algoritmus megfogalmazásához tételezzük fel, hogy a tömb első néhány eleme már rendezett. A következő elemet félretesszük, így felszabadítjuk a helyét. A félretett elemet összehasonlítjuk az előtte álló, rendezett elemekkel. Minden nála nagyobb elemet eggyel följebb léptetünk. Az így keletkező üres helyre pedig beírjuk a félretett értéket. Ha ezt az eljárást a másodiktól kezdve minden elemmel elvégezzük, akkor rendezett tömböt kapunk:

```
Eljárás
  Ciklus I := 2-től N-ig
    Temp := Tömb(I)
    J := I - 1
    Ciklus amíg J > 0 És Tömb(J) > Temp
      Tömb(J + 1) := Tömb(J)
      J := J - 1
    Ciklus vége
    Tömb(J + 1) := Temp
  Ciklus vége
Eljárás vége
```

A belső ciklus ismétlési feltételének kódolásánál figyelembe kell vennünk, hogy  $J = 0$  esetén a  $Tömb(J)$  olyan elemre hivatkozik, melynek nem adtunk értéket. Azokban a programnyelvekben, melyekben a tömbök indexelése 1-gyel kezdődik, nem is létezik ilyen elem. Így a logikai kifejezés kiértékelése futási hibához vezet.<sup>31</sup> Ezért helyette egy függvénnyel határozzuk meg az ismétlési feltétel logikai értékét, melynek átadjuk az aktuális indexet és a félretett elemet:

```
Ciklus amíg RosszHely(J, Temp)
```

A függvény definíciójában először megvizsgáljuk az indexet. Ha egynél kisebb, akkor nem kell megismételni a ciklust. A visszatérési érték ekkor megegyezik az  $Index > 0$  reláció értékével:

```
Függvény RosszHely(Index, Elem)
  RosszHely := (Index > 0)
```

Ha az index nagyobb mint 0, akkor a tömbelemet összehasonlítjuk a félretett értékkel. A visszatérési érték pedig éppen a  $Tömb(Index) > Elem$  reláció értéke lesz:

```
Ha RosszHely Akkor
  RosszHely := (Tömb(Index) > Elem)
Elágazás vége
Függvény vége
```

Az algoritmus működését a **11–11. példa** szemlélteti.

<sup>31</sup> A VBScriptben nincsen rövidzár, a logikai kifejezés minden tagja kiértékelésre kerül. Programnyelvtől független algoritmusban azonban nem szabad kihasználnunk az egyedi implementációk sajátosságait!

## Shell-rendezés

Egy hatékony rendezési algoritmus kidolgozása Donald Shell nevéhez fűződik. A Shell-módszerben a tömbnek egymástól adott lépésközzre eső elemeivel foglalkozunk, ezeket rendezzük. A rendezéshez az elemeket először az első elemtől kezdve kiválasztjuk az adott lépésközzel, majd a másodiktól, és így tovább, amíg el nem jutunk egy már előzőleg kiválasztott elemhez.

A lépésközt csökkentve ismételtetjük az eljárást. A folyamat addig tart, amíg a lépésköz le nem csökken 1-re. Ekkor már minden elemet rendeztünk.

A módszer hatékonyságának oka, hogy egyre több kis szám kerül a tömb elejére, és egyre több nagy érték a végére. Így egy nagyjából elvégzett rendezést kell egyre tovább finomítanunk. Ez viszonylag kevés cserével jár.

Lépésköznek tetszőleges számot választhatunk. A leggyorsabban akkor érünk célra, ha vesszük azt a 2 hatványt, amely még éppen kisebb, mint az elemek száma, és egyet csökkentjük. Ezt a következő kifejezéssel határozhatjuk meg:

Lépésköz :=  $2^{\text{Int}(\text{Log}(N)/\text{Log}(2))} - 1$

A lépésközt az eljárás során felezéssel csökkentjük.

A kiválasztott elemek rendezéséhez bármelyik módszert használhatjuk. Mi a beillesztéses rendezést választjuk. Ebben meghagyjuk az I és J indexeket, ezért a külső ciklus indexét K-val jelöljük.

A külső ciklus a lépésközt csökkenti:

Eljárás

Lépésköz :=  $2^{\text{Int}(\text{Log}(N)/\text{Log}(2))} - 1$

Ciklus amíg Lépésköz > 0

A következő ciklus megadja azt az elemet, amelytől kezdve az adott lépésközzel kiválasztott elemeket rendezzük:

K := 1

Ciklus amíg K <= Lépésköz

A belső ciklusok megegyeznek a beillesztéses rendezés ciklusaival, csak 1 helyett az aktuális lépésközzel történik a változók értékének módosítása:

Ciklus I := K + Lépésköz-től N-ig Lépésköz-zel növelve

Temp := Tömb(I)

J := I - Lépésköz

Ciklus amíg Ismétel(J, Temp)

Tömb(J + Lépésköz) := Tömb(J)

J := J - Lépésköz

Ciklus vége

Tömb(J + Lépésköz) := Temp

Ciklus vége

A kiválasztott elemek rendezését a következő elemtől kezdve is végrehajtjuk:

K := K + 1

Ciklus vége

majd a lépésközt megfelezve megismételjük az egész eljárást:

```
Lépésköz := Int(Lépésköz / 2)
Ciklus vége
Eljárás vége
```

A Shell-rendezést a **11–12. példában** láthatjuk.

### 11.3. Sorozatok hozzárendelése sorozatokhoz

Ezekben az algoritmusokban tömbökből készítünk újabb tömböket.

#### Kiválogatás

A legegyszerűbb feladatban egy tömbből válogatunk ki megadott tulajdonságú elemeket. A tömb minden egyes eleménél megnézzük, hogy rendelkezik-e a vizsgált tulajdonsággal. Ha igen, akkor átmásoljuk az új tömbbe.

Jelöléseinket megtartva a *Tömb* az eredeti tömböt jelöli, az *N* pedig az elemeinek a számát. Az új tömböt *ÚjTömb*-nek nevezzük, melynek elemeit *J*-vel indexeljük:

```
Eljárás
  J := 0
  Ciklus I := 1-től N-ig
    Ha Tömb(I) adott tulajdonságú Akkor
      J := J + 1
      ÚjTömb(J) := Tömb(I)
    Elágazás vége
  Ciklus vége
Eljárás vége
```

Az eljárás végrehajtása után a *J* az új tömb elemszámát, egyben az adott tulajdonsággal rendelkező elemek számát mutatja. Az algoritmus alkalmazását a **11–13. példa** szemlélteti. Az elemeket egy dinamikus tömbbe másoljuk, így nem kell előre ismerünk a méretét.

A kiválogatás algoritmusát könnyen módosíthatjuk, hogy az elemeket két tömbbe válogassa szét. Szétválogatásnál az egyik tömbbe az adott tulajdonsággal rendelkező elemek kerülnek, a másik tömbbe pedig a többi elem. A módosítást az Olvasóra bízunk. Nehezebb a megoldás, ha egy tömbön belül kell így szétválogatni az elemeket. Ezt a 11–8. feladatban mutatjuk be a függelékben.

#### Összefésülés

Az összefésülés két rendezett tömb elemeit egyesíti egy harmadik, szintén rendezett tömbben. Ehhez sorra vesszük a két tömb elemeit, és mindig a kisebbiket írjuk az új tömbbe. Ha valamelyik tömbnek elfogytak az elemei, akkor a másik tömb maradék elemeit is átmásoljuk az új tömbbe.

Az algoritmus leírásánál a *MaxIndex1* elemszámú *Tömb1* és a *MaxIndex2* elemszámú *Tömb2* elemeit fogjuk összefésülni a *Fésült* tömbbe. A *Tömb1* elemeit *I*-vel, a *Tömb2*-t *J*-vel, a *Fésült* tömböt pedig *K*-val indexeljük.

Először minden indexet a tömbök elejére állítunk:

```
Eljárás
  I := 1
  J := 1
  K := 1
```

Összehasonlítjuk a *Tömb1* és a *Tömb2* aktuális indexű elemét. A kisebbet beírjuk a *Fésült* tömb *K*-adik helyére, majd rátérünk a következő elemre. A folyamatot addig ismételjük, amíg végig nem értünk valamelyik tömbön. A *K* értékét minden egyes ismétlésnél megnöveljük:

```
Ciklus amíg I <= MaxIndex1 És J <= MaxIndex2
  Ha Tömb1(I) <= Tömb2(J) Akkor
    Fésült(K) := Tömb1(I)
    I := I + 1
  Egyébként
    Fésült(K) := Tömb2(J)
    J := J + 1
  Elágazás vége
  K := K + 1
Ciklus vége
```

Ha a *Tömb1*-en nem értünk végig, akkor a hátralévő elemeit is bemásoljuk a *Fésült* tömbbe. Az *I*-re szükségünk van, mert éppen a következő elem indexét mutatja, ezért új ciklusváltozót alkalmazunk:

```
Ha I <= MaxIndex1 Akkor
  Ciklus L := I-től MaxIndex1-ig
    Fésült(K) := Tömb1(L)
    K := K + 1
  Ciklus vége
```

Egyébként pedig a *Tömb2* maradék elemeit másoljuk át, a *J*-edik elemtől kezdve:

```
Egyébként
  Ciklus L := J-től MaxIndex2-ig
    Fésült(K) := Tömb2(L)
    K := K + 1
  Ciklus vége
Elágazás vége
Eljárás vége
```

Az algoritmust működés közben a **11–14. példában** láthatjuk.



## Közös elemek kiválogatása

Két tömb közös elemeinek kiválogatásánál tulajdonképpen halmazműveletet végzünk. Az eljárás két halmaz metszetét készíti el, csak most nem használhatjuk fel a szótárobjektumok metódusait.

Az algoritmus elkészítésénél feltételezzük, hogy egy-egy tömbön belül nincsenek egyforma elemek. A kiindulási tömböknél alkalmazott jelölések megfelelnek az előző algoritmus jelöléseinek.

Az eljárás során egy ciklussal kiválasztjuk a második tömb egy-egy elemét, majd összehasonlítjuk az első tömb elemeivel. Ha találtunk egyformákat, akkor befejezzük az összehasonlításokat:

```
Eljárás
  K := 0
  Ciklus J := 1-től MaxIndex2-ig
    I := 1
    Ciklus amíg I < MaxIndex1 és Tömb1(I) <> Tömb2(J)
      I := I + 1
    Ciklus vége
```

Az indexhatár túllépésének elkerüléséhez az utolsó elemet külön vizsgáljuk meg:

```
Ha Tömb1(MaxIndex1) <> Tömb2(J) Akkor
  I := I + 1
Elágazás vége
```

Így csak akkor találtunk egyforma elemeket, ha az I nem nagyobb, mint az első tömb elemszáma:

```
Ha I <= MaxIndex1 akkor
  K := K + 1
  Metszet(K) := Tömb2(J)
  Elágazás vége
Ciklus vége
Eljárás vége
```

Az eljárást a **11–15. példa** szemlélteti, ahol rendezett tömböket alkalmazunk, de az algoritmus ezt nem használja ki. Így azonban könnyebb ellenőrizni a helyes működést.

## Az elemek uniója

Gyűjtsük össze két tömb elemeit egy harmadik tömbben, de az egyforma értékeket csak egyszer vegyük át! Az eredmény megfelel két halmaz uniójának. Az előző feladathoz hasonlóan feltételezzük, hogy az eredeti tömböknél belül nincsenek ismétlődő elemek.

Először átmásoljuk az új tömbbe az egyik tömb elemeit, és a K változóban tároljuk, hogy hány eleme van eddig az uniónak:

```
Eljárás
  Ciklus I := 1-től MaxIndex1-ig
    Unió(I) := Tömb1(I)
  Ciklus vége
  K := MaxIndex1
```

Most a második tömb összes elemére megvizsgáljuk, hogy szerepel-e az unióban. Ha nem, akkor föl vesszük. Ehhez pontosan ugyanazt az algoritmust használjuk, mint a metszetnél, csak éppen ellenkező esetben másoljuk át az elemet az unióba:

```
Ciklus J := 1-től MaxIndex2-ig
  I := 1
  Ciklus amíg I < MaxIndex1 És Tömb1(I) <> Tömb2(J)
    I := I + 1
  Ciklus vége
  Ha Tömb1(MaxIndex1) <> Tömb2(J) Akkor
    I := I + 1
  Elágazás vége
  Ha I > MaxIndex1 Akkor
    K := K + 1
    Unió(K) := Tömb2(J)
  Elágazás vége
Ciklus vége
Eljárás vége
```

Az algoritmus alkalmazását a **11–16. példában** láthatjuk.

## 11.4. Rekurzív algoritmusok

### Rekurzív alprogramok

A programozási nyelvekben egy alprogram saját magát is hívhatja. Ezt a megoldást rekurziónak nevezzük, és nagyon sok bonyolult algoritmus megvalósítását lehet vele egyszerűsíteni.

Rekurzió: egy alprogram közvetlenül vagy közvetve (egy másik alprogramon keresztül) saját magát hívja.

A hívások során biztosítanunk kell, hogy leálljon a folyamat, ne kerüljünk végtelen hurokba.

Példaként definiáljuk egy természetes szám faktoriálisát a következőképpen. Az 1 faktoriálisa legyen 1. Az egynél nagyobb természetes számok faktoriálisát pedig úgy határozzuk meg, hogy a számot megszorozzuk a nála eggyel kisebb szám faktoriálisával. Ha bevezetjük a faktoriális felkiáltójellel történő jelölését, akkor:

- a)  $1! = 1$
- b)  $n! = n \cdot (n - 1)!$ , ha  $n > 1$

Az algoritmust egy függvénnyel valósíthatjuk meg:

```
Függvény Faktoriális(N)
  Ha N = 1 Akkor
    Faktoriális := 1
  Egyébként
    Faktoriális := N * Faktoriális(N-1)
  Elágazás vége
Függvény vége
```

A számítás során a függvény a paraméter egyre kisebb értékével hívja meg saját magát, amíg el nem jut az 1-hez. Majd visszafelé, az előző értékek ismeretében határozza meg a szorzatot. A 4 faktoriálisát például a következőképpen számítja ki:

$$\begin{aligned} 4! &= 4 \cdot 3! \\ 3! &= 3 \cdot 2! \\ 2! &= 2 \cdot 1! \\ 1! &= 1 \end{aligned}$$

$$\text{Így } 4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$$

A függvényt a **11–17. példa** mutatja be, melynek kódját kiegészítettük a szövegmezőbe írt szám ellenőrzésével. A faktoriálisok értéke gyorsan nő, ezért könnyen túlcsordulás léphet fel.

Az N faktoriális természetesen rekurzió nélkül is meghatározhatjuk, ha 1-től N-ig összeszorozzuk a számokat:

```
Szorzat := 1
Ciklus I := 1-től N-ig
    Szorzat := Szorzat * I
Ciklus vége
Ki: Szorzat
```

## A függvénynevek használata

A rekurzív hívások miatt félreérthetővé válik, ha a függvény nevét a definícióján belül az értékadás után egy kifejezésben használjuk. Példaként tekintsük az A és B globális változók legnagyobb közös osztóját meghatározó függvény alábbi VBScript forráskódját ( $A > B$ ):

```
Function Lnko()
    Lnko = A
    Do
        Maradek = Lnko Mod B
        Lnko = B
        B = Maradek
    Loop Until Maradek = 0
End Function
```

Mivel paraméterek hiányában nem kötelező az alprogramok hívásánál kitenni a zárójeleket, az `Lnko Mod B` kifejezésben nem egyértelmű, hogy az `Lnko` egy rekurzív hívás-e, vagy csak az előző értékadást akarjuk felhasználni. Célszerű a függvénynév helyett ideiglenes változót bevezetni, melynek értékét csak a kilépés előtt adjuk át a függvénynek. A fenti algoritmust például a következőképpen módosíthatjuk:

```
Function Lnko()  
  Temp = A  
  Do  
    Maradek = Temp Mod B  
    Temp = B  
    B = Maradek  
  Loop Until Maradek = 0  
  Lnko = Temp  
End Function
```

### A rekurzió alkalmazása

Készítsünk programot, amely meghatározza a nevezetes Fibonacci-féle sorozat elemeit. A sorozat első két tagja 1, egy következő tag pedig az előző két tag összegeként áll elő:

$$a_1 = 1, a_2 = 1, a_n = a_{n-1} + a_{n-2}$$

A sorozat első néhány tagja: 1; 1; 2; 3; 5; 8; 13; 21; 34; ...

A **11–18. példa** függvénye meghatározza a sorozat N-edik tagját. A függvényben először megadjuk az első két elemnek megfelelő értéket:

```
Function Fibonacci(N)  
  Fibonacci := 1
```

majd ha az N nagyobb, mint 2, akkor a sorozat definíciója alapján meghívjuk magát a függvényt a két előző elemre:

```
  If N > 2 Then  
    Fibonacci := Fibonacci(N-1) + Fibonacci(N-2)  
  End If  
End Function
```

A rekurzió nagyon elegáns módszer az összetett feladatok megoldására, de általában hosszadalmas, erőforrásigényes és nem túl hatékony eljárás.

Előző függvényünk például a `Fibonacci(5)` kiszámításánál meghívja a `Fibonacci(4)`-et és a `Fibonacci(3)`-at. A `Fibonacci(4)` megint meghívja a `Fibonacci(3)`-at, holott annak az értékét egyszer már meghatároztuk. Elképzelhető, hogy nagyobb N-ek vagy több tag kiszámítása esetén hány fölösleges függvényhívás történik. Ráadásul az interpreternek tudnia kell, hol folytassa a rekurzív hívás miatt félbehagyott kifejezés kiértékelését. Ehhez egy speciális memóriaterületet, az úgynevezett vermet használja fel. A verem mérete korlátozott, túl sok egymásba ágyazott hívás esetén betelik, és a programunk hibaüzenettel leáll.

A **11–19. példa** a 100. tagig számítaná ki a Fibonacci-sorozat elemeit. A ciklus futása az állapotsorban követhető. A 30. tag körül annyira lelassul a számítás, hogy hasznavehetetlenné válik a megoldás.

A fenti elemzés alapján egy sokkal egyszerűbb algoritmust is készíthetünk a sorozat elemeinek meghatározására. Tároljuk a két előző tagot, majd az újabb elem kiszámítása után módosítsuk az értéküket. A harmadik elemtől kezdve ciklust használunk:

```

Elozo1 := 1
Elozo2 := 1
Osszeg := 0
For I := 3 To N
    Osszeg := Elozo1 + Elozo2
    Elozo1 := Elozo2
    Elozo2 := Osszeg
Next
Fibonacci := Osszeg

```

A **11–20. példában** meghatározzuk a sorozat első 200, illetve az ezredik tagját. Hasonlítsuk össze a futási időt a 11–19. példáéval. Több egymás után következő tag kiszámításánál tovább egyszerűsíthetjük a megoldást, ha az előző két elemet globális változóban tároljuk. Így az algoritmus ezek összeadására és módosítására egyszerűsödik.

A rekurzív algoritmusok minden esetben átírhatók iterációra (ciklus). A rekurzív program rövidebb és áttekinthetőbb, futtatása azonban lassúbb, és nagymértékben lefoglalja az erőforrásokat. A választásnál mérlegelni kell a memóriaigényt és a futási időt. Ha ismerjük az iterációs megoldást, akkor célszerű azt alkalmazni.

### Gyorsrendezés (quick sort)

A rekurzió segítségével nagyon elegáns rendezési algoritmust készíthetünk. Nagyméretű tömbök és véletlenszerűen választott elemek esetén ez az egyik leggyorsabb rendezési módszer.

A gyorsrendezés során kiválasztjuk a tömb egyik elemét, például a középsőt (páros elemszám esetén az egyik középsőt), majd a tömb elejéről kezdve keresünk egy nála nagyobb, a végéről kezdve pedig egy nála kisebb elemet. Ezt a két elemet felcseréljük, majd haladunk tovább mindkét oldalról a másik felé.

Ha az ismétlések során a bal oldali elem indexe nagyobb lesz, mint a jobb oldalié (átlépték egymást), akkor a tömböt két részre osztottuk fel, melyek közül a bal oldali részben lévő elemek mind kisebbek, mint a jobb oldalon lévő elemek. Ezek után az eljárást rekurzív módon meghívjuk a tömb mindkét felére mindaddig, amíg a részek mérete le nem csökken egy elemre. Ekkor már rendezett lesz a tömb.

A rekurzió miatt az egész algoritmust egy külön eljárásba tesszük, melynek hívásánál megadjuk a vizsgált tartomány első és utolsó elemének indexét. Ezek először megfelelnek az egész tömb első és utolsó elemének:

```

Eljárás
    Gyorsrendezés(1, N)
Eljárás vége

```

Az eljárásban először egy ideiglenes változóban tároljuk a tömb középső elemét:

```

Eljárás Gyorsrendezés(Első, Utolsó)
    Temp := Tömb((Első + Utolsó) \ 2)

```

A keresést az első és utolsó elemmel kezdjük. A ciklust addig ismételjük, amíg össze nem érünk a két oldalról:

```
Bal := Első
Jobb := Utolsó
Ciklus amíg Bal <= Jobb
```

A ciklusban először a tömb elejétől kezdve megkeressük az első olyan elemet, amely nagyobb a félretett értéknél:

```
Ciklus amíg Tömb(Bal) < Temp
    Bal := Bal + 1
Ciklus vége
```

Ezután a tömb végétől kezdve megkeressük az első olyan elemet, amely kisebb a félretettnél:

```
Ciklus amíg Tömb(Jobb) > Temp
    Jobb := Jobb - 1
Ciklus vége
```

Ha még nem keresztezték egymást, akkor felcseréljük őket:

```
Ha Bal <= Jobb akkor
    Tömb(Bal) és Tömb(Jobb) cseréje
```

majd mindkét oldalon a következő elemre lépünk. Ezzel eljutottunk a ciklus végéhez:

```
Bal := Bal + 1
Jobb := Jobb - 1
Elágazás vége
Ciklus vége
```

A cserét befejeztük. Ha a tömb valamelyik része egynél több elemet tartalmaz, akkor ugyanezt az eljárást elvégezzük arra a részre is:

```
Ha Első < Jobb Akkor Gyorsrendezés(Első, Jobb)
Ha Bal < Utolsó Akkor Gyorsrendezés(Bal, Utolsó)
Eljárás vége
```

Az eljárás használatát a **11–21. példa** szemlélteti. A **11–22. példában** minden egyes csere után kiírtuk a sorrendet. A félretett értéket félkövéren jelenítettük meg, a bal oldali elemet piros, a jobb oldalt pedig kék számjegyekkel jelöltük. A rekurzív hívások sorszámánál pedig azt is jeleztük, hogy a bal vagy a jobb oldali részre térünk rá.

## 11.5. A visszalépéses keresés (back track)

Az előzőekben bemutatott keresési algoritmusokban egy elem kiválasztását nem befolyásolták az előző választások. A páros számok gyűjtésénél az újabb elemet hozzávettük az előzőleg találtak listájához, nem volt szükség törlésre.

Sok feladatban azonban egy elem kiválasztása módosíthatja az addigi választásokat. Ha egymással rokonszenvező diákokat szeretnénk egy adott létszámú csapatba összeválogatni, akkor megtörténhet, hogy valamelyik diákkal indulva nem jön össze a megfelelő létszám, és előről kell kezdenünk a válogatást.

Az összes eset végigpróbálása helyett az úgynevezett visszalépéses keresést alkalmazzuk. Ha egy választás után nem tudunk továbblépni, akkor visszalépünk az előző elemhez, és helyette egy másikat választunk.

A visszalépéses keresés alkalmazása már némi programozói gyakorlatot igényel. A kezdő programozó nyugodtan kihagyhatja ezt a részt.

### A visszalépéses keresés algoritmus

A visszalépéses keresés algoritmusát egy általános feladat megoldásával fogalmazzuk meg. Legyen  $N$  darab sorozatunk, melyek elemeinek száma  $MaxIndex(1)$ ,  $MaxIndex(2)$ , ...,  $MaxIndex(N)$ . Válasszunk ki mindegyikből egy-egy elemet úgy, hogy a kiválasztott elemek előre megadott kapcsolatban legyenek egymással (például a következő elem nagyobb legyen, mint az előző)!

Először az első sorozatból választunk egy elemet. Aztán rátérünk a következőre, és abból választunk. Ezt mindaddig ismételjük, amíg megfelelő elemeket tudunk választani. Ha megakadtunk, akkor visszalépünk az előző sorozatra, választunk egy másik elemet, majd újra próbálkozunk a következővel.

Az eljárás akkor ér véget, ha sikerült minden sorozatból egy elemet kiválasztani, vagy visszaértünk az első sorozathoz, és abból már nem tudunk választani. Ez utóbbi esetben nincsen megoldása a feladatnak.

Az algoritmus során a kiválasztott elemek sorszámát a *Választás* tömbben gyűjtjük. A tömb  $I$ -edik eleme megadja az  $I$ -edik sorozatból választott elem indexét. Ha még nem választottunk egy sorozatból, akkor az index értéke nulla.

*Az algoritmus mondatyszerű leírása*

A *Választás* tömb nullázása után a keresést az első sorozattal kezdjük. A vizsgált tömb sorszámát az  $I$  változóban tároljuk:

```
Eljárás
  a Választás tömb elemeinek nullázása
  I := 1
```

Az eljárást addig ismételjük, amíg ki nem választottuk az összes elemet, vagy az első sorozatnál is vissza kellene lépünk, mert nem tudunk belőle választani:

```
Ciklus amíg I >= 1 És I <= N
```

Végignézzük az I-edik tömb elemeit, hogy találunk-e közöttük megfelelőt. Ezt a *NemJóElem* függvénnyel fogjuk ellenőrizni, amely az ismétlést jelző logikai értéket ad vissza. A függvény utasításait egyelőre nem részletezzük. Ha egy elem nem felel meg, akkor rátérünk a következőre:

```
Ciklus
  Választás(I) := Választás(I) + 1
amíg NemJóElem(I) = igaz
Ciklus vége
```

A ciklus után megvizsgáljuk a *Választás(I)* értékét. Ha nem lépte túl az I-edik sorozat elemeinek számát, akkor találtunk megfelelő elemet, melynek indexét éppen a *Választás(I)* tárolja:

```
Ha (Választás(I) <= MaxIndex(I)) Akkor
  I := I + 1
```

Egyébként nullázzuk az előző választás során feljegyzett indexet, és visszalépünk az előző sorozathoz:

```
Egyébként
  Választás(I) := 0
  I := I - 1
Elágazás vége
Ciklus vége
```

Ha minden sorozatból sikerült elemet választani, akkor az I értéke túllépi a sorozatok számát. Ha az első sorozatból sem tudtunk választani, akkor az I értéke 0 lesz. Mindkét eset a ciklus befejezését okozza.

A megoldás létezését az I értéke jelzi:

```
VanMegoldás := (I > N)
Eljárás vége
```

### *A NemJóElem függvény algoritmusa*

A függvény meghatározza, hogy szükséges-e ismételni a keresés ciklusát.

Ha már túlléptük az I-edik sorozat maximális indexét, akkor nem ismétlünk:

```
Függvény NemJóElem(I)
  Ha Választás(I) > MaxIndex(I) Akkor
    NemJóElem := hamis
```

Egyébként összehasonlítjuk az újonnan kiválasztott elemet az előzőleg kiválasztott elemekkel, hogy megfelelnek-e az előírt tulajdonságnak:

```
Egyébként
  J := 1
  Ciklus amíg J < I És
    az I-edik sorozat Választás(I)-edik eleme és a
    a J-edik sorozat Választás(J)-edik eleme
    megfelelő kapcsolatban van egymással
  J := J + 1
Ciklus vége
```



Ha a vizsgálatok végén a *J* kisebb, mint az *I*, akkor találtunk hibás kapcsolatot, így a keresés ciklusát a következő elemmel meg kell ismételni:

```
NemJóElem := (J < I)
Elágazás vége
Függvény vége
```

Ezzel az algoritmus leírását befejeztük.

## A visszalépéses keresés alkalmazása

A visszalépéses keresés algoritmusát először egy olyan feladaton mutatjuk be, melynél könnyű ellenőrizni a megoldhatóságot. Megadunk 5 számsorozatot. Minden sorozatnak 10 eleme van, ezek 1 és 99 közé eső véletlenszámok. A számokat a *Tomb* kétdimenziós tömbben tároljuk. Az első dimenzió indexe a sorozat száma (1-től 5-ig), a második dimenzió indexe az elem sorszáma (1-től 10-ig).

Válasszunk ki a sorozatokból egymás után egy-egy elemet úgy, hogy a kiválasztott elemek nagyság szerint növekvő sorozatot alkossanak! Mivel az 1 és 99 közé eső számokkal majdnem mindig van megoldás, ezért az utolsó sorozat csak 30 és 39 közé eső elemeket tartalmaz. Így gyakrabban kapunk olyan számsorozatokat, melyeknél nincs megoldása a feladatnak.

Az elemek kiválasztását egyszerűbb algoritmussal is elvégezhetnénk, hiszen mindig a lehető legkisebb számot választva a sorozatokból könnyen eljutunk a megoldáshoz, ha létezik. Ezt az eljárást alkalmazhatjuk a visszalépéses kereséssel kapott eredmény ellenőrzésére.

A **11–23. példa** pontosan a fent leírt algoritmus VBScript programját mutatja be. Mivel most a sorozatok elemszáma megegyezik, ezt *MaxIndex*-szel jelöltük.

A mondatszerű leírással megadott algoritmus pontosítására egyedül a *NemJóElem* függvényben van szükség. Ha a kiválasztott tömbelemek kisebbek, mint az új elem, akkor jól választottunk. Így a ciklus ismétlési feltétele:

```
Ciklus amíg J < I És
    Tömb(J, Választás(J)) < Tömb(I, Választás(I))
```

Megjegyezzük, hogy a növekvő sorozat miatt elegendő lenne az új elemet az utoljára kiválasztott elemmel összehasonlítani. Az általános algoritmusokat egy konkrét feladatnál gyakran lehet egyszerűsíteni.

A példa forráskódjában a sorozatok és az eredmény megjelenítését végző eljárások hosszabbak, mint a keresés algoritmus! A kiválasztott elemek háttérét a táblázatban beszíneztük. A feladat megfogalmazásánál említett egyszerű eljárással ellenőrizhetjük a program által szolgáltatott megoldást.

## 8 vezér a sakktáblán

A visszalépés keresés alkalmazásának klasszikus feladata 8 vezér elhelyezése a sakktáblán úgy, hogy ne üssék egymást. A sakkban a vezérek vízszintesen, függőlegesen és átlósan tetszőleges számú mezőt léphetnek.

Az algoritmus teljes egészében megfelel az előző feladat módszerének. Egyedül a *NemJoElem* függvényben kell módosítani a vizsgálatot. Egy új vezért akkor helyeztünk el jó helyre, ha ugyanabban a sorban, ugyanabban az oszlopban és átlósan nincs másik vezér a táblán. A sakkfigurákat soronként helyezzük el, a *Valasztas* tömb elemei az általuk elfoglalt oszlopokat adják meg.

Mivel egy sorba csak egy vezért teszünk, elegendő ellenőrizni az oszlopot és az átlót. Nincsenek ugyanabban az oszlopban, ha:

```
Valasztas(I) <> Valasztas(J)
```

Nincsenek egy átló mentén, ha a sorok számának az eltérése nem egyezik meg az oszlopok számának az eltéréseivel:

```
Abs(I - J) <> Abs(Valasztas(I) - Valasztas(J))
```

Az algoritmus működését a **11–24. példa** mutatja be. Figyeljük meg, hogy a fenti vizsgálatot és a megjelenítést kivéve a program sorról sorra megegyezik a 11–23. példa szkriptjével!

## Stratégia meghatározása a visszalépés kereséssel

A visszalépés keresés alapelvét nem csak sorozatokból történő kiválasztásoknál alkalmazhatjuk. Minden olyan esetben célhoz vezet, ahol – jobb híján – próbálgatással jutunk el a megoldáshoz, de az összes lehetséges eset kipróbálása helyett céltudatosan szeretnénk előrehaladni.

Példaként röviden bemutatjuk egy logikai feladat megoldását a visszalépés kereséssel. Elhelyezünk egy sorba három kék és három piros kört, középen egy körnek megfelelő üres hellyel. Feladatunk a kék és piros körök felcserélése. A piros körök csak balra, a kékék csak jobbra léphetnek egy üres helyre. Egy ellenkező színű kört át lehet ugrani, ha mögötte van üres hely.

Stratégiánk a lehető legegyszerűbb alapelvre épül. Elkezdünk lépegetni a piros körrel, amíg csak lehetséges. Közben minden egyes lépésnél feljegyezzük, hogy léphetnénk-e kék körrel is. Egy ilyen elágazásig kell visszalépnünk, ha valahol elakadunk. A piros körökkel történő lépéseknél ezért meg kell vizsgálnunk, hogy az utolsó lépés nem visszalépés volt-e. Ekkor ugyanis nem léphetünk ismét piros körrel, mert végtelen ciklusba kerülünk:

```
Ha Nem Visszalép És PirosTudLépni Akkor  
  PirosLép
```

Ha nem tudunk piros körrel lépni, akkor kék körrel próbálkozunk. A kék körrel való lépésnél feljegyezzük, hogy az elágazás mindkét ágát bejártuk-e már. Ha ugyanis a visszalépések során jutunk egy ilyen elágazáshoz, akkor már kék körrel sem léphetünk, hanem még többet kell visszalépnünk, az előzőleg be nem járt elágazásig:

```
Egyébként ha KékTudLépni És
      (Nem Visszalép Vagy VanSzabadElágazásÁg) Akkor
      KékLép
```

Ha egyik körrel sem tudunk lépni, akkor vagy megoldottuk a feladatot:

```
Egyébként ha Vége Akkor
      megoldottuk a feladatot
```

vagy pedig vissza kell lépünk, amíg csak lehet:

```
Egyébként ha LehetMégVisszalépni Akkor
      Visszalép
Egyébként
      nincs megoldás
Elágazás vége
```

A **11–25. példa** *Kereses* eljárásának *Do* ciklusa ezt a folyamatot ismételteti, amíg csak lehet lépni. A körök számát a felhasználó adhatja meg.

Az egyes lépéseknél kialakuló pozíciókat a *Sor* kétdimenziós dinamikus tömbben jegyezzük fel. Az üres helyet szóköz, a piros köröket P betű, a kék köröket pedig K betű jelzi. A tömb elején és végén még két elemet fenntartunk, hogy a vizsgálatoknál az indexek ne lépjenek túl a megengedett határokat. A 0 indexű elemben egyben feljegyezzük, hogy van-e be nem járt elágazás, az 1-es indexű elemben pedig azt, hogy utoljára visszalépés történt-e.

A programot érdemes futtatni egyre növekvő számú körrel. Hamarosan felfedezhetjük azt a stratégiát, ami próbálkozások nélkül, nyílegyenesen is elvezet a megoldáshoz. Ennek felismerését segíti elő a visszalépéses keresés. A közvetlen algoritmust alkalmazó program megírását az Olvasóra bízunk.

## 11.6. A programfejlesztés folyamata

Egy program elkészítése nem a forráskód megírásával kezdődik, és nem is azzal fejeződik be. Az eddigiekben többször utaltunk a feladat pontos megfogalmazására, a tervezés fontosságára. A programot tesztelünk kell, és a programíráshoz tartozik a használati utasítás, illetve a későbbi módosításhoz, bővítéshez szükséges fejlesztői dokumentáció elkészítése is. A fejezet hátralévő részében röviden áttekintjük ezeket a tennivalókat.

### Tervezés és kódolás

Amikor felmerül egy program megírásának az ötlete, még nem világos, hogy pontosan mi a feladat. Célszerű áttekinteni a követelményeket és elvárásokat. Egyértelműen meg kell határozni, milyen adatokra van szüksége a programnak, illetve milyen adatokat eredményezzen. A bevitel és kivitel rögzítése sok későbbi felesleges munkától, módosításoktól, hibakereséstől mentesít.

Feladatspecifikáció: a programmal szemben támasztott követelmények, a bevitel és kivitel részletes megfogalmazása.

Gondoljunk a bővítésre! A későbbi fáradságos módosításokat kiküszöbölhetjük, ha nem egy konkrét eset, hanem egy hasonló, de általánosabb feladat megoldására készítünk programot.

A feladat egyértelmű megfogalmazását követi a tervezés folyamata. A feladatot szétválasztjuk önálló modulokra, elkülönítjük a megoldáshoz szükséges tennivalókat. A tervezés gyakran alkalmazott módszere a szükséges lépések felülről-lefelé történő lebontása. Először nagy vonalakban fogalmazunk meg egy-egy lépést, majd egyre kisebb részekre bontjuk fel, míg el nem érkezünk a szinte közvetlenül kódolható algoritmusig.

Pontosan meg kell határozni az egyes modulok feladatát, azokat az adatokat, amelyekre szükségük van és a visszatérési értékeket. Minden modul csak egyféle dolgot tegyen, de azt jól!

Már a tervezés során gondoljunk az áttekinthető képernyőképek létrehozására! Egyértelműen jelöljük meg, mikor mit várunk a felhasználótól. Adjunk lehetőséget a beviteli hibák javítására, de kíméljük meg a sok, fölösleges egérgattintástól. Tájékoztassuk a várt adatok mértékegységéről, az elfogadható értéktartományról.

Tervezés közben használjuk a megismert grafikus vagy szöveges algoritmusleíró eszközöket (folyamatábra, pszeudokód).

A programterv alapján készítjük el a forráskódot. Kódolás közben ügyeljünk a jól áttekinthető szerkesztésre, használjunk tömör megjegyzéseket! A beszédes azonosítók nagymértékben segítik a program áttekintését és a hibakeresést.

## **A programok tesztelése**

Bármilyen gondosan végeztük a tervezést és a kódolást, a program általában számos hibát tartalmaz.

A szintaktikus hibákat már az első próbánál kiszűri a böngésző. Sokkal nehezebb a logikai hibák felderítése és javítása.

Szemantikai hiba: logikai hiba. A program nem az elvárásoknak megfelelően működik.

A programban akkor is maradhatnak hibák, ha néhány próbafuttatás során látszólag jó eredményeket produkál. Ezért a tesztelés folyamán ne csak szokványos, gyakran előforduló adatokat használjunk. A tesztadatokat úgy kell összeállítani, hogy azok minden lehetséges esetet lefedjenek. Teszteljük az adatbevitelt az értéktartományok határain elhelyezkedő, illetve hibás adatokkal is!

A tesztelés során hajtódjon végre minden utasítás, a szelekciók minden egyes ága. A logikai kifejezések adjanak hamis, illetve igaz értéket is. Próbálkozzunk olyan adatokkal, melyeknél egy-egy ciklusnak egyszer sem kell lefutnia.

A tesztelést célszerű nem csak a tesztadatok futtatásával, hanem a forráskóddal is elvégezni. Vizsgáljuk meg, hogy minden változó kapott-e kezdőértéket, nem maradt-e a kódban egy-egy módosítás után már fel nem használt változó, egyáltalán végrehajtódik-e minden utasítás. A megírása után néhány nap elteltével olvassuk el újra az algo-

ritmust, a forráskódot. Minden tevékenységet a megfelelő helyen végez-e el, nem maradt-e ki szükséges vizsgálat?

Hiba esetén alaposan gondoljuk át a javítást. A módosítások újabb hibákhoz vezethetnek!

Ha nem jövünk rá a hibás működés okára, akkor a kritikus helyeken a *window.alert* metódus segítségével jelenítsük meg a gyanús változók értékét. Gyakori hibalehetőség a változók inicializálásának elmulasztása, globális változók egymással ütköző felhasználása. Az *alert* metódussal ellenőrizhetjük egy-egy szelekciós ág végrehajtását, egy-egy feltétel bekövetkezését is. Haladjunk a hiba helyétől visszafelé, amíg csak hibásnak találjuk a végrehajtást. Az utolsó ilyen lépés gyakran elvezet a hiba okának felderítéséhez.

A tesztelést hibakereső (debugger) programok segítik. Egy ilyen programmal töréspontokat helyezhetünk el a forráskódban. A töréspontoknál megszakad a program végrehajtása. Megvizsgálhatjuk és módosíthatjuk a változók értékét, lehetőségünk van az utasítások egyesével történő végrehajtására. Ezt lépésenkénti üzemmódnak nevezzük.

A függelékben ismertetjük a Microsoft webhelyéről letölthető Microsoft Script Debugger, és a Microsoft Office részét képező Microsoft Script Editor programokat, melyek hatékony hibakeresést tesznek lehetővé.

A program tesztelése legyen minél teljesebb, csökkentsük minimálisra a későbbiekben bekövetkező futási hibák lehetőségét!

## **A programok dokumentálása**

A forráskódban elhelyezett megjegyzések segítik a program későbbi áttekintését, módosítását, bővítését. Mellettük azonban szükség van egy felhasználói és egy fejlesztői dokumentáció megírására is.

A felhasználói dokumentáció (user's guide) a programot használó személyek számára készül. Ez tartalmazza:

- a program céljának megfogalmazását,
- a futtatáshoz szükséges hardver- és szoftverigényeket,
- a program telepítésének, indításának módját,
- a program kezelését, a menük részletes leírását,
- az input és output adatok ismertetését,
- a hibaüzenetek felsorolását, a szükséges teendőket.

Célszerű bemutatni egy mintapélda futtatását, tájékoztatni a felhasználót a tipikus futási időről és a program korlátairól is.

A fejlesztői dokumentáció (developer's guide) a programozók számára készül. Olyan szinten ismerteti a programot, az algoritmust, a forráskódot, hogy a program íróján kívül más hozzáértő személy is el tudja végezni a szükséges módosításokat.

A fejlesztői dokumentáció tartalmazza:

- a pontos feladat-specifikációt,
- az adatmodell az input és output adatokkal,
- a felhasznált változók szerepét, típusát,
- az algoritmus részletes leírását,
- a képernyőterveket,
- a tesztadatokat és teszteredményeket,
- magát a forrásprogramot.

A fejlesztői dokumentációban utalhatunk a későbbi bővítési lehetőségekre is.

A felhasználói és fejlesztői dokumentáció megírásának elmulasztása a későbbiekben sok bosszúsághoz, a forráskód utólagos „visszafejtésének” fáradságos munkájához vezet!

### A hatékonyság növelése

A programok hatékonyságának legfontosabb ismérvei a memóriaigény és a futási idő. Minél kevesebb memóriát igényel a program, és minél gyorsabban elvégzi a feladatát, annál hatékonyabbnak tekintjük a működését. Ez a két feltétel gyakran ellentétben van egymással. A program tervezésénél el kell dönteni, hogy melyiket tekintjük fontosabbnak, milyen kompromisszum köthető az optimális cél elérése érdekében.

Nem vezet jelentős sebességnövekedéshez olyan részletek gyorsítása, melyek csak ritkán kerülnek végrehajtásra. Néhány elem rendezésénél nem jelenti a memóriaigény növelését, ha először felvesszük őket egy rendezett listára, majd átmásoljuk a szükséges tömbbe. Azokra a programrészekre célszerű koncentrálni, ahol a legkevesebb munkával tudjuk a legtöbb előnyt elérni.

A hatékonyságra már a tervezés során, az algoritmus kiválasztásánál gondolnunk kell. Az utólagos módosításokkal nem csak a program szerkezetét bonyolíthatjuk el, de nehezen felderíthető hibákat is véthetünk.

Az alábbiakban felsorolunk néhány olyan tanácsot, melyek betartásával csökkenthetjük a végrehajtási időt vagy a felhasznált memória méretét:

- A gyakran használt mennyiségeket (numerikus értékeket és sztringeket) tároljuk konstansokban. A forráskódban szereplő adatokat az interpreternek minden egyes esetben át kell alakítani bináris értékkel, ami jelentős mértékben megnöveli a futási időt.
- Statikus tömbök helyett alkalmazzunk dinamikus tömböket, lehetőség szerint szabadítsuk fel az általuk elfoglalt helyet (*Erase*, illetve *Redim* utasítások).
- Ha csak lehet, szótárobjektumokat használjunk a tömbök helyett.
- A végrehajtási idő nagyságát főleg a ciklusok növelik. Az ismétlések számának csökkentésére használjuk ki az adatok speciális tulajdonságait (sokkal gyorsabb például a keresés, ha tudjuk, hogy a tömb elemei rendezettek).
- Gondoljuk meg, hogy egy-egy kifejezés értékének meghatározását nem végezhetjük-e el a ciklus előtt. Ugyanazt a kifejezést fölöslegesen ne értékeljük ki minden egyes ismétlésnél.

- Használjuk a *For Each...Next* ciklust a *For...Next* helyett.
- A gyakran hívott eljárások utasításait illesszük be a hívás helyére. Főleg a ciklusokon belül kerüljük el – lehetőség szerint – az alprogramok hívását.
- Ha nincs szükség a paraméterek alprogramon belüli módosítására, akkor a cím szerinti (*ByRef*) helyett érték szerinti paraméterátadást (*ByVal*) alkalmazunk.
- Az objektumok tulajdonságainak eléréséhez használjuk a *Set*, illetve a *With...End With* utasítást.
- Ha egy DHTML-objektum valamely tulajdonságára többször is szükségünk van, akkor célszerű először változóban tárolni, majd ezt a változót használni az értékadáshoz.
- A DHTML-objektumok tulajdonságainak olvasása, írása nagyban lelassítja a program végrehajtását. A kiírásra kerülő sztringet először egy változóban állítsuk össze, majd az elkészítése után adjuk át a megjelenítést végző objektumnak.
- Táblázatok készítésénél mindig adjuk meg a HTML-kódban a táblázat, illetve a cellák szélességét.
- Képek megjelenítésénél adjuk meg a HTML-kódban a kép méreteit.
- Fájlok használatánál egyszerre minél nagyobb részeket olvassunk be, illetve írjunk ki.
- Amint lehet, szüntessük meg a feleslegessé vált fájlrendszer-objektumokat.

A fentiekben főleg a forráskódra vonatkozó javaslatokat tettünk. A hatékonyság azonban növelhető az algoritmus módosításával is. A program szerkezetének egyszerűsítése, a fölösleges vizsgálatok elhagyása, a ritkán előforduló esetek kiküszöbölése, a matematikai ismeretek felhasználása nagymértékben csökkentheti a futási időt. Ezekre a tevékenységekre nem lehet általános receptet adni, minden egyes esetben gondosan elemezni kell a felhasznált algoritmust. A hatékonyság növelése sok tapasztalatot és nagy gyakorlatot igénylő tevékenység.





# FÜGGELÉK



# F1. A TEXTPAD ALKALMAZÁS

A TextPad a Helios Software Solutions szövegszerkesztő programja. A Windows bármely változatánál telepíthető. Az általánosan használt szövegfájlok túl alkalmas a C/C++, Java és HTML forráskód szerkesztésére. Felismeri a kulcsszavakat, különböző színnel emeli ki a szerkezeti elemeket. Alkalmas a bináris fájlok hexadecimális kódjának módosítására is.

Számos bővítéssel rendelkezik, melyek kiterjesztik a lehetőségeit a Pascal, C#, Lisp, Perl stb. szintaxisának a felismerésére. A bővítések a TextPad újabb változataival együtt a következő címről tölthetők le:

<http://www.textpad.com>

## A TextPad telepítése és indítása

A TextPad 4.7-es próbaváltozatának telepítőprogramja megtalálható a könyvhöz mellékelt CD-n. A próbaváltozat nincsen korlátozva sem a funkciók, sem az időtartam szempontjából. Felhívjuk az Olvasó figyelmét arra, hogy folyamatos használata esetén regisztráltatni kell a programot.

Indítsuk el az Intéző segítségével a CD-melléklet TextPad mappájában lévő TextPad470.exe fájlt. A licenszfeltételek elfogadása után beírhatjuk a felhasználó nevét és szervezetét, majd szükség esetén megváltoztathatjuk a telepítés helyét (mappáját). A telepítés minden beavatkozás nélkül, csupán a Next gombokra történő kattintással elvégezhető.

Telepítés után a program ikonja bekerül a Start/Programok listába, ahonnan a szokásos módon indítható. Indításkor és a használata során időnként figyelmeztet a regisztráció szükségességére.

A program kényelmesen indítható úgy is, hogy az Intézőben jobb egérgombbal kattintunk a megnyitni kívánt fájlra. Az előbukkanó menüben megtaláljuk a TextPad-et, amelyre rákattintva a kijelölt állomány betöltésével elindul a program.

## A program beállításai

A program beállításait a Configure menü Preferences parancsával módosíthatjuk. A színek megválasztásától kezdve, az automatikus mentés gyakoriságán keresztül, saját menüpontok felvételéig számos lehetőség áll a rendelkezésünkre.

A program használatához a következő beállításokat javasoljuk:

### *General:*

- Ne engedélyezzük a több példányban való indítást  
(ne legyen pipa az „Allow multiple instances to run” feliratnál).

### *File:*

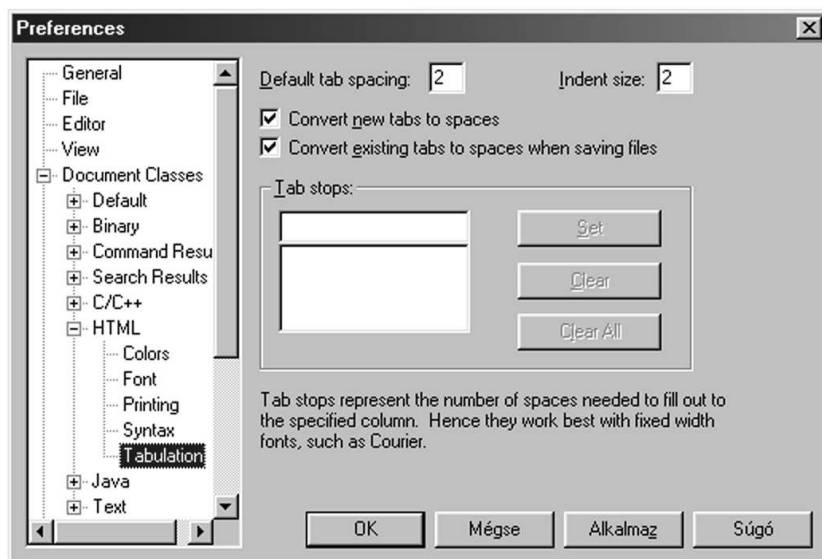
- Írjunk be htm-et a „Default file extension” szövegmezőjébe.

### *View: View Options*

- Kapcsoljuk be a „Line numbers” (sorok számozása) megjelenítését.

*Document Classes: HTML/Tabulation*

- Állítsunk be 2–2 karaktert a „Default tab spacing” és az „Indent size” értéknél.
- Jelöljük be a „Convert new tabs to spaces” és a „Convert existing tabs to spaces when saving files” szolgáltatásokat.



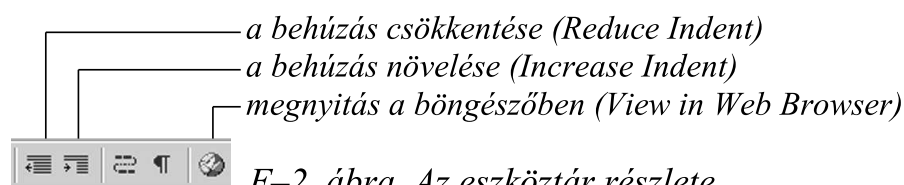
F–1. ábra. A tabulátorpozíciók állítása

Ez utóbbi beállításokkal elérjük, hogy az Enter billentyű lenyomásakor a program megtartja az előző sorok megfelelő behúzást, illetve a tabulátor billentyűt 2 szóközzel helyettesíti. Így könnyen strukturálhatjuk a forráskódot.

**A TextPad használata**

A TextPad legtöbb menüpontja, parancsa megegyezik a szövegszerkesztőknél megszokottakkal. A fájlok megnyitását, szerkesztését, mentését számos további szolgáltatás segíti, melyek részletes áttekintésére nincs módunk. A program Súgójában (Help) tájékoztatást találunk a használatról, itt csak néhány hasznos lehetőséget említünk meg.

A forráskód szerkesztése közben a kurzor sorának és oszlopának helyét az állapot-sorban látjuk. A sorok számozásának bekapcsolásával szükség esetén könnyen megtaláljuk a hiba helyét, amelyre a böngésző hibaüzenete utal.



F–2. ábra. Az eszköztár részlete

Az „Increase Indent” és a „Reduce Indent” ikonok segítségével az aktuális sor, illetve a kijelölt sorok behúzását növelhetjük vagy csökkenthetjük, a beállításoknál megadott mértékben (például 2 karakterrel). Ez megkönnyíti a strukturált szerkezet kialakítását. Ezeket a funkciókat a Tab, illetve a Shift+Tab billentyűkkel is elérhetjük.

HTML-fájlok esetén a „View in Web Browser” ikonnal megjeleníthetjük a weblapot a böngészőben.

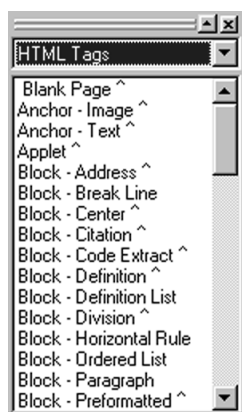
Új állomány mentésénél ne felejtsük el beállítani a HTML fájl típust!

## A kódrészletkönyvtárak

A TextPad a képernyő legnagyobb részét elfoglaló, a megnyitott fájl tartalmát mutató dokumentumablak mellett két kisebb ablakot is megjelenít. A felső, dokumentumválasztó ablakban a megnyitott fájlok listáját látjuk. Az alsó panel a kódrészletkönyvtárakat tartalmazza.

A kódrészletkönyvtárak olyan karaktersorozatokot tárolnak, melyek gyakran előfordulnak a forráskód írásánál. Ezeket dupla egérekattintással be tudunk illeszteni a készülő dokumentumba.

A kódrészletkönyvtárak között megtaláljuk az ANSI-karaktereket (ANSI Characters) és a karakterentitások jó részét is (HTML Characters).



F-3. ábra.  
A HTML Tags  
könyvtár

Válasszuk ki a legördülő menüből a „HTML Tags” könyvtárat. A dokumentum ablakba gépeljük be a „Címsor” szöveget, és jelöljük ki (például dupla egérekattintással). Ha most a kódrészletkönyvtárban duplán kattintunk a „Heading 1 ^” felírra, akkor a TextPad a kijelölt szöveget az 1-es szintű címsornak megfelelő nyitó és záró taggal veszi körül:

```
<H1>Címsor</H1>
```

Ha nem gépelünk be előzetesen szöveget, akkor csak a nyitó és záró tagot kapjuk meg, közbeszúrt tartalom nélkül.

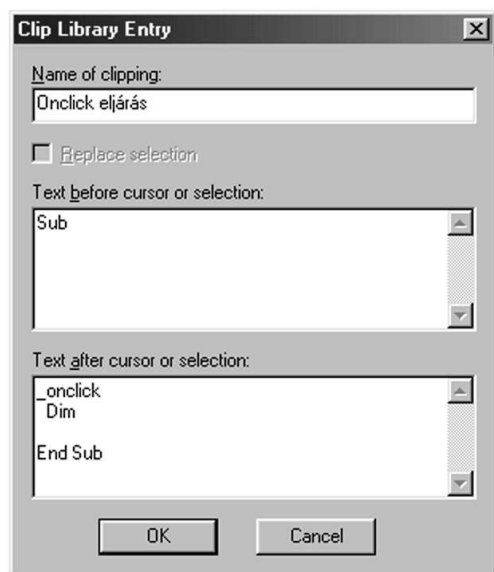
A „HTML Tags” könyvtár a leggyakrabban előforduló HTML-elemeket tartalmazza. A bekezdést a Block csoportban találjuk (Block - Paragraph). A lista első eleme, a „Blank Page ^” egy üres HTML-lapot készít az előforduló legfontosabb elemekkel.

Ha a kódrészletkönyvtár listájában egy elem fölé visszük az egeret, akkor a felbukkanó ablak megmutatja a beilleszthető kódrészletet. A „\^” karakterek a kijelölt szövegrészt helyettesítik.

## A kódrészletek módosítása

Ha a jobb egérgombbal a lista egy elemére kattintunk, akkor az Edit parancs kiválasztásakor a megnyíló ablakban módosíthatjuk a beilleszthető kódot. A „Text before cursor or selection” szövegmezőbe kerül a kijelölés elé, a „Text after cursor or selection” mezőbe pedig a kijelölés után beillesztett rész.

Ha a jobb egérgombra megjelenő menüben a „Paste new entry” parancsot választjuk, akkor új kódrészlettel bővíthetjük a könyvtárat. Ha bejelölve hagyjuk a „Replace selection” jelölőnégyzetet, akkor az előzőleg kijelölt szövegrész helyett kerül beillesztésre a kódrészlet. Ha töröljük a pipát, akkor a fentieknek megfelelően megadhatjuk a kijelölt szövegrész elé és mögé kerülő kódrészletet.



*F-4. ábra.  
Új kódrészlet készítése*

Ha egy kijelölt szövegrészt a vágólapra másolunk, és utána kiválasztjuk a „Paste new entry” parancsot, akkor a TextPad a vágólap tartalmát bemásolja a megnyíló ablakba. Így bonyolultabb kódrészleteket is könnyen felvehettünk a könyvtárba.

Új könyvtárat készíthetünk, ha egy könyvtár nevére (például „HTML Tags”) kattintunk a jobb egérgombbal, és kiválasztjuk a „New book” parancsot. Ekkor meg kell adni a fájl nevét és a mentés helyét (elfogadhatjuk a felajánlott mappát), majd az új kódrészletkönyvtár nevét.

A szerzők által használt könyvtárat a CD-melléklet VBScript.TCL állománya tartalmazza, amely a TextPad mappában található. Ha bemásoljuk a TextPad könyvtárai közé, akkor néhány hasznos kódrészlettel bővítjük az eredeti könyv-

tárakat. A kódrészletkönyvtárak helyét a Configure menü Preferences parancsa mutatja meg a Folders bejegyzés „Clip Library” sorában. Az itt feltüntetett mappába kell bemásolni a VBScript.TCL állományt.

## F2. A MICROSOFT SCRIPT DEBUGGER

A Microsoft Script Debugger a szkriptek készítését és hibakeresését segítő alkalmazás. Lehetővé teszi a programok megszakítását, a változók értékének lekérdezését, módosítását és a lépésenkénti végrehajtást.

Az alkalmazás telepítőprogramja a következő weblapról tölthető le:

<http://msdn.microsoft.com/downloads/list/webdev.asp>

A letöltéshez válasszuk a lista alján található „Microsoft Windows Script Debugger”-t. Ez a hivatkozás egy olyan weblapra mutat, amelyen a program Windows 98-hoz és Windows Me-hez készült változata érhető el. Innen azonban továbbléphetünk a Windows NT, 2000 és XP alatt futó változat letöltéséhez.

### A Script Debugger telepítése és indítása

Az alkalmazás telepítése nem igényel különösebb beavatkozást. Az Igen (Yes, OK) gombra való kattintással fogadjuk el a felajánlott lehetőségeket. A telepítés után általában újra kell indítani a számítógépet, amire egy párbeszédablak figyelmeztet.

Az alkalmazás ikonja nem kerül be a Start/Programok menübe. A kényelmes indításhoz célszerű parancsikont készíteni az asztalon. Ha nem változtattuk meg a telepítés helyét, akkor az Intézővel lépünk be a C:\Program Files\Microsoft Script Debugger mappába, majd a jobb egérgombbal kattintsunk az msscrdbg.exe fájlra. A felbukkanó menüből válasszuk a Küldés parancsot, célként pedig jelöljük meg az Asztalt. Így egy ikon került az Asztalra, melynek segítségével a szokásos módon indítható a program.

A Script Debuggert elindíthatjuk az Internet Explorer Nézet menüjéből a „Parancsfájl-hibakereső”, majd a Megnyitás parancs választásával is. A böngésző egy hiba bekövetkezésekor szintén rákérdez a hibakereső indítására. Ehhez előtte az Eszközök menüpontban válasszuk ki az Internetbeállítások parancsot, majd a Speciális panelen a Böngészés csoportban kapcsoljuk ki a hibakeresés tiltását a parancsfájlokban, illetve engedélyezzük az üzenet megjelenítését minden parancsfájlból.

### A hibakeresés folyamata

Hibakereséshez a fájlt a böngészőben kell megnyitni, majd át kell váltani a Debuggerre. A forráskódban úgynevezett töréspontokat (breakpoint) helyezhetünk el, melyeknél megszakad az utasítások végrehajtása. A töréspontok elhelyezése után visszaváltunk a böngészőre, majd valamilyen esemény létrehozásával (például parancsgombra való kattintással) elindítjuk a szkriptet. A töréspontnál a böngésző leállítja a végrehajtást, és átvált a Debuggerre. A hibakeresőben lekérdezhetjük, illetve módosíthatjuk az egyes változók értékét, valamint végrehajthatunk a programban nem szereplő VBScript utasításokat is.

Hibakereső üzemmódban alkalmazhatjuk a lépésenkénti végrehajtást (step into). Ekkor minden egyes utasítás után leáll a végrehajtás, és lehetőségünk van a fenti műveletek elvégzésére.

Ha egy függvény vagy eljárás utasításait nem akarjuk egyesével végrehajtani, akkor át is ugorhatjuk ezt a részt (step over). Ebben az esetben az utasítások leállás nélkül hajtódnak végre, a töréspont pedig az alprogram hívását követő utasításhoz kerül. Ha már beléptünk egy alprogramba, akkor is fel tudjuk függeszteni a lépésenkénti üzemmódot a további utasításoknál (step out). A végrehajtás szintén a hívást követő utasításnál áll le.

Ha a böngészőben frissítjük a dokumentumot, akkor egyben töröljük a benne elhelyezett töréspontokat. Ez megnehezíti az *onload* eseménykezelő vizsgálatát, mert a Debuggerre csak akkor tudunk átváltani, amikor már a dokumentum betöltése befejeződött. Az *onload* lépésenkénti végrehajtásához válasszuk az Explorer Nézet menüjében a „Parancsfájl hibakereső” almenü „Leállítás a következő utasításnál” parancsát, majd frissítsük a dokumentumot. Ugyanezt a hatást érhetjük el a Debugger megfelelő parancsával is (Break at Next Statement).

A hibakeresést a Debugger eszközeinek áttekintése után példákon keresztül mutatjuk be.

A Microsoft Script Debugger nem csak a szkriptek hibáinak felderítésére alkalmas. Új dokumentumokat készíthetünk, vagy módosíthatjuk a már meglévő weblapokat. Állományokat a File menü New parancsával hozhatunk létre, illetve az Open parancssal nyithatunk meg. Szokás szerint itt találjuk meg a mentés (Save) és a mentés másként (Save As) parancsokat is.

Megjegyezzük, hogy nyomkövetés közben a Debuggerben látható forráskódot nem tudjuk szerkeszteni. Ehhez más néven kell elmenteni a dokumentumot.

## A hibakeresés eszközei

A fájl forráskódját mutató dokumentum ablak mellett még három ablak áll a rendelkezésünkre, melyeket a View menü parancsainak vagy az eszköztár ikonjainak a segítségével nyithatjuk meg:



*Running Documents:* a böngészőben megnyitott weblapok listáját mutatja. Dupla kattintással innen is betölthetjük a forráskódot a Debuggerbe.



*Call Stack:* az éppen meghívott eljárások és függvények listáját mutatja. Ha egy alprogram végrehajtása befejeződik, akkor annak neve törlődik a listából.



*Command Window:* parancsablak. Itt lekérdezhetjük, és módosíthatjuk a változók értékét, végrehajthatunk VBScript utasításokat.

A hibakereséshez leggyakrabban a Debug eszköztár ikonjait használjuk:



*Toggle Breakpoint:* töréspont elhelyezése, illetve törlése az aktuális sorban. A töréspontnál leáll a szkript végrehajtása, és megjelenik a Debugger ablaka.



*Clear All Breakpoints:* törli az összes töréspontot a forráskódból.



*Run:* a lépésenkénti végrehajtás befejezése, a szkript futtatásának folytatása.





*Stop Debugging*: a hibakereső üzemmód befejezése.



*Break at Next Statement*: töréspont elhelyezése a következő (vagy a legelső) utasításnál. Főleg akkor használjuk, ha az *onload* eseménykezelőben keresünk hibát, vagy a szkript egy másik ablakot nyit meg.



*Step Into*: lépésenkénti üzemmód, a következő utasítás végrehajtása.



*Step Over*: alprogram meghívása esetén az utasítások folyamatos végrehajtása, majd leállítás a hívást követő utasításnál.



*Step Out*: az alprogram hátralévő utasításainak folyamatos végrehajtása, majd leállítás a hívást követő utasításnál.

A parancsok a Debug menüből, illetve a parancsok neve mellett feltüntetett billentyűkombinációkkal is elérhetők.

### Hibakeresés futási hiba esetén

A programok hibáit a 7.4. fejezetben három fő csoportba soroltuk. Szintaktikus hiba esetén nincs szükségünk nyomkövető programra, mert a böngésző kiírja a hibás sor és azon belül a karakter sorszámát. A szintaktikus hiba bármilyen szövegfájl szerkesztővel javítható.

Futási hiba esetén a böngésző leállítja a szkript végrehajtását, és rákérdez a Hibakereső indítására. Töltsük be a 3–4. példát a böngészőbe, és a párbeszédablak megjelenésekor kattintsunk az OK gombra. A hiba bekövetkezésekor (a HEAD-ben lévő szkript hivatkozik a BODY objektumra) engedélyezzük a Hibakereső indítását. A böngésző elindítja a Script Debuggert, és sárga nyíllal, illetve sárga háttérrel jelöli azt az utasítást, amelynél a hiba bekövetkezett.

Figyeljük meg, hogy a dokumentum betöltése még nem fejeződött be, nem látjuk a hibás sor után következő utasításokat! Nyissuk meg a parancsablakot, és kérdezzük meg a *Torzs.bgColor* értékét. Ehhez a tulajdonság nevét (illetve bármely változót) egy kérdőjel után kell begépelni:

```
? Torzs.bgColor
```

A megjelenő hibaüzenetből, és az előttünk lévő forráskódból világosan látható, hogy még nem létezik a *Torzs*-objektum, így nem használhatjuk a tulajdonságait!

Kattintsunk a „Stop Debugging” ikonra. Befejeződik a hibakereső üzemmód, és betöltődik a dokumentum hátralévő része. Egy szövegszerkesztővel kijavíthatjuk a forráskód hibáját.

## Logikai hibák keresése

A hibakereső programok főleg akkor lehetnek a segítségünkre, ha a szkript látszólag hiba nélkül lefut, de nem a várt eredményt szolgáltatja. Általában ezeknek a logikai hibáknak a felderítése a legnehezebb.

A CD-melléklet Függelék mappájában találjuk a 4–41. példa módosított változatát (**4–41a.htm**). A 10x10-es szorzótáblát úgy készítjük el, hogy egy-egy sor összeállításához meghívjuk a *Sor* eljárást, amely a paraméterként megadott tényezővel 1-től 10-ig megszorozza a számokat, és az eredményeket hozzáfűzi a *Lista* nevű sztringhez:

```
Sub Sor(Szam)
  For I = 1 To 10
    Lista = Lista & Szam * I & "; "
  Next
End Sub
```

A parancsgomb eseménykezelőjében a *Lista* változó inicializálása után 1-től 10-ig növekvő paraméterrel meghívjuk a *Sor* eljárást, soremelést fűzünk a kódhoz, majd megjelenítjük a sztringet:

```
Lista = ""
For I = 1 To 10
  Sor(I)
  Lista = Lista & "<BR>"
Next
ListaKi.innerHTML = Lista
```

Nem akarunk minden alprogramban deklarációs utasításokat használni, ezért a változókat a globális szkriptben deklaráljuk:

```
Dim I, Lista
```

Ha betöltjük a programot, és kattintunk a Kiír gombra, akkor szomorúan tapasztaljuk, hogy csak egyetlenegy sort látunk. Reméljük, az Olvasó már felfedezte a hibát, de próbáljuk a nyomkövető program segítségével kideríteni, miért nem kaptuk meg a teljes szorzótáblát!

### *Hibakeresés és lépésenkénti végrehajtás*

Hagyjuk megnyitva a böngészőt, és indítsuk el a Script Debuggert. Ehhez felhasználhatjuk az Internet Explorer Nézet menüjét. Mivel a ciklus nem annyiszor futott le, ahányszor vártuk, helyezzünk el egy töréspontot a ciklusmag első utasításánál:

- Sor(I)

Váltsunk vissza a böngészőre, és kattintsunk a parancsgombra. Amikor az utasítások feldolgozása elérkezik az eljáráshíváshoz, akkor leáll a végrehajtás, és megjelenik a Debugger ablaka. A sárga nyíl jelzi, hogy hol tartunk:

⇒ Sor(I)

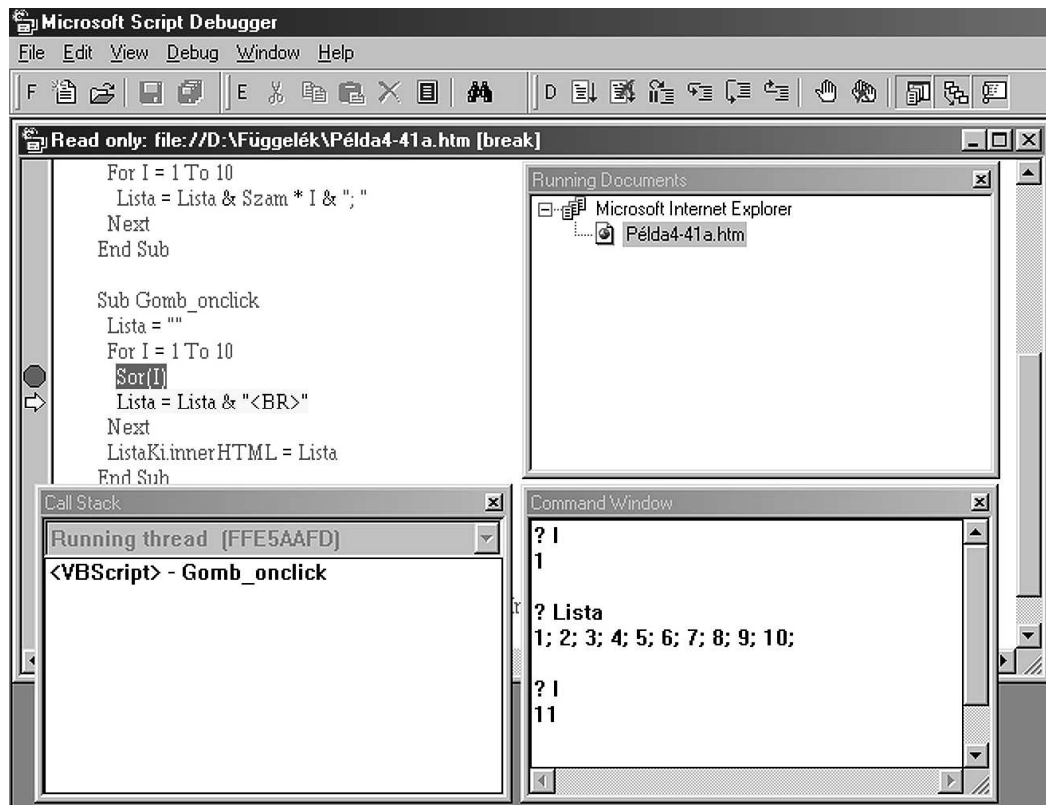
Kérdezzük meg a parancsablakban az I ciklusváltozó értékét:

? I

Válaszként 1-et kapunk, a ciklus első végrehajtásánál tartunk. A *Sor* eljárás hívása következik. Kattintsunk a „Step Over” ikonra, nem akarjuk lépésenként végigkövetni az eljárás végrehajtását. Amikor lefutott az eljárás, ismét leáll a végrehajtás. Az aktuális sort jelző sárga nyíl a

⇒      `Lista = Lista & "<BR>"`

utasításra mutat.



F-5. ábra. Logikai hiba keresése a Script Debuggerrel

Kérdezzük meg a *Lista* változó értékét:

? Lista

Láthatjuk, hogy elkészült a szorzótábla első sora, a *Sor* eljárás megfelelően végrehajtódott.

Kérdezzük meg újra a ciklusváltozó értékét:

? I

Válaszként 11-et kapunk, így érthető, hogy az eseménykezelő eljárás ciklusa befejezi az ismétléseket, és nem készül el a szorzótábla többi sora. Kiderítettük a hiba okát, az eljárás működése módosítja a (globális) ciklusváltozó értékét. Ugyanazt a globális ciklusszámlálót használja az eseménykezelő, és a *Sor* eljárás is. A hiba javítását az Olvasóra bízunk.

A hibakeresés után a „Stop Debugging” ikonra való kattintással befejezzük a lépésenkénti végrehajtást.

## A Stop utasítás és a Debug-objektum

Programfejlesztés közben töréspontot nem csak a hibakeresővel, hanem a VBScript *Stop* utasításával is elhelyezhetünk a forráskódban:

```
Stop
```

A *Stop* hatására elindul a Script Debugger, és lehetővé válik a lépésenkénti végrehajtás. Ezzel egyszerűbben végezhetünk nyomkövetést, de a végén törölnünk kell ezeket az utasításokat.

Nyomkövető üzemmódban a *Debug*-objektum *Write*, illetve *WriteLine* metódusával tudunk üzeneteket megjeleníteni a Debugger parancsablakában. A *Debug*-objektumot nem kell létrehozni, mindig a rendelkezésünkre áll. A *Debug*-objektum metódusait csak hibakereső üzemmódban hajtja végre az interpreter.

A CD-melléklet Függelék mappájában található **4-41b példában** módosítottuk az előző programot. A *Sor* eljárásban a ciklusváltozó értékét kiírjuk a Debugger parancsablakába:

```
Debug.Write("A ciklusváltozó értéke:")
For I = 1 To 10
    Debug.Write(I)
    Lista = Lista & Szam * I & "; "
Next
```

Az eseménykezelő eljárásban pedig az eljárás hívása előtt elindítjuk a nyomkövető üzemmódot:

```
Stop
Sor(I)
```

A parancsgombra való kattintás után a *Stop* utasításnál leáll a végrehajtás, és elindul a Debugger. Lépünk a következő utasításra („Step Into”), majd hajtjuk végre az alprogram utasításait („Step Over”). A parancsablakban megjelennek az eljárás ciklusváltozójának értékei.

## F3. A MICROSOFT SCRIPT EDITOR

A Microsoft Script Editor egy olyan korszerű, integrált fejlesztői környezet (IDE), amely nagymértékben segíti a HTML-kód, a Visual Basic Script (és Java Script) programok írását, szerkesztését, kipróbálását, hibakeresését.

A szintaktikus elemek különböző színnel történő kiemelése mellett számos szolgáltatást nyújt az objektumok „fogd és vidd” módszerrel történő elhelyezésére, mozgatására, megkönnyíti a tulajdonságok megadását, események hozzárendelését, az eseménykezelő eljárások elkészítését.

A fejlesztői rendszer részletes ismertetése külön kötetet igényelne. Az alábbiakban csak a legfontosabb eszközeire térünk ki, és néhány példán keresztül bemutatjuk a használatát.

### F3.1. A Script Editor telepítése és felépítése

#### A Script Editor telepítése és indítása

A program a Microsoft Office 2000, illetve Office XP programcsomag része, így a telepítéshez az Office CD-jére van szükség. A továbbiakban az Office 2000-nek megfelelő változatot ismertetjük. Az Office XP Script Editor a kisebb eltérésektől eltekintve hasonló módon használható.

A Script Editor telepítéséhez a Start/Beállítások menü segítségével nyissuk meg a Vezérlőpult ablakot, és kattintsunk duplán a Programok telepítése/törlése ikonra. Válasszuk a Programok módosítását, majd a megjelenő listában a Microsoft Office 2000 Professional-t. Kattintsunk a Módosítás gombra, majd a „Hozzáadás vagy eltávolítás” ikonra. Nyissuk ki az Office eszközök csoportját és a HTML-forráskódszerkesztőt. Kattintsunk a „Webes parancsnyelv” ikonjára, majd válasszuk „A Sajátgépről fut” lehetőséget. A Frissítés gombra való kattintás után megkezdődik a telepítés, amely nem igényel beavatkozást.

Telepítés után a program nem kerül be a Start menübe. Egyszerű indításához célszerű parancsikont létrehozni az Asztalon. Ha a telepítésnél nem módosítottuk, akkor a program a

C:\Program Files\Microsoft Visual Studio\Common\IDE\IDE98

mappában található. Kattintsunk rá az MSE.exe fájlra az egér jobb gombjával, majd válasszuk a „Küldés” parancsot, és célként jelöljük meg az Asztalt.

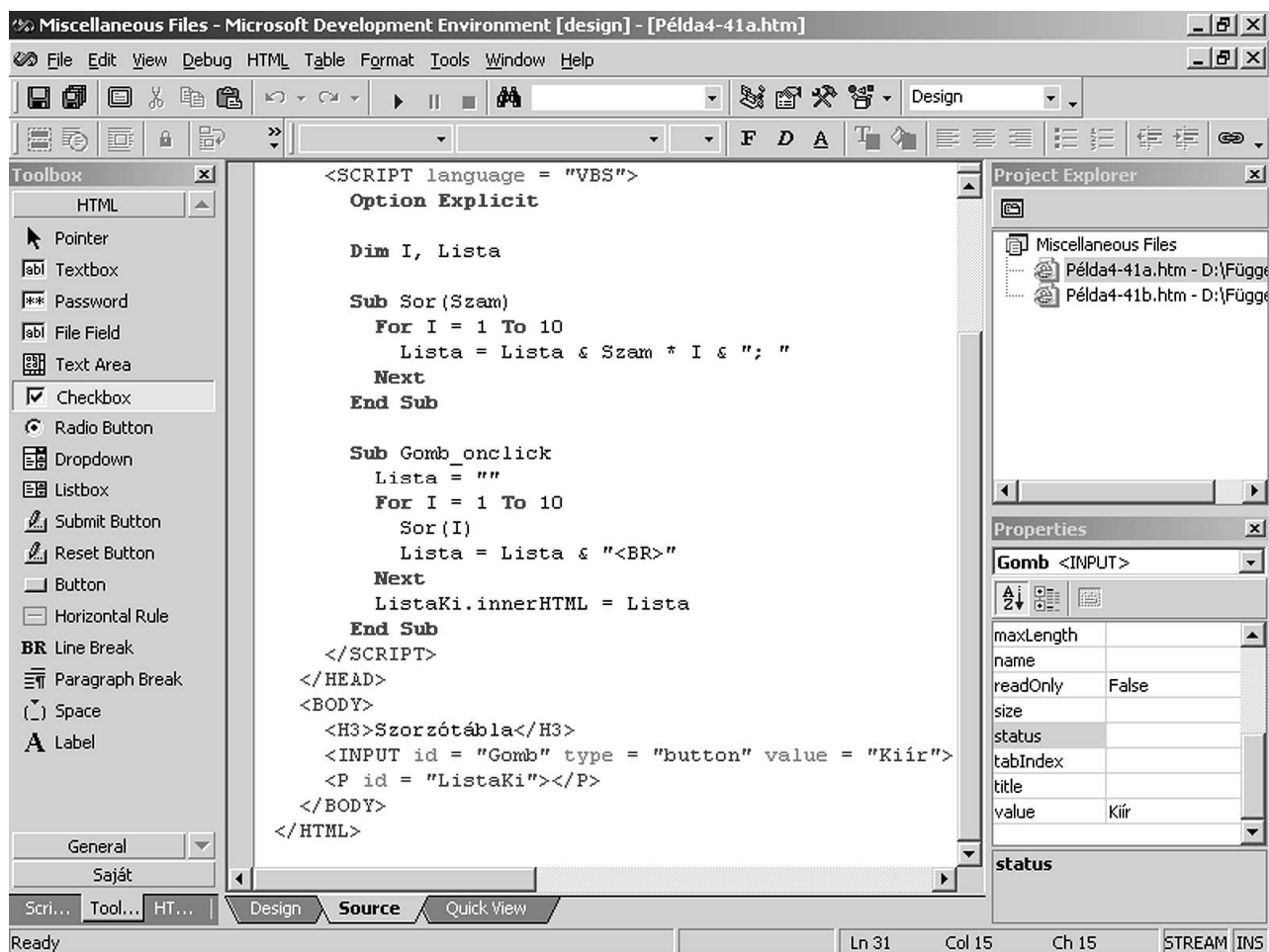
Az Office XP használata esetén a C:\Program Files\Microsoft Office\Office10 mappában az MSE7.exe állományhoz kell parancsikont készítenünk.

Megjegyezzük, hogy a Script Editor által alkalmazott sokféle billentyűparancs közül a jobb oldali Alt+C egy ablak (Call Stack) megnyitására szolgál, ezért nem tudunk & jelet írni a forráskódba. Ennek módosításához válasszuk a Tools menü Options parancsát. Az Environment csoportban láthatjuk a Keyboard alcsoportot a billentyűparancs hozzárendelésekkel. Kattintsunk a kategóriák (Categories) között a View cso-

portra, majd keressük meg a „Call Stack” parancsot (Commands), ahol láthatjuk a Ctrl+Alt+C hozzárendelést. Kattintsunk rá a „Current keys” szövegmezőben, és töröljük ki a Remove gombbal. Ha feltétlenül szükségünk van egy ilyen billentyűparancsra, akkor kattintsunk a „Press new shortcut key” szövegmezőre, majd nyomjuk le a Ctrl+Shift+C billentyűket (ezt nem használja másra a Script Editor). A hozzárendelést az Assign gombbal végezhetjük el.

## A dokumentumablak

A megnyitott fájlokat a középen elhelyezkedő dokumentumablakban látjuk. A dokumentumot különböző nézetekben jeleníthetjük meg, melyek között az ablak alján elhelyezkedő fülek segítségével válthatunk. A tervezési (Design) nézetben szerkeszthetjük a weblapot, elhelyezhetünk elemeket, megadhatjuk a tulajdonságait. A tervezési nézetben nem működnek a hivatkozások, és nem futtathatók a szkriptek. Az eszköztár Show Details (ikon) részletek megjelenítése) ikonjával ki-be kapcsolhatjuk a nem látható elemekre utaló ikonok megjelenítését (megjegyzések, szkriptek, bekezdések nyitó és záró tagja stb.).



F–6. ábra. A Microsoft Script Editor ablaka

A forráskód (Source) nézet a HTML-kódot mutatja. Itt szerkeszthetjük a szkripteket. A HTML-kód és a szkriptek kulcsszavait, az objektumok tulajdonságait, a kons-

tansokat különböző színek jelzik. A tagoló karakterek (white spaces) megjelenítését az Edit/Advanced menü „View White Space” parancsának kiválasztásával érhetjük el.

A gyorsnézet (Quick View) fül segítségével mentés nélkül megtekinthetjük, hogyan jelenik meg a weblap egy böngészőben.. A View menü „View in Browser” parancsával át is válthatunk az általunk használt böngészőre.

## **Segédablakok a Script Editorban**

A dokumentumablak mellett több segédablak támogatja a weblapok készítését és a programfejlesztést. A nem látható segédablakokat a View menüben, illetve „Other Windows” parancsánál találjuk meg.

A „Project Explorer” ablak az előzőleg használt dokumentumok listáját tartalmazza. Egy dokumentumot dupla kattintással lehet megnyitni.

A Properties (tulajdonságok) ablakban egy kiválasztott objektum tulajdonságainak értékét adhatjuk meg. Először – ponttal kezdve – a stíluselemeket látjuk. A Script Editor a forráskódba a megfelelő szintaxissal írja be a tulajdonságokat. A Properties ablak az integrált fejlesztőrendszerek egyik fontos összetevője, amely nagymértékben megkönnyíti az objektumok kezelését.

A Toolbox (eszközkészlet) segédablak több funkciót foglal magába. A HTML csoportban látható objektumokat az egérrel áthelyezhetjük a dokumentumablakba. Ehhez használhatjuk a tervezési vagy a forráskód nézetet is.

A General (általános) csoportban a már elkészített elemeket tárolhatjuk. Az egérrel áthelyezhetjük ide a dokumentumablakból a kijelölt objektumokat vagy forráskódrészleteket. A lista elemeit szintén az egérrel tehetjük vissza a dokumentumablak megfelelő helyére.

A Toolbox elemei a jobb egérgomb segítségével átnevezhetők, törölhetők, átrendezhetők. A felbukkanó menüben megjelenő „Add Tab” paranccsal új csoportot hozhatunk létre, így csoportosítani tudjuk az ide helyezett kódrészleteket.

A Toolbox ablak alján átválthatunk a „HTML Outline” (dokumentumvázlat) segédablakra. A vázlat a weblapot alkotó objektumok hierarchikus elrendezését ábrázolja. Egérkattintással kiválasztva egy objektumot a kurzor a dokumentumablakban rááll az objektumra. A „Script Outline” (szkript vázlat) nézetben a *document* és *window*-objektum mellett azon objektumok listája is megjelenik, melyeknek azonosítót adtunk. Ha a + jelre történő kattintással kinyitjuk az objektumhoz tartozó mappát, akkor az események listáját látjuk. Az eseménykezelő eljárással rendelkező események neve félkövér betűvel szerepel.

## **Az ablakok elrendezése**

A dokumentumablak mellett látható segédablakok sok helyet elfoglalnak a képernyőn. A kényelmesebb munkához többféleképpen is átrendezhetjük őket.

Az elrendezést megkönnyíti az ablakok Dockable (rögzíthető) tulajdonsága, melyet a Window menüpontban kapcsolhatunk be. A Script Editor ablakának széléhez mozgott segédablak „odatapad”, így nem takarja el a dokumentumablakot. Ha az egyik

ablakot egy másik ablak címsorára helyezzük, akkor egymásra kerülnek, és az alul megjelenő fülekkel válthatunk közöttük.

Ha szeretnénk visszatérni az ablakok eredeti elrendezéséhez, akkor válasszuk a View menü „Define Window Layout” parancsát. Jelöljük ki a Design vagy az „Edit HTML” nézetet, és kattintsunk az Apply gombra. Ha egy új nevet írunk a „View Name” szövegmezőbe, akkor az Add paranccsal az általunk létrehozott elrendezést is elmenthetjük.

Az Office XP-ben célszerű bekapcsolni az Automatikus elrejtés/megjelenítés funkciót. Ehhez kattintsunk az ablak tetején látható „Auto Hide” ikonra (☒). Ha elvisszük az egeret az ablakról, akkor az keskeny sávvá húzódik össze a képernyő szélén. Kiszélesítéséhez az egeret egyszerűen a sávban látható megnevezés fölé kell mozgatni. A folyamatos megjelenítés visszaállításához kattintsunk ismét (a kissé megváltozott) „Auto Hide” ikonra.

## **A menüsor és az eszköztárak**

A Script Editor menüi a jól ismert parancsok mellett (új fájl létrehozása, mentés, keresés stb.) számos lehetőséget bocsátanak a rendelkezésünkre. Az alábbiakban néhány fontosabb elemet emelünk ki.

A View menü „Full Screen” parancsával válthatunk át a dokumentumablak teljesképernyős megjelenítésére, majd vissza.

A Table menü parancsai leegyszerűsítik a táblázatok készítését, a sorok és a cellák elhelyezését, valamint összevonását.

A súgóban (Help) a Script Editor használatának ismertetésén túl megtaláljuk a HTML és a VBScript részletes dokumentációját is.

Az eszköztárak be- és kikapcsolásához kattintsunk a jobb egérgombbal valamelyik eszköztár üres területére. A Script Editor kezelését a Standard, a weblapok szerkesztését a Design és a HTML, a hibakeresést pedig a Debug eszköztár ikonjai segítik. A „Full Screen” eszköztár használata megkönnyíti a teljesképernyős üzemmódra történő váltást.

Az eszköztárak parancsai, ikonjai módosíthatók. A módosításához kattintsunk a jobb egérgombbal egy eszköztár üres területére, majd válasszuk a Testreszabás parancsot. A megjelenő ablakban a Parancsok fülénél látható ikonokat kitehetjük a megjelenített eszköztárakba. Célszerű kitenni egy megjelenített eszköztárba a File menü „Open File” (fájl megnyitása), illetve View menü „Full Screen” (teljesképernyős megjelenítés) ikonját. Az Eszköztárak fülénél saját eszköztárat is létrehozhatunk.

Ha a nyitott Testreszabás ablaknál rákattintunk egy megjelenített eszköztár valamelyik ikonjára, akkor a Kijelölés módosítása parancsgomb segítségével megváltoztathatjuk, sőt szerkeszthetjük is az ikont (Gombkép váltása, Gombkép szerkesztése).



## F3.2. A Script Editor használata

A Script Editor legfontosabb funkcióinak ismertetése után példákon keresztül mutatjuk be a használatát.

### Weblap létrehozása és szerkesztése

Készítsünk weblapot, melyen egy szövegmező és egy parancsgomb látható. A parancsgomb *onclick* eseménykezelője írja ki egy H3 címsorba a szövegmező tartalmát.

Új állományt a File menü „New File” parancsával hozunk létre. A megjelenő párbeszédablakban válasszuk az „Új HTML lap” sablont, majd a Megnyitás parancsot.



A Properties ablak legördülő menüjében váltsunk át a DOCUMENT-objektumra. Keressük meg a *bgColor* tulajdonságot, majd kattintsunk a három ponttal jelölt ikonra. A megjelenő párbeszédablakban válasszunk ki egy színt. Hasonló módon adjuk meg a betűk színét (*text* tulajdonság). A *title* tulajdonság értékéhez írjuk be az „MSE példa” szöveget, majd nyomjuk le az Entert.

Gépeljük be a dokumentumablakba az „Ide kerül a szövegmező tartalma.” szöveget, majd a HTML eszköztárban válasszuk a Normál stílus helyett a 3-as szintű címsort. Kattintsunk rá a szövegre, és a Properties ablakban írjunk be az (*Id*) tulajdonság értékéhez a *Cimsor* azonosítót. (Az objektum nevében nem használhatunk ékezetes magánhangzókat!) Az értékadást Enterrel zárjuk. Figyeljünk arra, hogy a segédablak tetején a <H3> megjelölés szerepeljen! Az *align* tulajdonság értékét állítsuk *center*-re (begépelés helyett a legördülő menüből is kiválaszthatjuk).

A címsor végén nyomjunk Entert, és a HTML eszköztáron állítsuk be a bekezdés balra igazítását (Align Left). A Toolbox HTML csoportjából helyezzünk el a weblapon egy Textboxot (szövegmező), és egy Butont (parancsgomb). Tegyük közéjük szóközt. Figyeljük meg a Properties ablakban, hogy a szövegmező a *text1*, a parancsgomb pedig a *button1* azonosítót kapta! Ezeket nem szükséges megváltoztatnunk. Kattintsunk rá a parancsgombra, majd a Properties ablakban írjuk át a *value* tulajdonság értékét „Button”-ról „Változtatás”-ra. Gépeljük be a szövegmező elé az „Írja ide az új címet!” szöveget.

Váltsunk át Source nézetre, és tekintsük meg az elkészült forráskódot. Néhány lényegtelen tipográfiai eltéréstől eltekintve megfelel az eddig megismert HTML-kódnak. Próbáljuk ki a Quick View nézetet is.

A további munka előtt mentsük el a fájlt.

Egy elem abszolút pozicionálásához a kiválasztása után a tervezési nézetben kattintsunk a Design eszköztár „Absolute Positioning” ikonjára (). Az egérrel a megfelelő helyre mozgathatjuk az objektumot. Ha kiválasztjuk az „Absolute Mode” ikont () , akkor a HTML Toolboxból már tetszőleges helyre áttehetjük az elemeket. Az átfedéseket a *zindex* tulajdonság segítségével határozhatjuk meg a Properties ablakban. Az abszolút pozicionálást a Format menüben is beállíthatjuk.

## Eseménykezelő eljárás készítése

Tervező nézetben kattintsunk duplán a dokumentumba helyezett parancsgombra. A Script Editor átvált forráskód nézetbe, a Toolbox helyén pedig megjelenik a Script Outline, a *button1*-objektum eseményeinek listájával. Kattintsunk duplán az *onclick* esemény nevére. A Script Editor elhelyez egy SCRIPT elemet a HEAD-be, ezen belül pedig egy *Sub* és *End Sub* utasítást az objektum és az esemény megjelölésével:

```
Sub button1 onclick

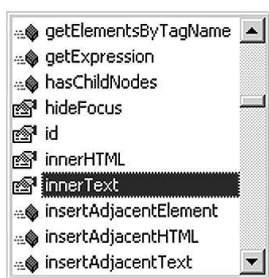
End Sub
```

A SCRIPT nyitó tagjában megadja a *language* tulajdonságot is.

Az eseménykezelő eljárásba gépeljük be a címsorobjektum azonosítóját, és írjunk utána egy pontot:

```
Sub Button1 onclick
  Cimsor.
End Sub
```

A pont beírásakor megjelenik az automatikus kódkiegészítés (IntelliSense) ablaka,



amely felsorolja az objektum összes tulajdonságát, metódusát és eseményét. Nyomjuk le az *innerText* tulajdonság kezdőbetűjét (i). A kurzor a listában az első i-vel kezdődő bejegyzésre ugrik (*id*). Lépünk a kurzormozgató billentyűvel az *innerText*-re, majd folytassuk a gépelést a tulajdonságnevet követő szóközzel és egyenlőségjellel. A Script Editor automatikusan beírja a kiválasztott tulajdonságot a forráskódba:

F-7. ábra  
Az IntelliSense  
ablaka

```
Sub Button1 onclick
  Cimsor.innerText =
End Sub
```

Ha a tulajdonságnév után nem akarunk írni semmit, akkor a tabulátor billentyű lenyomására jeleníthető meg a lista kiválasztott eleme a forráskódban.

Folytassuk a program írását a szövegmező azonosítójának (*text1*), majd egy pontnak a begépelésével. Az előzőekhez hasonlóan megjelenik a lista. Nyomjuk le a *value* szó kezdő v betűjét. A kurzor a *value* tulajdonságra ugrik, ami a tabulátor billentyű lenyomásakor bekerül a forráskódba. Ezzel befejeztük az eseménykezelő eljárás elkészítését:

```
Sub Button1 onclick
  Cimsor.innerText = text1.value
End Sub
```

A szkript működését a Quick View nézetben próbálhatjuk ki. Mentsük el a dokumentumot.

A tulajdonságok, metódusok, események listáját utólag is megjeleníthetjük, ha a kurzort az objektum nevét követő pont mögé helyezzük, és lenyomjuk a Ctrl+J billen-

tyűparancsot. A szkript más helyein a Ctrl+J-vel a VBScript utasításait, függvényeit, konstansait listázhatjuk ki.

Módosítsuk az eseménykezelő eljárást úgy, hogy csak az első öt karaktert írja ki a begépelt szövegből! Ehhez illesszük be a forráskódba a *text1.value* elé a *Left* függvénynevet, és egy kezdő zárójelet:

```
Cimsor.innerText = Left(text1.value
```

A zárójel beírásakor ismét működésbe lép az IntelliSense, megjeleníti a függvény szintaxisát a lehetséges paraméterekkel. Ezt a funkciót utólag a Ctrl+I billentyűparanccsal érhetjük el. A Ctrl+szóköz billentyűparancs pedig egy megkezdett szó esetén felajánlja annak kiegészítését az eddig használt szavak listája alapján.

A második paraméter beírásával fejezzük be a módosítást, és mentjük el a dokumentumot:

```
Sub button1_onclick  
    Cimsor.innerText = Left(text1.value, 5)  
End Sub~
```

Az így elkészített dokumentumot a CD-melléklet Függelék mappájában az **MSEpelda.htm** fájl tartalmazza.

## ActiveX objektum beillesztése

A Script Editor számos ActiveX objektummal bővíti a rendelkezésünkre álló készletet. A teljes listát a Tools menü „Customize Toolbox” parancsa jeleníti meg. Keresünk meg a „Microsoft Slider Control”-t (csúszka vezérlőelem), és tegyük pipát a jelölőnégyzetbe. A vezérlőelem bekerül a Toolboxba, ahonnan az egérrel áthelyezhetjük a weblapra.

A forráskód nézetben is magát a vezérlőelemet látjuk (DTC: design time control). A Properties ablakban azonban a HTML-tulajdonságok mellett az ActiveX objektum tulajdonságait szintén megtaláljuk (a HTML-kód PARAM elemei). A csúszka megjelenését a *TextPosition* (a szöveg helyzete), a *TickStyle* (a beosztások stílusa), a *TickFrequency* (osztásköz) és az *Orientation* (elhelyezkedés) tulajdonságok szabályozzák. Lehetséges értékeiket a Properties ablak legördülő menüiben látjuk. A csúszka szélső értékeit a *Min* és *Max* tulajdonságok adják meg. A beállított értéket a *Value* tulajdonság tartalmazza. A csúszka mozgatását a *scroll*, az egérrel történő kattintást pedig a *click* esemény jelzi.

A CD-melléklet Függelék mappájában található **Csúszka.htm** állomány szemlélteti a vezérlőelem használatát.

### F3.3. Hibakeresés a Script Editorral

A Script Editor sokrétű hibakeresési eszközzel rendelkezik. Lehetővé teszi töréspontok elhelyezését, a lépésenkénti üzem módot, a változók értékének futás közbeni lekérdezését és módosítását. Hibakereső üzemmódban célszerű bekapcsolni a Debug eszköztárat.

#### Hibakereső ablakok

A hibakereső ablakokat az eszköztár (vagy a View/Debug Windows menü) segítségével jeleníthetjük meg.

Az Immediate (közvetlen elérés) ablakban a változók értékét kérdezhetjük le, és módosíthatjuk. Lépésenkénti üzemmódban beírhatunk kifejezéseket is, melyeket a Script Editor kiértékel. A Watch (követés) ablakban egy kifejezés értékét kísérhetjük figyelemmel a szkript futása közben. A Locals (lokális változók) ablakban a végrehajtás alatt álló alprogram lokális változóinak értékét látjuk. Az Output (kimeneti) ablakot üzenetek küldésére használhatjuk fel a szkriptek futása során. Megemlíthetjük még a Call Stack (veremtár) ablakot, amelyben az aktív alprogramhívásokat, illetve a Running Documents (futó dokumentumok) ablakot, amely a megnyitott dokumentumokat (keretek, párbeszédablakok) mutatja.

#### A hibakereső üzemmód

A Script Editor hibakereső üzemmódja többféleképpen is elindítható.

A Debug menü Start parancsa betölti a dokumentumot a böngészőbe. Ekkor a Break parancs hatására az első szkript indításakor leáll a végrehajtás, és lehetővé válik a hibakeresés.

A Script Debuggerhez hasonlóan töréspontokat is elhelyezhetünk a forráskódban. Ehhez használhatjuk a Debug menü, illetve a Debug eszköztár parancsait, vagy a Source nézet bal oldali margóját, ahová egérekattintással helyezhetünk töréspontot.

Töréspontok elhelyezése esetén a Debug/Start parancssal tölthetjük be a dokumentumot egy böngészőbe. Az első töréspontnál leáll a szkript végrehajtása, és lehetővé válik a lépésenkénti végrehajtás.

A forráskódban elhelyezett Stop utasítás ugyancsak átvált a beállított hibakereső programra.

A hibakereső üzemmód indításához nincs szükség a Start parancs kiadására. A „Step Into” parancs betölti a böngészőbe a dokumentumot, majd az első utasításnál, a „Run To Cursor” parancs pedig a kurzor soránál átvált a hibakereső üzemmódba.

Hibakereső üzemmódban a „Step Into”, „Step Over” és „Step Out” parancsok meg egyeznek a Script Debugger azonos nevű parancsaival.

A hibakereső üzemmódot a Debug menü Continue, illetve End parancsával fejezhetjük be. Az előbbi az aktuális sortól folytatja a végrehajtást, az utóbbi pedig bezárja a böngésző ablakát, és visszatér a Script Editorhoz. A Restart parancs hatására a böngésző abbahagyja a szkript végrehajtását, és az elejétől kezdve futtatja az alkalmazást.

## A töréspontok tulajdonságai

A töréspontok tulajdonságait a Debug menü Breakpoints parancsával jeleníthetjük meg. A lista jelölőnégyzeteinek segítségével letilthatunk egy-egy töréspontot, azaz a törlése nélkül is megakadályozhatjuk a hibakereső üzemmód indítását. Ezt a funkciót a Debug vagy a jobb egérgomb használatakor felbukkanó menü „Disable Breakpoint” parancsával szintén elérhetjük.

Töréspontot a Breakpoints ablak Remove gombjával vagy a Debug menü „Remove Breakpoint” parancsával törölhetünk. A törlést végrehajthatjuk úgy is, hogy a Source nézet margóján rákattintunk a töréspontot jelölő piros körre.

Ha a Debug/Breakpoints ablakban kiválasztunk egy töréspontot, és rákattintunk a Properties gombra, akkor a töréspont tulajdonságait láthatjuk. A Condition mezőben megadhatunk egy kifejezést. A végrehajtás leáll, ha a kifejezés teljesül (is true) vagy ha értéke megváltozik (is changed). Ha például a *Lista* nevű változó értékének megváltozásakor szeretnénk belépni a hibakereső üzemmódba, akkor egyszerűen írjuk be a változó nevét a Condition mezőbe:

```
Lista
```

és jelöljük be az „is changed” választógombot. Az

```
I > 3 And I <= 5
```

feltétel pedig az „is true” választógomb kijelölésével,  $I = 4$  és  $I = 5$  esetén vált át a hibakereső üzemmódba. A végrehajtás folytatásához mindkét esetben válasszuk a Continue parancsot.

A Breakpoints ablakban a „Hit count” bejelölésével megadhatjuk, hogy a töréspont által kijelölt utasítás hányadik (is equal to) vagy legalább hányadik (is greather than or equal to) végrehajtásánál lépünk be a hibakereső üzemmódba. Használhatjuk a megadott érték egész számú többszöröseit (is a multiple of) is. Ha például egy ciklust minden negyedik végrehajtás után szeretnénk leállítani, akkor írunk 4-et a szövegmezőbe, és jelöljük be az „is a multiple of” választógombot.

A beállított feltételeket úgy is megtekinthetjük, hogy az egérmutatót a Source nézetben a töréspontot jelölő piros kör fölé visszük.

## A hibakereső üzemmód használata

A Script Editorral végzett hibakeresés módszere, logikája megegyezik a Script Debugger használatával, de több és kényelmesebb eszközt biztosít számunkra.

A hibakereső üzemmód használatát a CD-melléklet Függelék mappájában található **Példa4-41a.htm** segítségével mutatjuk be. Töltsük be a dokumentumot a Script Editorba, majd nyissuk meg az Immediate, a Watch és a Locals ablakokat.

Jelöljük ki a forráskódban az *I* változó nevét, és az egérrel tegyük át a Watch ablakba. Végezzük el ezt a műveletet a *Lista* változóval is.

Helyezzük a kurzort a Sub Gomb\_onclick sorába, majd válasszuk a Debug menü „Run To Cursor” parancsát. Ekkor megnyílik a böngésző ablaka, és megjeleníti a dokumentumot. A Kiír gombra való kattintással indítsuk el az eseménykezelő szkriptet.

Visszakapjuk a Script Editor ablakát, és a Source nézet margóján egy sárga nyíl jelzi a következő utasítást (`Lista = ""`). A Watch ablakban láthatjuk, hogy még egyik változó sem kapott értéket (mindkettő *Empty*).

Adjuk ki kétszer a „Step Into” parancsot. A Watch ablak jelzi a változók értékadását. Egy változó értékét úgy is megjeleníthetjük, hogy a forráskódban a változónév fölé visszük az egérmutatót.

A „Step Into” parancssal lépünk be a *Sor* eljárásba. A Locals ablakban láthatjuk az eljárás egyetlen lokális változóját, a *Szam* paraméter értékét. A „Step Into” parancssal hajtsuk végre néhányszor a ciklust, és figyeljük a Watch ablakban a változók értékét. Láthatjuk a *Lista* kialakulását. Ha kinyitjuk a Locals ablak tetején elhelyezkedő legördülő menüt, és kiválasztjuk a *Gomb\_onclick* alprogramot, akkor a forráskódban egy zöld nyíl jelzi az aktuális eljáráshívás helyét.

A „Step Out” parancssal fejezzük be a *Sor* eljárás végrehajtását, majd a „Step Over” parancssal ugorjuk a *Gomb\_onclick* eljárás ciklusának *Next* utasítására. Láthatjuk, hogy már a ciklus első végrehajtása után 11 lett a ciklusváltozó értéke, mert a *Sor* eljárás ugyanezt a változót használta. A Watch ablak Value oszlopában írjuk át 1-re az I-t, majd a Continue parancssal lépünk ki a hibakereső üzemmódból. A böngésző ablakában láthatjuk, hogy két sor készült el a szorzótáblából.

Hibakereső üzemmódban az Immediate ablakot a Script Debugger Command ablakához hasonló módon használhatjuk. Begépelhetünk értékadó utasításokat, a ? parancssal pedig lekérdezhethetjük a változók értékét.

## **A Visual InterDev**

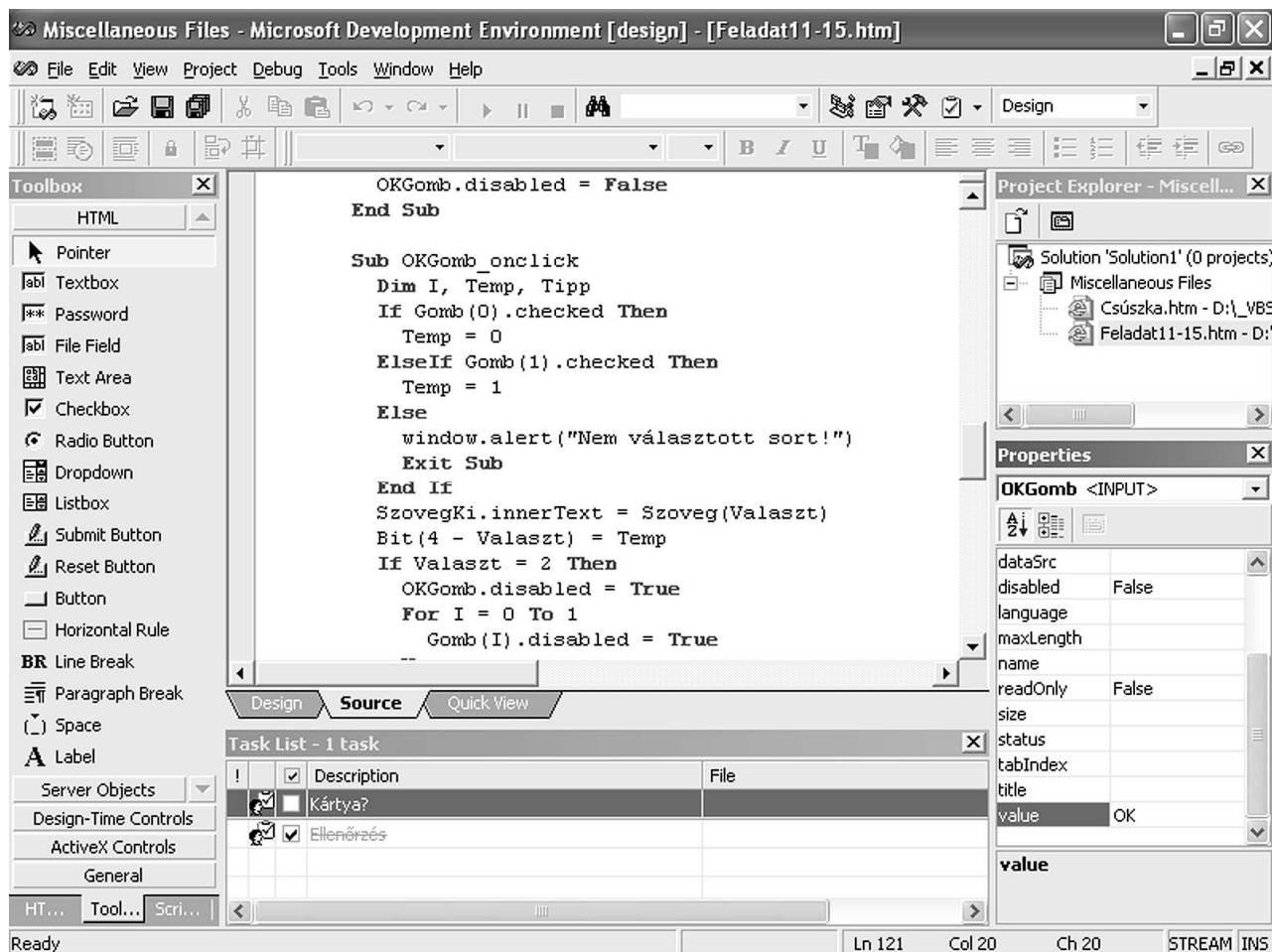
A weblapszerkesztő programokkal egyedi dokumentumokat készíthetünk. Elősegítik a hozzájuk kapcsolódó szkriptek írását, ellenőrzését, továbbá a hibakeresést. Webhelyek (web sites) létrehozásához, illetve karbantartásához már integrált fejlesztőrendszereket alkalmaznak. Egy ilyen eszköz a Microsoft Visual Studio, amely a Visual Basic, a Visual C++, a Visual J++ (Java) és a Visual FoxPro mellett a Visual InterDev-et is tartalmazza.

A Visual InterDev (Internet Development) a webes alkalmazások olyan fejlesztőrendszere, amely különböző sablonokkal, navigációs gombokkal, struktúra-diagramokkal segíti a webhelyek szervezését. Lehetővé teszi az ügyfél- és kiszolgáló oldali szkriptek készítését, hibakeresését, a fent felsorolt programnyelveken megírt komponensek beillesztését. Egységesen kezeli a webkiszolgálón létrehozott alkalmazásokat (webhálók), nyilvántartja a szerkezetet, a hivatkozásokat. Projekt-szemléletmódja támogatja a fejlesztést végző csoport tagjainak együttműködését.

A szkriptek írását tervezésidejű vezérlőelemek (DTC: design time control), azaz előre elkészített szkriptobjektumok segítik. A DTC-k lefedik a webhelyek működéséhez szükséges legfontosabb tevékenységeket. Olyan objektumokkal bővítik a dinamikus objektummodellt, melyek megkönnyítik a webkiszolgáló karbantartását, feladatainak végrehajtását, az ügyfelek kéréseinek teljesítését és nyilvántartását.

A Visual InterDev fő erőssége a webkiszolgáló és az adatbázis-kezelő rendszerek összekapcsolása. Könnyen készíthetünk olyan weblapokat, melyek egy-egy lekérdezés

eredményét jelenítik meg, vagy módosításokat hajtanak végre az adatbázison. A kapcsolatok létrehozását, az adatbázisban történő navigálást, a táblázatok összeállítását, a karbantartás műveleteit számos szkriptobjektum segíti.



F-7. ábra. A Visual InterDev 6.0 ablaka

A Visual InterDev részletes bemutatására nincs módunk. Ezzel a rövid áttekintéssel hívjuk fel az Olvasó figyelmét erre a jól használható fejlesztőeszköze. Megjegyezzük, hogy a 6.0-s változat után már a Visual Studio .Net 2003 is megjelent a piacon. A felsőfokú oktatásban részt vevő hallgatók számára mindkét változat ingyenesen elérhető és jogtisztán felhasználható a Microsoft Campus Szerződés keretében. A Campusról további információk találhatók a <http://www.campus.hu> weblapon. Innen tölthető le a Visual Studio is, de csak a Nemzeti Információs Infrastruktúra Fejlesztési Program címtartományából (a felsőoktatási intézmények számítógépes hálózataiból).

A kiszolgálóoldali programozás iránt érdeklődő Olvasó figyelmét felhívjuk az ASP.NET Web Matrix alkalmazásra, amely a Visual Studio .Net egyszerűsített változata a webes alkalmazások készítéséhez. A Web Matrix ingyenesen letölthető a

<http://www.asp.net>

weblapról, ahol részletes ismertetést is találunk róla.





## F4. FELADATGYŰJTEMÉNY

A feladatokat fejezetenként csoportosítottuk. A megoldások a CD-melléklet Feladatok mappájának megfelelő almappáiban található. Javasoljuk az Olvasónak, hogy akkor is nézze át ezeket a forráskódokat, ha sikerült önállóan elkészítenie a fájlokat.

### 2. fejezet

**2–1. feladat.** Készítsen weblapot, amely a böngésző címsorában üdvözli a felhasználót!

**2–2. feladat.** Bővítse ki az előző feladatot úgy, hogy a weblapon megjelenjen a „Tanuljunk programozni!” szöveg!

**2–3. feladat.** Készítsen weblapot, amelynek kódjában a HEAD tartalmazza a BODY-t! Figyelje meg, hogy mi történik!

**2–4. feladat.** Készítsen weblapot, amelynek kódjában a TITLE-t a BODY-ba helyezi! Figyelje meg, hogy mi történik!

**2–5. feladat.** Készítsen weblapot, amelynek kódjában a BODY tartalmazza a HEAD-et! Figyelje meg, hogy mi történik!

**2–6. feladat.** A karakterentitások felhasználásával készítse el az alábbi weblapot!

**jumbó<sup>♠</sup>, dáma<sup>♥</sup>, király<sup>♣</sup>, ász<sup>♦</sup>**

♠ 2, 3, 4, 5, 6, 7, 8, 9, 10, J, D, K, A

♥ 2, 3, 4, 5, 6, 7, 8, 9, 10, J, D, K, A

♣ 2, 3, 4, 5, 6, 7, 8, 9, 10, J, D, K, A

♦ 2, 3, 4, 5, 6, 7, 8, 9, 10, J, D, K, A

**2–7. feladat.** Készítse el az alábbi weblapot. A háttérszín világoskék, a szöveg színe barna legyen!

**Lépegető**

Lépcső Lépcső le ne ess **VIGYÁZZ!!!**

**2–8. feladat.** Készítsen olyan weblapot, amely az alábbi díszítősort tartalmazza narancssárga színben, kék háttéren:

**O<sub>o</sub>O<sup>o</sup>O<sub>o</sub>O<sup>o</sup>O O<sub>o</sub>O<sup>o</sup>O<sub>o</sub>O<sup>o</sup>O**

**2–9. feladat:** Készítsen olyan weblapot, amely a

V  
A  
K  
Á  
C  
I  
Ó

feliratot jeleníti meg az itt látható elrendezésben! A háttérszín kék, a szöveg színe sárga legyen.

**2–10. feladat.** Készítsen weblapot a PRE-objektum használatával, amely az alábbi formában, szóközökkel középre igazítva jeleníti meg a szöveget!

T A N U L J U N K

- - -

P R O G R A M O Z N I !

**2–11. feladat.** Készítsen weblapot, amelyen egy 2. szintű címsor szövegére kattintva a BODY szövegszíne pirosra, duplán kattintva pedig sötétzöldre változik!

**2–12. feladat.** Készítsen weblapot, amelyen egy 1. szintű címsor szövegére egyszer kattintva középre, duplán kattintva balra igazodik! Kezdetben a címsor szövege jobbra legyen igazítva.

**2–13. feladat.** Készítsen weblapot, amelyen ha a felhasználó a **ComputerBooks** feliratra kattint, akkor betöltődik a kiadó weblapja (<http://www.computerbooks.hu>)!

**2–14. feladat.** Készítsen weblapot, amely két 3. szintű címsort tartalmaz! Ha az első címsor szövegére kattintunk egyszer, akkor változzon meg az ablak címsorának szövege, ha a második címsor szövegére kattintunk duplán, akkor változzon meg az állapot-sor tartalma.

**2–15. feladat.** Készítsen weblapot, amely két 3. szintű címsort tartalmaz! Ha az első címsor szövegére kattintunk, akkor egy párbeszédablakban üdvözlje a felhasználót, ha második címsor szövegére kattintunk, akkor egy párbeszédablakban köszönjön el a felhasználótól, majd zárja be az ablakot.

**2–16. feladat.** Készítsen olyan weblapot, amelyre ha bárhol egyszer kattintunk, akkor az ablak mérete 600x400 pixelre változik! Ha duplán kattintunk, akkor záródjon be a böngésző ablaka.

**2–17. feladat.** Készítsen olyan weblapot, amelyen egy bekezdésben a „Sárga vagy piros legyen a háttér?” mondat olvasható! Az aláhúzott szóra kattintva a háttérszín sárgára, a dőlt betűs szóra kattintva pirosra változzon. Mindkét esetben egy párbeszédablak tájékoztassa a felhasználót az esemény bekövetkezéséről.

**2–18. feladat.** Készítsen weblapot a következő formában:

Kattintson a kívánt méretre!

**200 x 300 pixel**

**400 x 300 pixel**

**640 x 480 pixel**

**800 x 600 pixel**

***Az ablak bezárása (Kattintson duplán bárhova!)***

A megfelelő méretre kattintva változzon meg az ablak mérete, bárhová duplán kattintva záródjon be a böngésző ablaka!

**2–19. feladat.** Készítsen weblapot, amely egy középre igazított bekezdésben tartalmazza a „**Visual Basic Script**” szöveget, egy újabb középre zárt bekezdésben pedig három parancsgombot, amelyek segítségével a szöveg balra, középre és jobbra igazítható!

**2–20. feladat.** Készítsen weblapot, amely négy parancsgombot tartalmaz az ábrán látható elrendezésben! A megfelelő gombra kattintva változzon meg a „**Visual Basic Script**” felirat szöveg-, illetve az ablak háttérszíne.

## Visual Basic Script

Sárga szöveg

Kék szöveg

Rózsaszín háttér

Zöld háttér

**2–21. feladat.** Készítsen weblapot, amely három parancsgombot tartalmaz középre igazítva! A bal oldali gomb tiltsa, a jobb oldali pedig engedélyezze a középső gomb használatát. A középső parancsgombra kattintva záródjon be a böngésző ablaka.

**2–22. feladat.** Készítsen weblapot, amely öt parancsgombot tartalmaz az alábbi elrendezésben!

### Az ablak méretének változtatása

100 x 100 pixel

200 x 300 pixel

400 x 300 pixel

640 x 480 pixel

800 x 600 pixel

BEZÁRÁS

**Az ablak bezárásához duplán kattintson a BEZÁRÁS gombra!**

A megfelelő parancsgombra kattintva változzon meg az ablak mérete, illetve záródjon be a böngésző ablaka.

**2–23. feladat.** Készítsen weblapot, amely egy szövegmezőt, mellette pedig egy „Monogram” feliratú parancsgombot tartalmaz! Ha a szövegmezőbe a felhasználó begépel a monogramját, majd a parancsgombra kattint, akkor egy párbeszédablakban jelenjen meg a begépelte monogram.

**2–24. feladat.** Készítsen weblapot az ábrán látható tartalommal! A megfelelő szövegmezőbe szélesség, illetve magasság adatokat gépelve, majd a „Méretezés” feliratú gombra kattintva változzon meg az ablak mérete az adatoknak megfelelően.

**Az ablak méretének beállítása**  
Írja be az ablak szélességét!  pixel  
Írja be az ablak magasságát!  pixel

**2–25. feladat.** Készítsen weblapot, amely két szövegmezőt tartalmaz, mellettük egy-egy „Változtat” feliratú parancsgommbal! Ha a szövegmezőkbe színek angol nevét írjuk, akkor a megfelelő parancsgombra kattintva változzon meg az ablak háttérszíne, illetve egy 3. szintű címsor betűszíne.

**2–26. feladat.** Készítsen weblapot, amely egy „<I><B><U></U></B></I>” karakter-sorozatot tartalmazó szövegmezőt, mellette pedig egy „Megjelenítés” feliratú parancsgombot tartalmaz! A parancsgombra kattintva a szövegmezőbe írt szöveg jelenjen meg egy külön bekezdésben a megfelelő stílusban.

**2–27. feladat.** Készítsen weblapot az ábrán látható formában! A parancsgombok valamelyikére kattintva mindhárom bekezdés igazítása változzon a feliratnak megfelelően.

**Igazítás parancsgombokkal**  
**1. bekezdés**  
**2. bekezdés**  
**3. bekezdés**

**2–28. feladat.** Készítsen weblapot, amely egy szövegmezőt, egy „Kattintson ide” feliratú parancsgombot és „A kör sugara cm.” szöveget jeleníti meg! A szövegmezőbe egy kör sugarát gépelve, majd a parancsgombra kattintva a beírt értékkel egészítse ki a szöveget. A számok háttérszíne világoskék, betűszíne kék legyen.

**2–29. feladat.** Készítsen weblapot az ábrán látható elrendezésben!

### A háromszög adatai

Írja be a szövegmezőkbe az oldalak hosszát!

### A háromszög oldalai:

**a** =

**b** =

**c** =

A szövegmezőkbe adatokat gépelve, majd a „Megjelenítés” gombokra kattintva, a beírt adatok jelenjenek meg a szövegmezők alatt, a megfelelő helyeken.

**2–30. feladat.** Készítsen weblapot az ábrán látható tartalommal!

### Szövegmező tartalmának másolása

### A szövegmező tartalma:

A szövegmező tartalma a „Megjelenítés” gombra való kattintáskor középre igazítva jelenjen meg a felirat alatt, a „Törlés” gombra kattintva pedig tűnjön el onnan.

**2–31. feladat.** Stílusok alkalmazásával készítsen weblapot az ábrán látható formában!

### Színes vers

„Úgy született hajdan a vers az ujjam alatt,  
ahogy az Úr alkotott valami szárnyas  
fényes, páncélos, ízelt bogarat”

**Babits Mihály**

A verssorok háttere világoskék, betűszíne pedig egymás után kék, sötétkék és barna legyen.

### 3. fejezet

**3–1. feladat.** Készítsen weblapot, amely egy parancsgombot tartalmaz! A parancsgomb *onclick* eseménykezelő szkriptje egy párbeszédablakban küldjön üzenetet a felhasználónak, írjon az állapotsorba, majd a *document*-objektum *write* metódusával változtassa meg a weblap kódját.

**3–2. feladat.** Készítsen weblapot, amely egy balra zárt bekezdést tartalmaz! A bekezdés igazítását egy szkript megváltoztatja, középre, illetve jobbra zárja. Az igazítás előtt egy párbeszédablakban tájékoztassa a felhasználót a soron következő műveletről.

**3–3. feladat.** Készítsen dokumentumot színes háttérrel, amely egy parancsgombot tartalmaz! Írjon *onclick* eseménykezelő szkriptet a parancsgombhoz, amely egy 1-es és egy 2-es szintű címsort jelenít meg a weblapon. A megjelenítés előtt egy párbeszédablakkal figyelmeztesse a felhasználót a változásra.

**3–4. feladat.** Készítsen weblapot, amely két parancsgombot tartalmaz „Piros háttér”, illetve „Sárga szöveg” felirattal! A parancsgombokhoz készítsen eseménykezelő szkripteket, amelyek a megfelelő gombra való kattintás során megváltoztatják a szöveg, illetve a háttér színét.

**3–5. feladat.** Készítsen weblapot, amely egy bekezdést és három parancsgombot tartalmaz, „Balra”, „Középre”, illetve „Jobbra” felirattal! A gombokhoz készítsen eseménykezelő szkripteket, amelyek kattintásra megváltoztatják a bekezdés igazítását.

**3–6. feladat.** Írjon programot, amely a HEAD-ben deklarál három változót, *Nev*, *Lakhely* és *Született* azonosítókkal! A változóknak adjon kezdőértéket. A BODY kódjában elhelyezett szkript DIV- és SPAN-objektumok segítségével írja ki a változók értékét a következő formában:

*Nev*

Lakhelye: *Lakhely*. Születési év: *Született*

**3–7. feladat.** Írjon programot, amely a HEAD-ben deklarál négy változót, *Nev\_1*, *Nev\_2*, *Kor\_1* és *Kor\_2* azonosítókkal! A változók kapjanak értéket is. A BODY-ban lévő szkript SPAN-objektumok segítségével írja ki a változók értékét a következő formában:

Az 1. személy adatai: *Nev\_1* *Kor\_1* éves

A 2. személy adatai: *Nev\_2* *Kor\_2* éves

**3–8. feladat.** Írjon programot, amelyben két változót deklarál! A változók az *InputBox* függvény segítségével kapjanak értéket a felhasználótól. A változók értékét a *document*-objektum *write* metódusának segítségével írja ki a weblapra.

**3–9. feladat.** Írjon programot, amelyben két változót deklarál! A változók az *InputBox* függvény segítségével kapjanak értéket a felhasználótól. A változók értékét jelenítse meg egy párbeszédablakban.

**3–10. feladat.** Készítsen weblapot, amely az ábrán látható objektumokat tartalmazza! A parancsgombhoz készítsen eseménykezelő szkriptet, amely egy bekezdésben megjeleníti az oldalak hosszát a megadott mértékegységben.

### A háromszög adatai

Írja be a három oldal nagyságát!

a:

b:

c:  Mértékegység:

A háromszög oldalai: 3 méter, 4 méter és 5 méter.

**3–11. feladat.** Készítsen programot, amely kiírja a képernyőre a legkisebb és legnagyobb 15-jegyű számot, illetve a legkisebb és egy tetszőleges 16-jegyű számot! Figyelje meg a kiírt számok alakját!

**3–12. feladat.** Készítsen programot, amely kiírja az alábbi aritmetikai kifejezések értékét:

```

5 + 4 * 6 =
10 + 6 / 3 - 1 =
10 / 2 - 6 / 3 =
12 * 3 ^ 2 / 2 ^ 2 =
5 * 7 Mod 10 \ 3 =
5 * 7 \ 3 Mod 10 =
3 & 9 * 5 & 6 =

```

**3–13. feladat.** Írjon programot, amely párbeszédablakban bekéri egy kör sugarának értékét, majd kiírja a kör kerületét és területét! A kiszámított értékek csak az adatbevitel után jelenjenek meg a weblapon.

**3–14. feladat.** Írjon programot, amely egy szövegmezőben bekéri az ÁFA százalékos értékét és a nettó árat, majd kiírja a bruttó árat! Ügyeljen arra, hogy ha bármelyik értéket módosítja, a bruttó ár törlődjön.

**3–15. feladat.** Készítsen weblapot, amely tartalmaz egy szövegmezőt és két parancsgombot „Szorzás”, illetve „Törlés” felirattal! A szövegmező kezdeti értéke legyen 1. Írjon programot, amely összeszorozza az egymás után beírt értékeket, és kiírja a szorzatot. A „Törlés” gombra kattintva a szorzat törlődjön.

**3–16. feladat.** Írjon programot, amely bemutatja a *Round* függvény használatát!

**3–17. feladat.** Készítsen programot, amely bekéri két pont síkbeli koordinátáit, és kiírja a megadott pontokon áthaladó egyenes meredekségét!

**3–18. feladat.** Írjon programot, amely bekéri egy síkbeli pont és egy normálvektor koordinátáit, majd kiírja a megadott ponton áthaladó, megadott normálvektorú egyenes egyenletét, illetve irányszögét!

**3–19. feladat.** Készítsen programot, amely bekéri egy számtani sorozat első elemét és különbségét, majd a megadott elemszámig kiírja az elemek összegét!

**3–20. feladat.** Írjon programot, amely bekéri egy mértani sorozat első elemét és hányadosát, majd kiírja a sorozat megadott sorszámú elemét, illetve a megadott elemszámig az elemek összegét!

**3–21. feladat.** Készítsen programot, amely bekéri egy háromszög három oldalának hosszát, és a Heron-képlettel kiszámítja a háromszög területét!

**3–22. feladat.** Írjon eseménykezelő eljárást a *window*-objektum *onload* eseménye segítségével, amely megjelenít egy üzenetablakot „Most készül a weblap!” üzenettel, majd megjelenít egy középre igazított 2. szintű címsort, alatta pedig egy jobbra igazított bekezdést!

**3–23. feladat.** Írjon eseménykezelő eljárást, amely gombnyomásra véletlenszerűen változtatja a háttér- és a szöveg színét!

A véletlen színt előállító kódot (VéletlenSzin.vbs) csatolja a dokumentumhoz.

**3–24. feladat.** Készítsen weblapot körhenger felszínének és térfogatának kiszámítására, amely három parancsgombot tartalmaz „Adatok beolvasása”, „Felszín” és „Térfogat” feliratokkal! A parancsgombokhoz írjon eseménykezelő eljárásokat, amelyek az *InputBox* függvény segítségével elvégzik az adatok (sugár és magasság) beolvasását, majd kiszámítják a henger felszínét és térfogatát.

**3–25. feladat.** Írjon közös eseménykezelő eljárást, amely egy parancsgombra vagy egy bekezdésre kattintva átformázza a weblapot!

**3–26. feladat.** A 3–20. feladatot mentse el .hta kiterjesztéssel! Az ablak tulajdonságait állítsa be a következőképpen: használjon párbeszédablak-keretet, a dokumentum kerete megemelkedő és bemélyedő legyen, használjon ikont, tiltsa le a „Kis méret” gombot és a görgetősávot, az ablak mérete pedig teljes képernyő legyen.

## 4. fejezet

**4–1. feladat.** Írjon programot, amely bekér egy számot, megvizsgálja az előjelét, és ha a szám nem negatív, akkor kiírja a négyzetgyökét!

**4–2. feladat.** Készítsen programot, amely 1 és 100 között „gondol” egy számot, amire lehet tippelni! A beírt értéktől függően írja ki, hogy a tipp túl nagy, túl kicsi, vagy éppen talált. Számolja a próbálgatásokat, és írja ki a tippek sorozatát is.

**4–3. feladat.** Írjon programot, amely bekér két pozitív egész számot, és kiírja, hogy a kisebb szám osztója-e a nagyobbknak!

**4–4. feladat.** Készítsen programot, amely bekér egy pozitív egész számot, és megvizsgálja, hogy számjegyek száma háromnál kevesebb, több, vagy pontosan három!

**4–5. feladat.** Írjon programot, amely bekér két számot, majd átírja az előjeleket a nagyobb abszolút értékű szám előjelenek megfelelően!



**4–6. feladat.** Készítsen programot, amely bekéri egy vizsgázó pontszámát, amelyet a következő módon minősít:

0 és 50 között: „Nem felelt meg”  
 51 és 100 között: „Megfelelt”  
 101 és 150 között: „Kiválóan megfelelt”.

(0-nál kevesebbet, vagy 150-nél több pontot nem lehet elérni.)

**4–7. feladat.** Készítsen weblapot a következő elrendezésben:

**Születési dátum ellenőrzése**

Írja be az idei évszámot!

Írja be a születési dátumot (a hónapot is számmal adja meg)!

év

hónap

nap

Egy parancsgombra kattintva egy eseménykezelő eljárás értékeli ki, hogy a megadott születési dátum érvényes-e (1880 előtti születési évet ne fogadjon el). A program azt is figyelje, hogy minden szövegmezőbe számérték kerüljön. Érvénytelen dátum esetén, annak megfelelően, hogy az „év”, a „hónap” vagy a „nap” a rossz, jelölje ki a megfelelő szövegmező tartalmát.

**4–8. feladat.** Írjon programot, amely bekéri három szakasz hosszát, és eldönti, hogy a megadott szakaszok háromszöget alkotnak-e!

**4–9. feladat.** Írjon programot, amely bekéri egy pont koordinátáit, és kiírja, hogy a pont melyik koordinátatengelyen, hányadik síknegyedben, illetve az origóban helyezkedik el!

**4–10. feladat.** Készítsen programot, amely megkérdezi a kitevő értékét, majd az alapot 1-től 20-ig változtatva egymás alá kilistázza az adott kitevőjű hatványok értékét! A kiírást formázva végezze, a dokumentumhoz csatolja a *Formaz.vbs* kódot.

**4–11. feladat.** Írja ki az egész számokat 1-től 100-ig egy 10x10-es táblázatban úgy, hogy az egymást követő számok ne egy sorba, hanem egy oszlopba kerüljenek! A formázáshoz használja a *Formaz.vbs* fájlban definiált függvényt.

**4–12. feladat.** Készítsen programot, amely bekér egy pozitív egész számot, és megvizsgálja, hogy prímszám-e!

**4–13. feladat.** Írjon programot, amely bekér egy pozitív egész számot, és a megadott számig megkeresi az összes prímszámot!

**4–14. feladat.** Készítsen weblapot, amely megjeleníti az alábbi menüt:

### Menüvezérelt program

Menü

1. Színes háttér
2. Színes szöveg
3. Üzenet
4. Ablak bezárása

Írjon programot, amely egy parancsgombra való kattintáskor egy párbeszédablakban bekéri a választott menüpont sorszámát, majd végrehajtja a menüpontnak megfelelő utasítást!

**4–15. feladat.** Készítsen weblapot, amely megjeleníti az alábbi menüt:

### Függvényérték-számolás

Menü

1. Sqr(x): négyzetgyök függvény
2. Sgn(x): előjel függvény
3. Abs(x): abszolút érték függvény
4. Int(x): egészrész függvény

Írjon programot, amely egy parancsgombra való kattintáskor párbeszédablakban bekéri a választott menüpont sorszámát és az x értékét, majd kiszámolja a megfelelő függvényértéket!

**4–16. feladat.** Készítsen weblapot, amely megjeleníti az alábbi menüt:

### Függvényérték-számolás

Menü

1. Sqr(x): négyzetgyök függvény
2. Sgn(x): előjel függvény
3. Abs(x): abszolút érték függvény
4. Int(x): egészrész függvény

Választás:

x értéke:

Írjon programot, amely egy parancsgombra való kattintáskor megvizsgálja, hogy megfelelő-e választott menüpont sorszáma! Ha nem, akkor jelölje ki a „Választás” szövegmező tartalmát. Érvényes választás esetén írja ki az x-hez tartozó függvényértéket.

**4–17. feladat.** Készítsen programot, amely bekér két egész számot, és kiírja a legkisebb közös többszörösüket! A legkisebb közös többszöröst a következő algoritmussal határozza meg: addig adja a bekért kisebbik számhoz önmagát, amíg a másik számnál nagyobbá nem válik. Majd ugyanezt teszi a másik számmal is, amíg az előző összegnél nagyobbá nem válik. Ezt a folyamatot ismétli, amíg a hozzáadásokkal egyenlő értéket nem kap! Ez lesz a legkisebb közös többszörös.

**4–18. feladat.** Írjon programot, amely bekér két egész számot, és kiírja a legnagyobb közös osztójukat! Ennek meghatározásához a nagyobbik számot maradékosan ossza el a kisebbel, majd végezze el a maradékos osztást a kapott hányadossal és a maradékkal, egészen addig, amíg 0 maradékot nem kap (euklideszi algoritmus). Az utolsó nem 0 maradék a legnagyobb közös osztó.

## 5. fejezet

**5–1. feladat.** Készítsen programot, amely 1 és 100 közötti véletlen számokkal feltölt egy tömböt, majd a tömbelemeket vesszővel elválasztva kiírja egymás mellé! A tömbelemek számát párbeszédablakban kérje a felhasználótól. A program egy „Feltölt és listáz” feliratú gombra kattintva induljon.

**5–2. feladat.** Írjon programot, amely 1 és 100 közötti véletlen számokkal feltölt egy meghatározott elemszámú tömböt, és megjeleníti az elemeket! Egy „Rendez” feliratú parancsgombra kattintva a program rendezze csökkenő sorrendbe a tömbelemeket, majd ismét írja ki őket a képernyőre.

**5–3. feladat.** Készítsen programot, amely párbeszédablakban bekéri egy tömb maximális indexének értékét, majd 1 és 100 közötti véletlen számokkal feltölti azt, és kilistázza a képernyőre! Egy „Kiválasztás” feliratú gombra kattintva a program írja ki a páros tömbelemeket, és azok számát.

**5–4. feladat.** Írjon programot, amely betűkkel ír ki egy 100 000-nél kisebb pozitív egész számot!

**5–5. feladat.** Készítsen weblapot, amely 4 parancsgombot tartalmaz 100, 200, 250, 300 pixel feliratokkal! Írjon programot, amely ha valamelyik gombra kattintunk, akkor a megfelelő méretre változtatja a parancsgombok szélességét.

**5–6. feladat.** Oldja meg objektumváltozó bevezetésével az előző feladatot!

**5–7. feladat.** A 4–16. feladatban szereplő weblapot bővítse ki egy „Színezés” feliratú parancsgommbal, amelyre kattintva a különböző objektumok szöveg-, illetve háttérszíne véletlenszerűen változzon meg! Csatolja a dokumentumhoz a VeletlenSzin.vbs fájlt.

**5–8. feladat.** Írjon programot, amely bekér egy  $x$  értéket, és egy táblázat első sorában megjeleníti az  $x$ ,  $x+1$ , ...,  $x+9$  számokat, a második sorában pedig az első sor celláiban lévő számok reciprokát! A reciprok értékek számításához készítsen függvényt.

**5–9. feladat.** Készítsen weblapot az ábrán látható elrendezésben és objektumokkal!

3. szintű címsor színes háttérrel

Ez egy bekezdés.

Ez a SZÓ egy SPAN-objektumban van!

Ezt a sort egy DIV-objektum tartalmazza.

Szövegmező:

Kattintson a gombra!

Háttér színezés

Írjon programot, amely ha a „Háttér színezés” feliratú gombra kattintunk, akkor az objektumokból alkotott kollekciók segítségével a hátteret felváltva sárga és narancs színűre színezi.

**5–10. feladat.** Írjon programot, amely tárolja és megjeleníti a következő adatokat:

Név:	Születési év:	Születési hely:
László Zsolt	1951	Debrecen
Kiss István	1963	Budapest
Dékány Endre	1971	Szeged
Szabó László	1972	Győr
Nagy Leó	1966	Pécs

A táblázat alatt egy szövegmezőben kérjen be egy nevet. Ha a név szerepel a táblázatban, akkor írja ki a nevet, és alá a születési adatokat, egyébként írja ki, hogy nincs ilyen személy.

**5–11. feladat.** Írjon programot, amely tárolja és megjeleníti az 5–10. feladat adatait! A program egy parancsgombra való kattintásra rendezze név szerint ábécé sorrendbe a táblázat sorait.

**5–12. feladat.** TILI-TOLI játék. Készítsen weblapot, amely megjeleníti a következő táblázatot:

7	8	9	10
6	1	2	11
5	4	3	12
*	15	14	13

A játék célja, hogy a 15 szám a bal felső saroktól kezdve, sorfolytonosan növekvő sorrendbe kerüljön. A táblázat alá helyezzen el egy szövegmezőt. Ebben annak a cellának a számát kell megadni, amelyet mozgatni szeretnénk. Csak olyan cella mozgatható, amelynek egyik szomszédja a csillaggal jelölt cella. Az állapotsorban legyen látható a mozgítások száma. Írjon programot, amely egy parancsgombra kattintva a szövegmezőbe írt értéknek megfelelő cella tartalmát felcseréli a csillagot tartalmazó celláéval.

**5–13. feladat.** Készítsen weblapot az ábrán látható formában!

Pontok koordinátái		
Pontok	1. koordináta	2. koordináta
1. pont		
2. pont		
3. pont		
4. pont		

Hányadik pont (1, 2, 3, 4)?

Hányadik koordináta (1, 2)?

Értéke:

Írjon programot, amely a pont sorszámának és egyik koordinátájának megadása után, egy parancsgombra kattintva, a táblázat megfelelő cellájába írja a megadott értéket.

**5–14. feladat.** Készítse el az ábrának megfelelő weblapot!

Változó cella	
1. sor, 1. cella	1. sor, 2. cella
2. sor, 1. cella	2. sor, 2. cella

Kattintson a megfelelő hivatkozásra!

[1. sor, 1. cella](#)   [1. sor, 2. cella](#)  
[2. sor, 1. cella](#)   [2. sor, 2. cella](#)

A táblázat alatt elhelyezkedő, aláhúzott feliratokhoz rendeljen hivatkozást. Egy felírra kattintva a program egy párbeszédablakban kérje be a táblázat megfelelő cellájának értékét, majd írja be a megadott helyre. A párbeszédablakban jelenítse meg a sor és a cella sorszámát is.

**5–15. feladat.** Írjon programot, amely létrehoz egy dinamikus tömböt, egy párbeszédablakban beolvassa a maximális indexet, majd bekéri a tömbelemeket! Az utolsó tömbelem beírása után jelenjen meg a listájuk weblapon.

**5–16. feladat.** Írjon programot, amely beolvas egy pozitív egész számot ( $N$ ), majd létrehoz egy  $(N+1) \cdot (N+1)$  elemből álló dinamikus tömböt, amelyet 1 és 100 közötti véletlen számokkal tölt fel! A tömbelemeket táblázatban jelenítse meg. (Az  $N$  legfeljebb 30 lehet.)

**5–17. feladat.** Egészítse ki az előző feladat programját a főátlóban található elemek listázásával! (Azok az elemek helyezkednek el a főátlóban, melyek első indexe megegyezik a második indexszel.) A megjelenítést egy parancsgomb *onclick* eseménykezelője végezze.

**5–18. feladat.** Írjon programot, amely személyi adatokat olvas be (név, életkor), és tárol egy dinamikus tömbben! Az adatokat kor szerint növekvő sorrendben listázza ki egy dinamikus bővülő táblázatban.

**5–19. feladat.** Egészítse ki az 5–54. példa kódját a tanulók jegyeinek módosításával! Egy parancsgombra való kattintás esetén a program kérdezzen rá a névre, majd listázza ki a jegyeket, és nyújtson lehetőséget a megváltoztatásukra. Készítsen egy Törlés gombot is, melynek segítségével egy tanulót lehet törölni a táblázatból.

## 6. fejezet

**6–1. feladat.** Írjon programot, amely kiszámolja a  $\pi$  közelítő értékét a

$$\pi = 4 \cdot (1 - 1/3 + 1/5 - 1/7 + 1/9 - \dots)$$

képlet alapján! A program kérje be a relatív hibakorlátot.

**6–2. feladat.** Készítsen programot, amely bekér egy pozitív számot (*Szam*) és a relatív hibakorlátot, majd intervallumfelezéssel kiszámolja a megadott szám négyzetgyökének közelítő értékét! Az első intervallum a  $[0, Szam]$  legyen. Vegye az intervallum felezőpontjának a négyzetét, és hasonlítsa össze a számmal. Attól függően, hogy melyik a nagyobb, vegye az előző intervallum megfelelő felét, és ismétlje meg az eljárást, amíg a relatív hibakorlát a megadott értéknél kisebb nem lesz. A relatív hibakorlátot az intervallum hosszának és a felezőpont x-koordinátájának a hányadosa adja meg (negatív előjel esetén vegyük az abszolút értékét).

**6–3. feladat.** Írjon programot, amely intervallumfelezéssel meghatározza egy függvény zérushelyét! Ehhez meg kell adni egy olyan intervallumot, amelyen belül csak egyetlen zérushely található, és amelynek a végpontjaiban felvett függvényértékek ellentétes előjelűek. Felezzük meg az intervallumot, és válasszuk ki azt a felét, amelynek végpontjaiban továbbra is ellentétes előjelűek a függvényértékek. Ismétljük meg az eljárást addig, amíg a relatív hibakorlát egy megadott értéknél kisebb nem lesz. A relatív hibakorlátot az intervallum hosszának és a felezőpont x-koordinátájának a hányadosa adja meg (negatív előjel esetén vegyük az abszolút értékét).

**6–4. feladat.** Készítsen programot, amely bekéri egy kör középpontjának koordinátáit és a sugarát, illetve egy tetszőleges pont koordinátáit! A program döntse el, hogy a pont hogyan helyezkedik el a körhöz viszonyítva. Az összehasonlítást a relatív hibakorlát segítségével végezze.

**6–5. feladat.** Írjon programot, amely bekér egy nevet (vezeték- és keresztnév, szóközzel elválasztva), majd kiírja a név monogramját! A program kezelje a kétjegyű mássalhangzókat is.

**6–6. feladat.** Készítsen programot, amely a beírt szöveget madárnyelven jeleníti meg (lásd a 6–34. példát)! A program kezelje a nagybetűket, és a hosszú magánhangzók ismétlésénél először a rövid párjukat írja ki (például: í helyett iví, ő helyett övő stb.).

**6–7. feladat.** Írjon programot, amely bekér egy szöveget, és kiírja a benne szereplő magánhangzók számát!

**6–8. feladat.** Készítsen programot, amely bekér egy szöveget és két karaktert! A program cserélje le szövegben az egyik megadott karakter összes előfordulását a másik megadott karakterre.

**6–9. feladat.** Írjon programot, amely bekér egy szöveget, majd a benne szereplő kisbetűket nagybetűkre, a nagybetűket pedig kisbetűkre cseréli!

**6–10. feladat.** Készítsen programot, amely bekér egy személyi számot, majd kiírja az adott személy nemét és születési dátumát!

**6–11. feladat.** Írjon programot, amely kiírja az aktuális dátumot, és bekér egy születési dátumot év, hó, nap felbontásban! A program döntse el, hogy a megadott dátum érvényes születési dátum-e. (1880 előtti időpontot ne fogadjon el.) Vizsgálja meg, hogy a szövegmezők nem maradtak-e üresen, számértéket, illetve érvényes adatot tartalmaznak-e. Üzenetablakban figyelmeztessen, hogy melyik adat rossz, és jelölje ki a hibás szövegmező tartalmát.

**6–12. feladat.** Készítsen programot, amely kiírja hogy egy szövegmezőben megadott számú nap elteltével mi lesz a dátum, és ez a hét melyik napjára fog esni! A program jelenítse meg az elvégzett műveletet is (aktuális dátum + napok száma).

**6–13. feladat.** Írjon programot, amely bekér egy dátumot, és kiírja, hogy adott év múlva milyen napra esik a megadott dátum!

**6–14. feladat.** Készítsen programot, amely bekéri egy háromszög oldalainak hosszát, majd egy megfelelően paraméterezett eljárás kiszámolja a háromszög területét és területét!

**6–15. feladat.** Írjon programot, amely bekéri egy sorozat első elemét és különbségét vagy hányadosát aszerint, hogy a sorozat számtani vagy mértani! A hozzájuk tartozó parancsgombok eseménykezelői hívják meg a megfelelő sorozat adott sorszámú elemét kiszámító függvényt. A függvényeknél használjon érték szerinti paraméterátadást.

**6–16. feladat.** Alakítsa át a 6–3. feladat programját úgy, hogy a zérushelykeresést egy megfelelően paraméterezett függvény végezze!

**6–17. feladat.** Készítsen egy *Töröl.vbs* fájlt, amely a *Torol(Index, Tomb)* eljárás kódját tartalmazza. A függvény a paraméterként megadott dinamikus tömbből törölje ki a megadott indexű elemet, és ennek megfelelően csökkentse a tömb méretét. Készítsen olyan programot, amely egy véletlenszámokkal feltöltött tömbből a *Torol* eljárás meghívásával kitörli a felhasználó által kijelölt indexű elemet.

**6–18. feladat.** Írjon programot, amely derékszögű háromszögek átfogóit számolja ki a befogók ismeretében! A program töltsön fel két, befogókat tartalmazó tömböt véletlen számokkal, majd a CD-melléklet Fejezet03 mappájában lévő *Háromszög.vbs* fájlban lévő függvény segítségével írja ki az átfogók hosszát. Az eredményeket táblázatos formában, esztétikusan jelenítse meg.

## 7. fejezet

**7–1. feladat.** Készítsen weblapot, amely egy verset tartalmaz! Ha egy verssor első szavára kattintunk, akkor változzon meg a sor betűszíne, ha a sor többi részére kattintunk, akkor pedig a verssor háttérszíne. A színeket tíz szín közül válassza ki véletlenszerűen.

**7–2. feladat.** Írjon programot, amely egy parancsgombra kattintva egy új SPAN-objektumot helyez el egy bekezdésben, „Kattintson ide!” szöveggel! Ha erre a szövegre kattintunk, akkor változzon meg a szöveg tartalma és színe.

**7–3. feladat.** Készítsen programot, amely tíz darab különböző színű parancsgombot jelenít meg, alattuk pedig egy nagy négyzetet! Egy parancsgombra kattintva változzon meg a négyzet hátterének a színe a parancsgombnak megfelelő színre.

**7–4. feladat.** Írjon programot, amely egy bekezdésobjektum *onmousedown* eseménykezelőjében a szöveget jobbra igazítja, az *onmouseup* eseménykezelőjében a betűket kékre színezi, az *onclick* esemény hatására pedig a háttérszínt pirosra állítja! Az *onclick* bekövetkezéséről üzenetablakban is tájékoztassuk a felhasználót.

**7–5. feladat.** Készítsen weblapot, amely egy 1. szintű címsort és egy bekezdést tartalmaz, benne egy SPAN-objektummal! Írjon programot, amely figyeli az egérmutató pozícióját az objektumokhoz képest (felette van vagy elhagyta), és ezt kiírja az állapotsorba. Ha az egérmutató a címsor felett helyezkedik el, akkor a betűszín kékre, a bekezdés felett pirosra, a SPAN-objektum felett pedig a háttér sárgára változzon. A módosítás csak arra az objektumra vonatkozzon, amely fölé került az egérmutató.

**7–6. feladat.** Írjon programot, amely az egér görgetőgombjának hatására folyamatosan változtatja az ablak méretét!

**7–7. feladat.** Módosítsa az 5–12. feladat Tili-toli játékprogramját úgy, hogy a cellák cseréje a cellára való kattintásra történjen! Csak érvényes cellára kattintva cserélje fel a cellákat.

**7–8. feladat.** Írjon programot, amely ha egy szövegmezőbe karaktereket gépelünk, akkor a billentyű lenyomására a szövegszínt kékre, a háttérszínt világoszöldre, a billentyű felengedésére a szövegszínt zöldre, a háttérszínt pedig világoskékre változtatja!



**7–9. feladat.** Készítsen weblapot, amely világoskék háttér előtt, középre igazítva, egymás alatt jeleníti meg a „Sárga”, „Piros”, „Kék”, „Zöld” feliratokat úgy, hogy a szavak első betűje megfelelő színű legyen! Írjon programot, amely a feliratok első betűjének lenyomására a háttérszínt a megfelelő színre változtatja.

**7–10. feladat.** Írjon programot, amely ha egy szövegmezőbe kisbetűt gépelünk, akkor azt nagybetűként, ha nagybetűt gépelünk, akkor pedig kisbetűként jeleníti meg! A többi karakter maradjon változatlan.

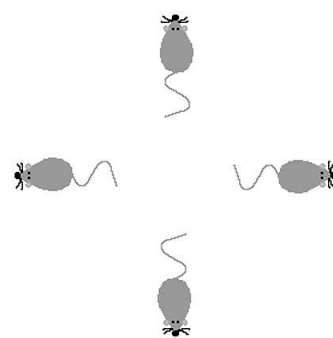
**7–11. feladat.** Készítsen programot, amely csak a magyar ábécé betűit fogadja el egy szövegmezőbe való gépelésnél! Más billentyű lenyomására ne reagáljon.

**7–12. feladat.** Írjon programot, amely egy szövegmezőbe írt adatokat az Enter billentyű lenyomására sorszámmal ellátva, egymás alatt megjeleníti a weblapon! A program üres szövegmezőre ne reagáljon.

## 8. fejezet

**8–1. feladat.** Készítsen weblapot, amely négy különböző irányba néző egeret ábrázoló képet tartalmaz az ábrán látható elrendezésben! Az elhelyezéshez használjon táblázatot.

**8–2. feladat.** Írjon programot, amely a b (bal), j (jobb), f (fel), l (le) billentyűk lenyomására a kijelölt irányba néző egeret jeleníti meg! Használja a Fel.gif, Le.gif, Bal.gif és Jobb.gif képeket.



8–1. feladat

**8–3. feladat.** Készítsen weblapot, amely egy 2x2-es táblázat celláiban az „Apró nőszirm.jpg”, „Tavaszi hérics.jpg”, „Sárga hegyiternye.jpg”, és „Tavaszi pityérfű.jpg” képeket tartalmazza! Valamely képre való kattintáskor a program üzenetablakban kérdezze meg a kép szélességét és magasságát, majd a megfelelő képet a táblázat mellett jelenítse meg a megadott méretben.

**8–4. feladat.** Készítsen weblapot, amely egy 2x2-es táblázat celláiban az „Apró nőszirm.jpg”, „Tavaszi hérics.jpg”, „Sárga hegyiternye.jpg”, és „Tavaszi pityérfű.jpg” képeket tartalmazza! Valamely képre kattintva tapétázza ki a weblap háttérét az adott virág képének halványított változatával („Apró nőszirmH.jpg”, „Tavaszi héricsH.jpg”, „Sárga hegyiternyeH.jpg”, „Tavaszi pityérfűH.jpg”).

**8–5. feladat.** Készítsen weblapot, amelyen egy 2·2-es táblázat celláinak háttere a négyféle virág halványított képe!

**8–6. feladat.** Írjon programot, amely egy szövegmezőben megadott idő elteltével bezárja a böngésző ablakát!

**8–7. feladat.** Készítsen programot, amely egy pénzérme feldobását szimulálja (váltogatja a Fej.gif és Írás.gif képek megjelenítését)!

**8–8. feladat.** Írjon programot, amely a „Csóvál” gomb hatására felváltva jeleníti meg az Egér1.gif és Egér2.gif képeket! Az animációt egy „Leáll” gomb állítsa meg.

**8–9. feladat.** Készítsen programot, amely gombnyomásra két labdát (Labda.gif) mozgat egymással szemben a képernyő bal szélétől a jobb széléig (faltól falig), illetve vissza, amíg egy parancsgombbal le nem állítjuk! Kezdetben az egyik labda a képernyő bal, a másik pedig a jobb szélén várakozzon.

**8–10. feladat.** Írjon programot, amely egy bolygót (Szaturnusz.gif) mozgat a Nap (Napocska.gif) körül!

**8–11. feladat.** Készítsen weblapot, amelyen négy választógomb látható „Treff”, „Káró”, „Kör” és „Pikk” feliratokkal! Valamelyik gombra kattintva jelenjen meg az adott színből egy véletlenszerűen kiválasztott lap. (A Kártya mappa tartalmazza az 52 kártyalap képét.)

**8–12. feladat.** Oldja meg az előző feladatot lista használatával!

**8–13. feladat.** Írjon programot, amely a megfelelő választógombra kattintva az 5-ös, a 6-os vagy a 7-es LOTTO számait sorsolja! Ügyeljen arra, hogy a kisorsolt számok különbözőek legyenek.

**8–14. feladat.** Készítsen programot az 5-ös LOTTO kitöltéséhez! A weblap jelölőnégyzetekkel rajzolja ki a lottószelvényt, és tegye lehetővé 5 szám kitöltését. A tippelés után a program sorsoljon ki 5 különböző számot, majd írja ki a találatok számát.

**8–15. feladat.** Írjon programot, amely egy listából választott félkövér, dőlt, aláhúzott vagy normál stílus hatására módosítja egy bekezdés szövegének stílusát! A program engedje meg egyszerre több stílus választását is.

**8–16. feladat.** Készítsen programot, amely egy pénzérme feldobását szimulálja (váltogatja a Fej.gif és Írás.gif képek megjelenítését)! A felhasználó választógombok segítségével tippelhesse meg az eredményt (fej vagy írás). A játék egy parancsgombra való kattintással induljon, majd véletlen számú váltás után írja ki a program, hogy a játékos nyert vagy veszített.

**8–17. feladat.** Írjon programot, amely kiszámítja az  $\text{Int}(x)$ ,  $\text{Sgn}(x)$ ,  $\text{Abs}(x)$  vagy  $\text{Sqr}(x)$  függvények értékét! A függvényt választógombokkal jelölhesse ki a felhasználó. Az  $x$  értékét szövegmezőben kérje be. Ügyeljen az adatbevitel ellenőrzésére.

**8–18. feladat.** Készítsen programot, amely egy űrlapon bekér egy e-mail címet, amit csak akkor fogad el, ha szerepel benne a @ karakter! Az Elküld gombra kattintva a program töltse be a Válasz.htm fájlt, amely dekódolja az üzenetet, kiírva a dekódolás lépéseit. Egy Vissza gombra kattintva térjen vissza az eredeti űrlapot tartalmazó web-laphoz.

**8–19. feladat.** Írjon programot, amely sütiben tárolja az elektronikus LOTTO öt tippjét! A program csak megfelelő számokat fogadjon el. Hiba esetén jelölje ki a szövegmező tartalmát, és írja ki, hogy mi a hiba. Az Értékelés gombra kattintva töltse be a

Sorsol.htm fájl, amely kiolvassa a sütiből a tippeket, sorsol, majd kiírja a találatok számát. A játékot a Vissza gombra kattintva lehessen megismételni.

**8–20. feladat.** Készítsen olyan weblapot, amely lejátszik egy listából kiválasztott zeneszámot! (Hangfájlokat a c:\Windows\Media mappában talál.)

**8–21. feladat.** Írjon programot, amely megjelenít egy Excel-táblázatot a weblapon! A szkript vizsgálja meg az első oszlop celláit, és az első üres cella esetén fejezze be a beolvasást. Minden egyes sort az első üres celláig jelenítsen meg.

**8–22. feladat.** Készítsen programot, amely megjelenít egy Excel-táblázatot a weblapon! A táblázat módosítandó cellájára kattintva, a program egy párbeszédablakban kérje be a cella új értékét, majd mentse el a módosított táblázatot. A Frissítés gombra kattintva jelenítse meg újra a táblázatot.

**8–23. feladat.** Írjon programot, amely az 5–19. feladatban a jegyek módosítását egy modális párbeszédablak segítségével végzi!

**8–24. feladat.** A 7–7. feladat Tili-toli játékához készítsen egy parancsgombra való kattintáskor megjelenő nem modális párbeszédablakot, amely ismerteti a játékszabályt!

## 9. fejezet

**9–1. feladat.** Egészítse ki a 9–1. példa programját úgy, hogy a heti sorsolásokat tartalmazó tömb bővíthető legyen az újabb húzások eredményével!

**9–2. feladat.** Alakítsa át az 5–32. példát úgy, hogy kétdimenziós tömb helyett a személyek adatait háromelemű, egydimenziós tömbök tárolják, melyek egy újabb, egydimenziós tömb elemeit alkotják!

**9–3. feladat.** Alakítsa át a 9–2. példát úgy, hogy a maximálisnál eggyel nagyobb sorszám beírása esetén újabb diák jegyeivel bővíthessük a listát!

**9–4. feladat.** Írjon programot, amellyel borkatalógus készíthető! Minden borról két adat szerepeljen a nyilvántartásban: a fajtája és az évjárata. A feladatot tömböt tartalmazó tömb segítségével oldja meg.

**9–5. feladat.** Készítsen egyirányú, rendezett listát, amelybe új adatot lehet beilleszteni, és létező adatot lehet törölni! A weblapon jelenítsen meg egy Tömörít feliratú gombot, amelynek *onclick* eseménykezelőjében a törölt listaelemeket a tömb végén összegyűjti, majd elhagyásukkal csökkenti a tömb méretét.

**9–6. feladat.** Írjon olyan programot, amely a rendezést a tömbelemek cserélgetése nélkül végzi el! A program egy tömbben tárolt kételemű tömbök első elemeibe olvasson be neveket, majd a tömbök második eleme által alkotott mutatók segítségével, listaként rendezze a névsort.

**9–7. feladat.** Készítsen kétirányú listát, amelyben a listaelemek két mutatót tartalmaznak, egyet az előző, egyet pedig a következő elemre! Írjon programot, amely kiírja a lista adatait az egyik vagy a másik irányban haladva végig a mutatókon.

**9–8. feladat.** Készítsen kétirányú rendezett listát, amelynek adatokkal való feltöltését a felhasználó végezheti! Egy adat bevitele után az aktuális lista elemei jelenjenek meg növekvő, illetve csökkenő sorrendben is.

**9–9. feladat.** Készítsen angol–magyar szótárt! A szótárba angol szavakat tudjunk felvenni, magyar megfelelőikkel együtt. A program adjon lehetőséget a szavak törlésére.

**9–10. feladat.** Készítsen olyan szótárprogramot, amely választhatóan angol–magyar, illetve magyar–angol sorrendben listázza ki a szavakat! A sorrendet egy parancsgomb-ra való kattintással tudjuk megváltoztatni. A változtatásnál ábécé szerint rendezze a listát. A felhasználónak legyen módja a szótár bővítésére is.

**9–11. feladat.** Készítsen olyan programot, amely rákérdez egy véletlenszerűen kiválasztott angol szó magyar fordítására! Rossz tipp esetén kérdezze meg újra az angol szó jelentését. A program nyújtson lehetőséget az általa ismert szópárok bővítésére.

**9–12. feladat.** Írjon programot, amely borok adatait (fajta, szín, évjárat) tartja nyilván egy szótárobjektumban! A borok fajtáját használja kulcsként, – egyforma értékek nem fordulhatnak elő. A kulcsokhoz tartozó értékeket egy-egy tömb tárolja, melynek elemei megadják a bor színét és évjáratát. Az adatokat táblázatban jelenítse meg! Legyen módunk a lista bővítésére, egy-egy bor törlésére, az adatok módosítására és fajta szerinti rendezésére.

**9–13. feladat.** Készítsen programot, amely egy szöveg madárnyelvre való fordítását (lásd a 6–34. példát) halmaz segítségével oldja meg!

**9–14. feladat.** Írjon programot, amely halmaz segítségével végzi az 5-ös, 6-os, illetve 7-es LOTTO sorsolását!

**9–15. feladat.** Készítsen olyan weblapot, amely az egérrel való kattintás helyén megjelenít egy véletlenszerűen változó kártyalapot vagy egy dobókockát! Ha valamelyik képre kattintunk, akkor tűnjön el a képernyőről. Az ablakban tetszőleges számú kártyalapot, illetve dobókockát tudjunk létrehozni.

**9–16. feladat.** Készítsen olyan weblapot, amely a kattintás helyén megjelenít egy egeret, amely jobbra-balra mozog az ablakban! Ha az egérre kattintunk, akkor tűnjön el a képernyőről. Az ablakban tetszőleges számú egeret tudjunk létrehozni.

**9–17. feladat.** Alakítsa át az 5–41. példát úgy, hogy az adatokat egy modális párbeszédablakban kérdezze meg! A párbeszédablak egy objektum tulajdonságaiként adja vissza a begépelt értékeket.

**9–18. feladat.** Készítsen programot a következő játékhoz! Három kalóz véletlenszerűen választ 40–40 számot 10 és 99 között. Az a kalóz nyer, aki a legtöbbet eltalálja a többiek által választott számok közül. A program írja ki a választásokat, és értékelje az eredményt.

## 10. fejezet

Az alábbi feladatok általában a C:\Gyakorlás mappában hoznak létre fájlokat, ezért a megoldás betöltése előtt hozzuk létre ezt a mappát! A megoldások által használt szövegfájlok a CD-melléklet \Feladatok\Feladat10\Szövegfájlok mappájában találhatók. A módosításra kerülő fájlokat (például a 10–11. feladatban) először át kell másolni a merevlemezre.

**10–1. feladat.** Írjon programot, amely bekéri egy mappa elérési útját, és ha a mappa létezik, akkor kiírja a tulajdonságait!

**10–2. feladat.** Készítsen programot, amely egy megadott mappát egy megadott helyre másol vagy áthelyez!

**10–3. feladat.** Írjon programot, amely egy adott fájlt töröl, vagy megadott helyre másol, illetve áthelyez!

**10–4. feladat.** Készítsen programot, amely egy megadott mappában az előfordulás sorrendjében sorszámot illeszt az eredeti fájlnevek elé!

**10–5. feladat.** Írjon programot, amely egy megadott mappa htm kiterjesztésű állományait úgy nevezi át, hogy a kiterjesztésük hta legyen!

**10–6. feladat.** Egészítse ki az előző feladatot úgy, hogy a program a kiterjesztés megváltoztatása mellett a HEAD nyitó tagja után egy egyszerű HTA:Application elemet is illesszen be a fájlba (például Szövegfájlok\HTA.txt)!

**10–7. feladat.** Készítsen „vírust”! A hta fájl indítás után figyelmeztetés nélkül törölje a C:\Gyakorlás mappa tartalmát. Ha a mappa nem létezik, akkor a program hibaüzenet nélkül érjen véget.

**10–8. feladat.** Írjon programot, amely pontok koordinátái olvassa be egy szövegfájlból (Szövegfájlok\Pontok.txt), és jeleníti meg egy táblázatban! A szövegfájl első sora a pontok számát tartalmazza, további sorai pedig egy-egy pont koordinátáit. A két koordinátát szóköz választja el a soron belül.

**10–9. feladat.** Módosítsa az előző feladatot úgy, hogy a program kérje be egy újabb pont koordinátáit, majd számolja ki, hogy ettől a ponttól melyik pont van a legtávolabb!

**10–10. feladat.** Írjon programot, amely személyek adatait (vezetéknév, keresztnév, születési év, születési hely) fájlból olvassa be (Szövegfájlok\Személyek.txt). Egy személy adatai egy sorban, szóközzel elválasztva helyezkednek el. A vezetéknév és a keresztnév között is szóköz van. Nem tudjuk előre a rekordok számát, ezért a beolvasásnál figyelje a program az *EndOfStream* jelet.

**10–11. feladat.** Egészítse ki az előző feladat programját úgy, hogy a személyi adatokat fájlba menti, felcserélve a születési évet és a születési helyet!

**10–12. feladat.** Módosítsa úgy az előző feladatot, hogy az adatokat táblázatosan megjelenítő HTML-fájl készüljön a szövegfájlból! Az állomány elkészítése után a program töltsse is be a weblapot.

**10–13. feladat.** „Titkosítson” egy szövegfájlt (Szövegfájlok\Szöveg.txt), például helyettesítsen minden karaktert a hárommal nagyobb ANSI-kódú karakterrel! A titkosított szöveg legyen fájlba menthető.

**10–14. feladat.** Írjon programot, amely visszafejti az előző feladat által titkosított fájlt!

**10–15. feladat.** Módosítsa úgy az 5–19. feladatot, hogy a tanulók adatait szövegfájlba tudjuk menteni! A szövegfájlban egy sor egy tanuló nevét és jegyeit tartalmazza, egymástól egy-egy szóközzel elválasztva. A program az ablak bezárásakor szükség esetén kérdezzen rá a mentésre.

**10–16. feladat.** Az előző feladatot módosítsa úgy, hogy a tanulók adatai fájlból olvashatók, és fájlba menthetők legyenek! A program a betöltéskor ellenőrizze a C:\Gyakorlás\tanulók.txt állomány létezését. Ha van ilyen fájl, akkor olvassa be, és listázza ki a benne található adatokat.

**10–17. feladat.** Az előző feladatot alakítsa át úgy, hogy a felhasználó tallózhasson a háttértárakon, megkeresve a megfelelő adatfájlt, amit gombnyomásra megnyithat (Szövegfájlok\tanulók.txt)!

**10–18. feladat.** Készítsen olyan weblapot, melynek segítségével Microsoft Office alkalmazásokat (Word, Excel, PowerPoint) lehet elindítani! A menüt hivatkozásokkal és VBScript függvényhívásokkal hozza létre. A program adjon hibaüzenetet, ha nem nyitható meg egy alkalmazás.

**10–19. feladat.** Módosítsa a 10–25. példa programját úgy, hogy többoldalas táblázatok esetén írjon ki oldalszámot a táblázatok alá!

**10–20. feladat.** Írjon programot, amellyel egy 24-bites színmélységű, BMP kiterjesztésű képfájlt lehet sötétíteni vagy világosítani! A módosítás mértékét a felhasználó adja meg százalékban. 100 %-nál kisebb érték sötétebb, nagyobb érték pedig világosabb képet eredményezzen.

## 11. fejezet

**11–1. feladat.** Készítsen egy 10000 elemből álló tömböt, melyet 1 és 99 közé eső véletlen egész számokkal tölt fel! Írjon programot, amely meghatározza az elemek átlagát (számtani közepét), megkeresi a legkisebb és a legnagyobb elemet, illetve megszámlolja, hogy hány elem kisebb, és hány elem nagyobb 50-nél.

**11–2. feladat.** Vizsgálja meg, hogy a VBScript véletlenszám-generátora mennyire szabályos dobókockát szimulál! Írjon programot, amely 6000 dobást végez (1 és 6 közé eső véletlenszámokkal tölt fel egy 6000 elemű tömböt), majd megszámlolja, hányszor kapott 1-est, 2-est, 3-ast stb. Az eredményt jelenítse meg táblázattal.

**11–3. feladat.** Határozza meg, hogy 1000 pénzfeldobás esetén hány elemű a leghosszabb csak fejből, illetve csak írásból álló sorozat! A dobások szimulálásához egy 1000-elemű tömböt töltsön fel véletlenszerűen választott nullákkal (fej) és egyesekkel (írás).

**11–4. feladat.** Írjon játékprogramot, amely megpróbálja kitalálni, hogy a felhasználó fejet (F) vagy írást (I) választott-e! A program figyelje a felhasználó egymás után következő választásaiban az alábbi sorozatok előfordulását:

```
FFFF...  
IIII...  
FIFIFI...  
FIIIFII...
```

Ha valamelyik sorozatot felismeri, akkor ennek megfelelően tippelje meg a következő választást. Ha a felhasználó választása nem felel meg egyik sorozatnak sem, akkor a program véletlenszerűen tippeljen. A felhasználó meglepően sokszor fog veszíteni, ha nincs tudomása a gép által követett algoritmról! (1956-ban – kissé módosított, a véletlennek nagyobb szerepet adó algoritmussal – önálló elektronikus gépet építettek erre a játékra.)

**11–5. feladat.** Írjon programot, amely összehasonlítja a megismert rendezési módszerek végrehajtási idejét! A rendezéseket három tömbön hajtsa végre. Az elsőnél véletlenszerűen választjuk meg az elemeket, a második tömb nagyság szerint növekvő, a harmadik pedig csökkenő elemekből álljon. A program táblázatosan jelenítse meg a végrehajtási időket.

**11–6. feladat.** Módosítsa az előző feladatot úgy, hogy az egyes rendezéseknél végrehajtott összehasonlítások és cserék számát jelenítse meg a táblázatban!

**11–7. feladat.** Írjon programot, amely kártyalapokat oszt a felhasználónak a römihez! Jelenítsen meg a képernyőn 13 véletlenszerűen választott lapot, majd rendezze a kártyákat a beillesztéses algoritmussal.

**11–8. feladat.** Készítsen programot, amely véletlenszerűen kiválaszt a csomagból 10 kártyát, majd úgy rendezi el a lapokat, hogy előre kerüljenek a piros színűek, mögéjük pedig a feketék!

**11–9. feladat.** Két szövegfájlban egész számokból álló, rendezett sorozatot tárolunk (Sorozat1.txt, Sorozat2.txt). Írjon programot, amely egy harmadik szövegfájlban szintén rendezve összefésüli a két sorozatot! Az összefésülést és a mentést olvasás közben, folyamatosan végezze el anélkül, hogy tömbökben tárolná a teljes sorozatokat.

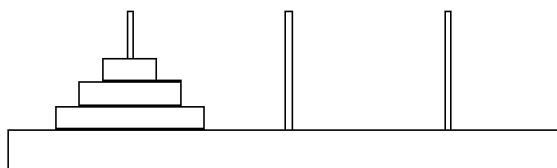
**11–10. feladat.** Írjon rekurzív programot, amely egy számtani vagy mértani sorozat megadott indexű elemét számolja ki az első elem és a különbség, illetve a hányados segítségével!

**11–11. feladat.** Készítsen rekurzív programot, amely az első tömbelemhez viszonyított nagyság szerint válogatja szét a tömb elemeit! Az első elemnél nem nagyobb elemek a tömb elejére, a nála nagyobb elemek a tömb végére kerüljenek!

**11–12. feladat.** Írjon programot, amely rekurzív algoritmussal kilistázza egy mappa tartalmát a benne lévő almappákkal együtt! Az eljárás vázlata:

```
Eljárás Listáz (Mappa)
  a mappában lévő fájlok listázása
  Ha van almappa Akkor
    Ciklus minden almappára
      Listáz (Almappa)
    Ciklus vége
  Elágazás vége
Eljárás vége
```

**11–13. feladat.** Írjon rekurzív programot a Hanoi-torony nevű játékra, melyben egy rúdon egymásra rakva csökkenő sugarú, lyukas korongok helyezkednek el! Feladatunk a korongok áthelyezése egy másik rúdra úgy, hogy egyszerre csak egy korongot mozgathatunk, és kisebb korongra nem tehetünk nagyobbakat. A megoldáshoz egy harmadik rúd is a rendelkezésünkre áll.



**11–14. feladat.** Módosítsa a sakktáblán 8 vezért elhelyező programot (11–24. példa) úgy, hogy az egyik vezér helyét a felhasználó adhassa meg!

**11–15. feladat.** Utoljára – rendhagyó módon – egy kész program megfejtése a feladat. A CD-mellékleten található program 16 kártyából négy kérdés alapján kitalálja a felhasználó által választott lapot. Milyen algoritmussal jut el a megoldáshoz? Annyit elárulunk, hogy a kártyák keverése csak a felhasználó figyelmének elterelését szolgálja.



# IRODALOMJEGYZÉK

- Angster Erzsébet: Programozás tankönyv  
(Ötödik, átdolgozott kiadás. Kiadja a szerző, 1997)
- Angster Erzsébet: Objektumorientált tervezés és programozás Java (4KÖR Bt., 2001)
- Csákány Antal – Dr. Vajda Ferenc: Játékok számítógéppel  
(Műszaki Könyvkiadó, 1980)
- Hillier, Scot: Inside Microsoft Visual Basic, Scripting Edition (Microsoft Press, 1996)
- Holzner, Steven: Web Scripting with VBScript  
(M&T Books, A Subsidiary of Henry Holt and Company, 1996)
- Jamsa, Kris – Klander, Lars: Tippek a Visual Basichez  
(Kossuth Kiadó, Budapest, 1998)
- Jamsa, Kris – Lalani, Suleiman „Sam” – Weakley, Steve: A web programozása  
(Kossuth Kiadó, Budapest, 1997)
- Kerningham, B.W. – Plauger, P.J.: A programozás fortélyai  
(Műszaki Könyvkiadó, Budapest, 1982)
- Kuzmina Jekatyerina – Dr. Tamás Péter – Tóth Bertalan:  
Programozzunk Visual Basic rendszerben! (ComputerBooks, Budapest, 2003)
- Microsoft Visual Basic 6.0 Programozói kézikönyv  
(Microsoft Press, Park Könyvkiadó, Budapest, 1998)
- Microsoft Visual InterDev 6.0 Programmer's Guide (Microsoft Press, 1998)
- Microsoft Visual InterDev 6.0 Web Technologies Reference (Microsoft Press, 1998)
- Nagy Péter: JavaScript 1.2 kézikönyv (Kiskapu Kft., Budapest, 1998)
- Revoly András: A JavaScript (Panem Kft., Budapest, 1998)
- Szlávi Péter – Zsakó László: Módszeres programozás  
(Műszaki Könyvkiadó, Budapest, 1986)
- Windows Script Technologies VBScript Reference (Microsoft Corporation, 2001,  
<http://msdn.microsoft.com/downloads/list/webdev.asp>)



# TÁRGYMUTATÓ

&	59
&nbsp;	25
:=	71, 333
<!--	29
-->	29
8 vezér	354

## A, Á

A-objektum	29
ablakobjektum	21
Abs	73
absolute	249
abszolút pozicionálás	377
accessKey	263
action	263
ActiveMovie	273
ActiveX komponensek	273
ActiveX objektum beillesztése	379
adatbeviteli hibák	72
adatmodell	287
Adatszerkezetek	287
adattag	300
hatóköre	304
add	261
Add	293
afterBegin	271
afterEnd	271
aktuális mappa	320
aktuális paraméter	81
alapértelmezett érték	189
alapértelmezett metódus	306
alaphelyzet gomb	258
alapobjektumok	21
alert	35
algoritmus	11, 333
elemi	333
strukturált	134
algoritmus-vezérelt program	17
align	27, 153
all	145
állapotsor	38
állománylista	258
alprogram	84
altKey	235
altLeft	235
always	329
And	111
ANSI-kód	10, 209

Application	277
AREA-objektum	267
argumentum	81
aritmetikai műveletek sorrendje	65
aritmetikai operátorok	64
Array	143, 288
Asc	209
AscB	211
ASCII-kód	9
AscW	211
assembler	12
assembly nyelv	11
asszociatív tömb	293
AtEndOfLine	323
AtEndOfStream	323
Atn	73
auto	284
automatikus kódkiegészítés	378
automatikusan működő alkalmazás	325
automatizmus komponensek	273, 276
autostart	272
azonosító	20

## B

B-objektum	28
back track	351
background	252
background-color	45
background-image	252
background-position	252
backgroundPositionX	252
backgroundPositionY	252
background-repeat	252
bájt	9
BASIC	14
beforeBegin	271
beforeEnd	271
behavior	285
beillesztéses rendezés	341
bekezdésobjektum	26
belső cookie-k elfogadása	268
bevitel ellenőrzése	105, 206
bezárás (objektumok)	299
bgColor	23
bgProperties	252
bináris fájl feldolgozása	327
bináris keresés	337
bináris kód	9

bit	9
blokkobjektum	43
BMP	327
body	286
BODY-objektum	21
Boolean	186
border	153, 246
border-color	246
borderColor	154
Böhm-Jacopini tétel	134
BR-objektum	26
breakpoint	367
buborékos rendezés	340
button	37, 232, 258
BUTTON-objektum	263
ByRef	216
Byte	186
ByVal	216

## C

Call	83, 218
Call Stack	380
cancelBubble	36, 228
Case Else	107
CBool	188
CByte	188
CCur	188, 200
CDate	188, 213
Cdbl	188
cellaobjektum	153
cellPadding	154
cells	156
Cells	277
cellSpacing	154
center	27, 280
CENTER-objektum	44
change	275
checkbox	258
checked	258
ChooseColorDlg	276
Chr	209
ChrB	211
ChrW	211
ciklus	120
egymásba ágyazás	126
előírt menetszámú	120
előtesztelő	128
feltételvezérelt	127
háultesztelő	128
léptető, számláló	120
végtelen	125

ciklusfej	120
ciklusmag	120
ciklusváltozó	120
címobjektum	22
címsorobjektum	27
CInt	188
cirkuláris lista	292
Class ... End Class	300
Class_Initialize	306
Class_Terminate	307
classid	273, 274
Clear	240
clearInterval	255
clearTimeout	253
click	275, 379
clip	284
CLng	188
close	34
Close	277, 322
codebase	276
color	45
commandLine	92, 211
CompareMode	294
compiler	13
confirm	87
Const	68
content	22
cookie	267, 268
CopyFile	320
CopyFolder	319
Cos	73
Count	296, 318
createElement	261, 271
CreateFolder	319
CreateObject	276
createPopup	285
CreateTextFile	322
CSng	77, 188
CStr	188
ctrlKey	235
ctrlLeft	235
Currency	186, 200
CurrentDirectory	320

## Cs

csúszka vezérlőelem	379
---------------------	-----

## D

Date	186, 213
DateCreated	320
DateSerial	214
dátum megadása	212
Debug	372
debugger	357
decimális számrendszer	9
default	284
Default	306
definiáló egyenlőségjel	333
deklaráció	54
deleteCell	167
DeleteFile	320
DeleteFolder	319
deleteRow	167
Description	240
Design View	374
DHTML	46
Dialog Helper	276
dialogArguments	280, 316
dialogHeight	280
dialogLeft	280
dialogTop	280
dialogWidth	280
Dictionary	293
Dim	54
dimenzió	137
dinamikus HTML objektummodell	46
dinamikus tömb	160
dinamikus tömbelemek	288
disabled	38, 42
display	286
DIV-objektum	43
Do ... Loop	128
Dockable	375
document	21
dokumentálás (programé)	357
dokumentum objektummodell	45
dokumentumablak	374
dokumentumvázlat	375
DOM	45
Double	186
Drive	317
DriveLetter	318
Drives	317
DriveType	318
DTC	379, 382

## E, É

edge	280
eldöntés	334
elemi algoritmusok	333
elérési út	29
teljes	247
eljárás	83
elnevezése	83
készítése	84
megszakítása	117
paraméterei	83
eljárastag	300
elküld gomb	258
előjelfüggvény	73
előletesztelő ciklus	128
ElseIf	103
EMBED-objektum	272
Empty	190
End If	96
EndOfLine	321
EndOfStream	321
Erase	161
Err.Number	240
Err-objektum	240
Error	186
értékadó utasítás	56, 66
értékes jegy	196
érvényességi idő	268
beállítása	269
érvényességi kör	268
és művelet	111
igazságtáblázat	112
escape kód	265
esemény	223
eseménybuborék	35, 227
leállítása	36
eseménykezelés	17, 30, 86
parancsgommbal	37
szkriptekkel	50
eseménykezelő eljárások	30, 85
esemény-modell	223
eseményvezérelt program	17
event	36, 50, 223
e <sup>x</sup>	73
Excel táblázatok	277
Execute	114, 331
Exists	295
Exit	
Do, For, Function, Sub	117
Exp	73
expires	269

## F

fájlobjektum	319
fájlrendszerobjektum	317
faktoriális	346
False	38, 55
fejlesztői dokumentáció	357
fejobjektum	21
feladat-specifikáció	355
felbukkanó ablak	285
felhasználói dokumentáció	357
feltételes elágazás	93
feltételvezérelt ciklus	127
fgColor	26, 32
Fibonacci-sorozat	348
file	258, 326
File	317
File System Object	317
Files	317, 318
filter	286
Fix	73
fixed	252
fixpontos mennyiség	194
focus	108
fókusz	109
Folder	317
FolderExists	319
Folders	317
folyamatábra	95
folyamatjelző	123, 283
folyamatjelző komponens	274
font-size	45, 280
font-style	280
font-weight	280
for	50
For ... Next	121
For Each ... In	146
fordítóprogram	12, 13
FORM-objektum	263
formális paraméter	81
FormatCurrency	201
FormatNumber	76
Formaz	82
formázott megjelenítés	75
forms	264
formula	277
forráskód	13
forráskód nézet	374
FreeSpace	318
FSO	317
Function	79
futási hiba	240

függvény	73
meghívása	74
paraméterei	74
függvényhívás	
a HTML-kódból	157

## G

gépi kód	11
get	264
getAttribute	192
GetDrive	318
GetFile	319
GetFolder	318
GetLocale	214
GetRef	226
globális szkript	90
globális változók	88
GMT formátum	269
görgetősáv vezérlőelem	275
grafikus felhasználói felület	10

## Gy

gyorsbillentyű	263
gyorsnézet	375
gyorsrendezés	349

## H

H-objektum	27
halmaz	297
elemek felvétele	297
halmazműveletek	297
hanggenerálás	272
háromállapotú konstans	76
hatékonyság (programé)	358
hatékonyság növelése	358
hatókör	88
hátultesztelő ciklus	128
HEAD-objektum	21
height	154, 246
helyiértékes megjelenítés	82
hexadecimális kód	9
hibakeresés	
Script Debuggerrel	367, 369
Script Editorral	380
hibakereső ablakok	380
hibakereső program	357
hibakereső üzemmód	380
hibakezelés	
fájlműveleteknél	324

hidden	58, 167, 258, 266
hide	286
hivatkozások	28
href	29
hta	91
HTA:Application	91
HTML	19
HTML Outline	375
HTML-alkalmazás	91
HTML-állomány készítése	326
HtmlDlgSafeHelper	276
HTML-kód hibái	23
HTML-objektum	21
http-equiv	23
Hypertext Markup Language	<i>Lásd HTML</i>

## I, Í

I-objektum	28
id	20
IDE	13, 373
identifier	<i>Lásd id</i>
idézőjel	
sztringen belül	55
időpont megadása	212
időzítő	253
If ... ElseIf	103
If ... Then	96
If ... Then ... Else	100
igazítás, bekezdése	27
igazságtáblázat	111
image	258
images	248, 267
IMG-objektum	245
Immediate	380
index	137
Infinity	70
inicializálás	99, 306
Initialize	306
inline eseménykezelés	85
inline-objektum	43
innerHTML	40
innerText	39, 57
input	10, 62
INPUT-objektum	37, 258, 326
InputBox	60
kezdőértéke	61
insertAdjacentElement	271
insertCell	165
insertRow	165
InStr	205
InStrB	211

InStrRev	205
Int	73
Integer	186
integrált fejlesztői környezet	373
integrált fejlesztői rendszer	13
IntelliSense	378
interaktív feldolgozás	17
interfész	304
interpreter	13
Is	193
IsDate	213
IsEmpty	190
ismétlési feltétel	127
IsNull	192
IsNumeric	115
IsReady	318
IsRootFolder	318
Item	293
iteráció	120
iterátor	146

## J

JavaScript	47
jelölőnégyzet	258, 259
Join	208
JScript	48
justify	27

## K

karakter	9
karakterentitás	25
karakterkészlet megadása	23
karakterlánc	55
karaktorsorozat	55
kép beillesztése	245
kép eszköztár letiltása	245
képek átméretezésének letiltása	245
képmező	258
kerekítés	75
keresés	
bináris	337
lineáris	335
rendezett tömbben	156, 337
tömbelemé	155
keresés a listában	290
kétágú szelekció	99
Key	294
keyCode	235
keypress	275

kifejezés	66
aritmetikai	66
logikai	114
kifejezések egyszerűsítése	67
kilépési feltétel	127
kiszolgáló	
folyamaton kívüli	276
kiterjesztés	5
kiválogatás	343
két tömbbe	343
közös elemeké	345
kivétel	244
kivételkezelés	244
kód csatolása	81
kódkomponensek	273, 276
kódrészletkönyvtár	365
kollekció	145
almappáké	317
fájloké	317
meghajtóké	317
komplementer szín	79
konkatenáció	59
konstansok	
deklarálása	68
háromállapotú	76
VBScript	69
konstruktor	311
konténer	43
konzol	10
köteget feldolgozás	17
kulcs	293
kulcsszó	53
különbség (halmazoké)	298

## L

language	30, 47
lapdobás	329
LargeChange	275
LCase	202
lebegőpontos forma	63
lebegőpontos mennyiség	194
left	27, 249
Left	203
LeftB	211
legkisebb elem kiválasztása	336
Len	202
LenB	211
length	146, 148, 156, 167, 262
Length	318
lépésenkénti finomítás	171
lépésenkénti végrehajtás	367, 370

lineáris keresés	335
link	<i>Lásd</i> hivatkozás
lista	289
cirkuláris	292
elem keresése	290
elem törlése	292
elemek megjelenítése	290
rendezett feltöltése	291
lista vezérlőelem	260
listaelemek indexelése	260
listafej	290
ln x	73
Locals	380
location	265
Log	73
logikai érték	110
logikai hiba	239
logikai kifejezés	114
logikai műveletek	111
igazságtáblázat	111
precedencia	112
logikai változó	55, 110
lokális változók	88
Long	186
LTrim	202

## M

magas szintű programnyelv	12
mantissza	186
mappa	
aktuális	320
mappaobjektum	318
matematikai függvények	73
Max	275, 379
maximum kiválasztása	336
néhány változó esetén	337
maxLength	42
Me	225
media	330
meghajtó	317
megjegyzések	29
szkriptekben	50
megszámolás	336
menüsor készítése	284
META-objektum	22
method	264
metódus	34, 87
alapértelmezett	306
metszet	298



mező (rekordé)	312
objektumot tartalmazó	316
tömböt tartalmazó	315
Microsoft Campus Szerződés	383
Microsoft Development Network	4
Microsoft Script Debugger	367
Microsoft Script Editor	373
Microsoft Visual Studio	382
Mid	203
MidB	211
Min	275, 379
minimumkiválasztásos rendezés	339
minősített hivatkozás	31
modális párbeszédablak	279
moduláris program	135
mondatszerű leírás	96
MoveFile	320
MoveFolder	319
MsgBox	278
multilista	292
multiple	261
mutató	289

## N

name	22, 258
Name	318, 320
NaN	70
navigate	263
nbspace	25
négyzetgyök függvény	73
nem modális párbeszédablak	279
nem művelet	111
igazságtáblázat	111
New	302
no-repeat	252
Not	111
Nothing	193
Now	213
Null	191
Number	240
numerikus változó	55, 63

## Ny

nyomtatás	328
táblázaté	330
nyomtatási formátum	330

## O, Ó

Object	186
OBJECT-objektum	274
objektumok	15, 299
adatainak elrejtése	299
beillesztése	271
elrejtése	58
inicializálása	306
konstruktora	311
létrehozása	302
megjelenítése	58
megszüntetése	307
metódusai	16, 303
szerepe	299
tagjai	300
tárolása tömbökben	303
tulajdonságai	15
objektumalapú programnyelv	47, 300
objektum-hivatkozás	148
objektum-orientált programnyelv	300
objektumosztály definíciója	300
objektumváltozó	148
offsetX	229
offsetY	229
On Error GoTo 0	244
On Error Resume Next	240, 244, 324
onafterprint	328
onbeforeprint	328
onblur	237
onchange	260
onclick	31, 231
ondblclick	33, 231
onerror	240, 241
onfocus	237
onkeydown	71, 235
onkeypress	235
onkeyup	235
onload	89
onmousedown	231
onmousemove	229
onmouseout	230
onmouseover	230
onmouseup	231
onmousewheel	232
onreset	264
onsubmit	264
OpenAsTextStream	322
OpenTextFile	321, 322
operandus	66
operátor	64
OPTION	260

Option Explicit	55
Or	111
Orientation	379
osztály	15
definiálása	299
osztály-azonosító	273
output	10, 63
Output	380

## Ö, Ó

öröklődés	300
összefésülés	343
összegezés	334
összehasonlítás	
bájtónként	95
szövegszerű	95

## P

P-objektum	26
page-break-after	329
page-break-before	329
PARAM	275
paraméter	34
paraméterátadás	
alapértelmezett mód	217
cím szerint	216
érték szerint	215
parancsfájl	14
parancsgomb	37, 258
engedélyezés	38
mérete	39
tiltás	38
parancsnyelv	14
parancssor	92
párbeszédablak	
modális	279
nem modális	279
párbeszédsegítő	276
parentElement	224
ParentFolder	318, 320
password	42, 258
Path	318, 320
pénznem típusú változók	200
pixel	34
pixelLeft	249
pixelTop	249
play	273
popup ablak	285
PopupMenu	285
position	249

post	264
precedencia	65
prefixumok	187
PRE-objektum	28
Preserve	161
print	328, 330
Private	300
program (számítógépes)	11
programfejlesztés folyamata	355
programnyelv	
generációk	12
magas szintű	12
negyedik generációs	12
objektumalapú	300
objektum-orientált	300
programok	
indítása a parancssorból	92
szerkezete	90
programozási nyelv	11
progressbar	274
prompt	61
Properties ablak	375
Property	304
Property Get	304
Property Let	304
Property Set	305
pszudokód	97
pt	329
Public	300
px	<i>Lásd pixel</i>

## Q

quick sort	349
Quick View	375
Quit	277

## R

radián	73
radio	258
Raise	240, 243
Randomize	78
Read	323
ReadAll	323
ReadLine	323
readOnly	42
rect	284
ReDim	160
Registry Editor	273
regisztrálás	273
rejtett mező	258

rekord	312	screenY	229
létrehozása	313	Script Debugger	367
mezői	312	Script Editor	373
mozgatása	313	Script Outline	375
törlése	314	Script Runtime	293, 317
rekurzió	346	Scripting	293
hátrányai	348	SCRIPT-objektum	30, 47
reláció	93	scroll	275, 280, 379
relációk		search	265
precedenciája	113	segédablak	375
relációs operátorok	93	select	109
relatív hiba	196	SELECT-objektum	260
relatív hibakorlát	196	Select Case	107
Rem	51	selected	260, 261
remove	262	selectedIndex	260
Remove	295	Set	148
RemoveAll	295	setAttribute	193
removeAttribute	260	setInterval	255
removeNode	272	SetLocale	215
rendezés	141	setTimeout	253
adatbevitelnél	163	Sgn	73
beillesztéssel	341	Shell-rendezés	342
buborékos	340	shiftKey	235
gyorsrendezés	349	shiftLeft	235
minimumkiválasztásos	142, 339	show	286
Shell	342	showModalDialog	279
rendezési algoritmusok	339	showModelessDialog	279
rendszerleíró adatbázis	273	Sin	73
Replace	206	Single	186
reset	258	size	41, 260
resizable	280	Size	318, 320
resizeTo	34	Skip	323
returnValue	228, 264, 280, 316	SkipLine	323
RGB	78	SmallChange	275
right	27	sorobjektum	153
Right	203	soros elérésű állomány	321
RightB	211	sortörés-objektum	26
Rnd	78	Source	240
RootFolder	318	Source View	374
Round	75	Space (függvény)	204
rows	156	SPAN-objektum	43, 44
rögzíthető ablak	375	Split	208, 266
RTrim	202	Sqr	73
Run	320	src	81, 245
Running Documents	380	srcElement	224
		statikus tömb	160
		status	38, 280
		Step	122
		step into	367
		Step Into	369
		step out	368
		Step Out	369
Save	278		
scrollbar	275		
screen	330		
screenX	229		

## S

Step Over	368, 369
stílus alkalmazása	45
stíluselem	45
Stop	372
StrComp	95
String (változó altípus)	186
String (függvény)	204
StrReverse	207
struktúra	312
strukturált algoritmus	134
strukturált elrendezés	20
style	39, 45
Sub ... End Sub	83
SubFolders	318
submit	258
süti	267
mappája	270

## Sz

számlálás	99
szekvencia	134
szekvenciális állomány	321
szelekció	93
egyágú	95
egymásba ágyazás	102
kétágú	99
többágú	103
szemantikai hiba	356
szín-függvény	78
szintaktikus hiba	239
szintaxis	11
szkript vázlat	375
szkriptek	13
hibái	52
sorrendje	49
szerkesztőprogramok	52
szókőz, nem törthető	25
szótár	
elemek rendezése	296
keresés a szótárban	295
törlés a szótárból	295
szöveges változó	55
szövegfájl	10, 317, 321
beolvasása	323
írása	322
szövegfolyam	321
szövegmező	258
jelszóbekérő	258
szövegmező-objektum	41

sztringek	55
átalakítás numerikus értékké	77
összefűzése	59
összehasonlítás	94
szubrutin	83

## T

tabIndex	262
táblázat	
összeállításának menete	179
szerkezete	153
táblázat nyomtatása	330
táblázatobjektum	153
TABLE-objektum	153
tag	
nyitó, záró	19
tagfüggvény	300
tagName	146
tagoló karakterek	24
tags	147
Tan	73
tárgykód	13
TD-objektum	153
Terminate	307
Területi beállítások	
decimális kód	215
módosítása	214
rövidítés	215
tervezés (programé)	355
felülről lefelé	356
tervezési nézet	374
tervezésidejű vezérlőelem	382
tesztelés (programé)	356
text	25, 41, 258, 260
TextPad	363
beállításai	363
TextPosition	379
TextStream	317, 321
TickFrequency	379
TickStyle	379
Time	186, 213
Timer	214
TimeSerial	214
típuseltérés hibaüzenet	72
típuskonverzió	77, 187
title	285
TITLE-objektum	22
Toolbox	375
top	249
TotalSize	318
többágú szelekció	103

tömb	137	altípusok jelölése	187
asszociatív	293	bájt típusú	186
átadása az alprogramnak	220	dátum típusú	186
átadása az eljárásnak	221	dupla pontosságú	187
dinamikus	160	egész típusú	185, 186
egydimenziós	137	egyszeres pontosságú	187
statikus	160	egyszerű	185
többdimenziós	150	értékének beolvasása	323
tömböt tartalmazó	288	értékének kiírása	323
töréspont	367, 380	értéktartományok	186
tulajdonságai	381	lebegőpontos	185
törzsobjektum	21, 23	logikai	110
TR-objektum	153	összetett	185
Trim	202	pénznem típusú	186
tristate	76	valós típusú	185, 186
TristateFalse	76	változók	
TristateTrue	76	értékkadás	56
TristateUseDefault	76	hatókör	87
True	38, 55	neve	53
túlsordulás	188	száma	53
type	37, 224	típusa	55
Type	320	value	37, 42, 258
TypeName	189	Variant	185
		VBA	276
		vbAbort	279
		vbAbortRetryIgnore	278
		vbCancel	279
		vbCritical	279
		vbExclamation	279
		vbIgnore	279
		vbInformation	279
		vbNewLine	69
		vbNo	279
		vbObjectError	240
		vbOK	279
		vbOKCancel	278
		vbOKOnly	278
		vbQuestion	279
		vbRetry	279
		vbRetryCancel	279
		VBS	30, 48
		VBScript	30, 47
		szintaxisa	48
		VBScript.TCL	366
		vbYes	279
		vbYesNo	278
		vbYesNoCancel	278
		végtelen ciklus	125
		véletlenszámok előállítása	78
		vezérlő karakter	9
		vezérlőelemek	258
		vezérlőelem-komponensek	273
<b>U, Ú</b>			
U-objektum	28		
UBound	143		
UCase	201		
Unescape	248, 265		
Unicode	10		
unió	297		
tömbelemeké	345		
Until	128		
<b>Ü, Ű</b>			
ügyfél–kiszolgáló modell	273		
üres objektum	20		
üres sztring	71		
űrlap-mezők	258		
űrlapobjektum	263		
ütköző	164, 335		
<b>V</b>			
vagy művelet	111		
igazságtáblázata	112		
választógomb	258		
változó	53		
alapértelmezett érték	189		
altípusok	185		

# TÁRGYMUTATÓ

---

videófájl beillesztése	273
visibility	58, 166, 286
hidden	166
visible	166
visible	58, 166, 167
Visual Basic	47
Visual Basic for Applications	47, 276
Visual Basic Script	47
Visual InterDev	382
Visual Studio	382
Visual Studio .Net 2003	383
visszalépéses keresés	351
stratégia meghatározása	354
VolumeName	318

## W

Watch	380
Web Matrix	383
Weekday	214
WeekDayName	214
wheelDelta	232
While	128
width	154, 246
window	21
metódusok	34

Windows Script Host	320
Windows Script Technologies	320
With ... End With	250
Workbooks	277
Worksheets	277
write	41
Write	322, 372
WriteLine	322, 372
writeln	41
WSH	320
WshShell	320

## X

x (tulajdonság)	229
-----------------	-----

## Y

y (tulajdonság)	229
-----------------	-----

## Z

z-index	250
zoom	246



