



Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Egyszerűen Web

Készítette:

Kromesch Sándor (polgári szolgálatos)

Konzulens:

Dr. Charaf Hassan

2003.

HTML (Hyper Text Markup Language)	7
A HTML dokumentumokról	7
Néhány szóban a HTML alapvető szabályairól	7
A HTML dokumentum elvi felépítése	8
A HTML dokumentum fejléce	8
A HTML dokumentum szövegtestének felépítése	9
A HTML dokumentum címszintjeinek felépítése	10
Bekezdések a HTML dokumentumban	10
Kereszthivatkozások HTML dokumentumok között	11
Karakterformátumok a HTML dokumentumban	12
Képek elhelyezése a HTML dokumentumban	13
Különböző listaformátumok a HTML dokumentumban	15
A HTML formátum táblázatai	17
A HTML dokumentumablak felosztása	20
Kérdőívek a HTML dokumentumban	21
A HTML dokumentum egyéb elemei	23
A JavaScript használata a HTML dokumentumban	23
A HTML speciális karakterei	24
Karaktertáblák	27
Cascading Style Sheets (CSS)	27
Bevezetés	27
A stíluslap csatolása	28
Csportosítás	28
Öröklődés	29
A class szelektor	29
Megjegyzendő, hogy szelektoronként (HTML elemenként) csak egy osztály definiálható!	30
Az ID szelektor	30
Összekapcsolt szelektorok	30
Megjegyzések	30
Látszólagos osztályok és elemek	30
Látszólagos osztályok az élőkapcsokban	31
Tipografikai látszólagos elemek	31
A 'first-line' látszólagos elem	32
A 'first-letter' látszólagos elem	32
Látszólagos elemek a szelektorokban	33
Látszólagos elemek többszörözése	33
Rangsor	33
'important'	34
A rangsor felállítása	34
Formázásmodell	35
Blokkszintű elemek	36
Függőleges formázás	38
Vízszintes formázás	39
Listaelemek	39
Lebegő elemek	40
Soron belüli elemek	41
Helyettesített elemek	41
Sorok magassága	42
A Váson	42
CSS tulajdonságok	43
A tulajdonság-érték párok jelölési rendszere	43
Font tulajdonságok	43
Szín- és háttértulajdonságok	48
Szöveg tulajdonságok	51
Doboz-tulajdonságok	54

Osztályozó tulajdonságok	61
Egységek	64
Hosszúság egységek	64
Százalékos egységek	65
Színjelölések	65
URL	66
Összhang	66
Előre-kompatibilis elemzés	66
JavaScript	69
Mi is az a JavaScript?	69
JavaScript és a böngészők	69
JavaScript beágyazása a HTML dokumentumba	70
Legfontosabb események	71
Használjuk amit tudunk, 2. példa	71
Link a Script-re	71
Csoportosítsunk, avagy a függvények	72
Objektumok, dokumentumok	73
Ha hibáztunk	74
Események	74
Változó képek	75
Előtöltött képek	76
Időzítések	76
Status sor	77
SWITCH használat	78
Ellenőrzések	80
"Üres szöveg"	81
Az XML	82
Bevezetés	82
Az első XML dokumentumunk létrehozása	83
XML alkalmazások	84
Az XML adatfolyam felépítése (XML fájl-ok)	86
Elemek (Elements)	86
Megjegyzés	86
Tulajdonságok (attributes)	86
Egyed hivatkozások (entity references)	87
Feldolgozási utasítások (processing intructions)	87
A CDATA szakaszok	87
Névterek	87
Az XML adatfolyam érvényességének ellenőrzése	89
Elem típus deklaráció (Element Type Declaration)	90
A tulajdonságlista (attributum lista) deklarációja	90
Egyed deklaráció (Entity declaration)	91
A VKONYV XML adatfolyam nyelvtani szabályai	92
Az XML adatfolyam megjelenítése	92
A CSS	93
Az XSL	93
Egy XML adatfolyam elemzése (Parse)	94
Egy DOM elemző működése	95
Egy SAX elemző működése	96
ASP – (Active Server Pages)	97
Egy kis alapozás	97
Mozgásba lendül a kód	98
ASP a láthatáron	98
Az ASP kódok beágyazása	99
Az ASP objektummodell	101

Egy HTTP válasz – a Response objektum	102
A válaszpuffer	102
HTTP fejlécek küldése	102
Tartalom és karakterkészlet	103
HTTP státusz és átirányítás	104
Ha sokáig tart az oldal előállítása	104
Gyorsítótárak – a cache	105
HTTP kérés – a Request objektum	105
Adatfeltöltés a POST HTTP paranccsal	106
A HTTP tartalom kiolvasása	107
Cookie	108
Az IIS naplójánnak írása	109
Felhasználóazonosítás névvel és jelszóval	109
A Request kollekciók	110
Az ASP alkalmazás és munkamenet	111
Az ASP munkamenet	111
A Session objektum	112
A global.asa fájl	113
Az ASP alkalmazás	114
Az alkalmazás védelme	115
Az Application objektum	115
A global.asa fájl mégegyszer	116
Objektumok és típuskönyvtárak	117
A Server objektum	118
Naplózás az eseménynaplóba	119
Speciális karakterek a HTML kódban	120
Árvíztűrő tükörfürögép	120
További META elemek	121
Kódtáblák	121
Az UTF-8 kódolás	122
Lokalizálás	122
Hibakezelés az ASP oldalban	123
VBScript röviden	127
Adatípusok	127
Konstans definiálás	127
VBScript operátorok:	127
VBScript változók:	128
Változó deklarálás:	128
Elágazások	129
Ciklusok:	129
Előtesztelő ciklus amíg igaz:	129
Háttesztelő ciklus amíg igaz:	130
Előtesztelő ciklus amíg igaz nem lesz:	130
Háttesztelő ciklus amíg igaz nem lesz:	130
Kiléps ciklusból EXIT DO segítségével:	130
FOR ciklus:	130
CGI	131
Perl	134
Perl 5 nyelv rövid összefoglalása	134
Indulás	135
Adatstruktúrák	135
Környezet	136
Skalárok	136
Tömbök	137
Szintaxis ... utasítások	137

Egyszerű utasítás	138
Összetett utasítás	138
Operátorok és precedenciájuk	139
címpoperátor	140
ismétlés	140
konkatenáció	140
file tesztelés	140
szöveg összehasonlítás	140
szám összehasonlítás	140
I/O	140
A szövegekhez még járul egypár "idézőjel" operátor	140
Mintaillesztés	141
Használat	141
Módosítók	141
Regular Expressions, reguláris kifejezések	141
Az alapok	141
Karakter sorozatok	142
Speciális karakterek	142
Csoportosítás	143
Operátorok átlapolása	143
definiálatlan	144
TRUE	144
FALSE	144
Beépített függvények	144
néhány függvény	144
Előre definiált változók	147
Alprogramok írása	149
Alprogramok deklarálása:	149
Alprogram hívása:	149
Névtelen alprogram létrehozása:	149
Névtelen alprogramok definiált környezete	149
Beépített függvények átlapolása	150
Modulok	150
package	150
Szimbólumtábla	150
Konstruktor, destruktor	150
Bonyolultabb struktúrák	151
Referenciák létrehozása	151
Referenciák használata	151
Szimbólikus hivatkozások	152
OOP	153
Amit a Perl objektumokról tudni kell:	153
Objektum	153
Osztály	153
Metódus	154
Destruktor	155
PHP	155
A PHP rövid története	155
Alapok	156
Változók PHP-ban	157
Tömbök a PHP-ban	158
Változók hatásköre	161
Konstansok	163
Kommentek	163
Operátorok	163
Operátorok precedenciája	163

Aritmetikai operátorok	164
Hozzárendelő operátorok	164
Bitorientált operátorok	165
Összehasonlító operátorok	165
Hibakezelő operátorok	166
Növelő/csökkentő operátorok	167
Logikai operátorok	167
String operátorok	167
Vezérlési szerkezetek	168
if	168
else	168
elseif	168
Vezérlési szerkezetek alternatív szintaxisa	169
while	169
do..while	170
for	171
foreach	172
break	172
continue	173
switch	173
declare	175
Tick-ek	175
return	176
require()	176
include()	176
require_once()	178
include_once()	178
Függvények	179
Felhasználó által definiált függvények	179
Függvényargumentumok	179
Referencia szerinti argumentumfeltöltés	179
Argumentumok kezdőértékei	180
Változó hosszúságú argumentumlista	180
Visszatérési értékek	180
Függvényváltozók	181
Fejezet. Osztályok, objektumok	181
class	181
extends	183
Konstruktor	184
::	185
parent	186
Objektumok szerializációja, objektumok session-ökben	187
A speciális __sleep és __wakeup metódusok	188
Referenciák a konstruktorban	188
Referenciák	189
Mik a referenciák	189
Mit lehet referenciákkal tenni	189
Mit nem lehet referenciákkal tenni	190
Referenciakénti paraméterátadás	190
Referencia visszatérési-érték	191
Referenciák megszüntetése	191
A PHP által használt referenciák	191
global referenciák	191
\$this	192
Hibakezelés	192
Adatbázis kezelés (MySQL függvények)	194

HTML (Hyper Text Markup Language)

A HTML dokumentumokról

A HTML dokumentum-formátumot tekinthetjük az ún. hyper-text egyik megvalósítási formájának is. A HTML dokumentum egy olyan szövegfájl, amely a szövegen kívül tartalmaz ún. "HTML-tag"-eket - formázóutasításokat -, valamint megjelenítendő objektumokra történő hivatkozásokat is. Ezek a HTML formázóutasítások (más szóval: parancsok, elemek) befolyásolják a dokumentum megjelenítését, kapcsolatait. Ezeket az utasításokat a böngészőprogram értelmezi és végrehajtja. Ezen okból a formázóutasítás mindig megelőzi azt a részét a dokumentumnak, amelyre vonatkozik. A dokumentumkészítéshez használható HTML utasítások köre állandóan bővül, a nyelv fejlődik. A szabványosítás csak lassan követi a fejlődést. Ezért nem minden böngészőprogram tudja a HTML utasítások mindegyikét értelmezni. Egy böngésző, ha számára értelmetlen utasítással találkozik, akkor kihagyja, így nem okoznak problémát az újabb keletű - még szabványosítatlan - utasítások a régebbi kiadású WWW-böngészőknek sem. Sajnos a fentiek miatt ugyanazt a dokumentumot két különböző program nem biztos, hogy azonos formában fogja megjeleníteni. Más oka is van ennek. A WWW-n kalandozónál kicsi a valószínűsége annak, hogy rendelkezésére áll ugyanaz a betűtípus, mint a WWW-oldalt fejlesztőnek. Vagy képek esetén semmi garancia nincs arra, hogy minden böngészőprogram ugyanazon felbontásban és színszámmal tudja megjeleníteni a képet. És így tovább ... A HTML-ben mégis az a nagyszerű, hogy nagymértékben megközelíti a platformfüggetlenséget. Egy HTML dokumentum - ha nem is azonos módon - mindenki számára megtekinthető.

Néhány szóban a HTML alapvető szabályairól

A HTML dokumentum normál szövegfájl. Bármely szövegszerkesztővel létrehozható, ill. módosítható, amely nem használ különleges fájlformátumot, vagy ha létezik TEXT formátumú mentési lehetőség benne. A HTML utasításokat a szövegben < és > jelek közé kell zárni. Egy-egy utasítás - HTML parancs, HTML elem - hatását általában a záró utasításpárja szünteti meg, amely megegyezik a nyitó utasítással, csak a / jel vezeti be (természetesen a < és a > jelek között). Az utasítások nagy része opcionális elemeket is tartalmazhat, melyek csak a nyitóutasításban szerepelhetnek, a záróban nem. Az opciók értékadásánál az idézőjel nem mindig kötelező, csak ajánlott. A HTML utasítás kulcsszavaiban nem különböztetjük meg kisbetűket és nagybetűket.

A HTML dokumentum elvi felépítése

Minden HTML formátumú szövegfájl a **<HTML>** utasítással kezdődik és a **</HTML>** záróutasítással végződik. Ezen elemek közé kell zárni a teljes dokumentumot - formázóutasításokkal és hivatkozásokkal együtt.

A HTML dokumentumot két részre lehet bontani a fejlécre és dokumentumtörzsre. (Egy harmadik rész lehet a keretek definíciója.)

A dokumentumot a fejlécelemek vezetik be, melyek kezdetét a **<HEAD>** utasítás jelzi. A fejlécelemek között szokás a dokumentumcím megadni, mely címet a **<TITLE>** és a **</TITLE>** utasítások közé kell zárni. A fejlécet a **</HEAD>** utasítás zárja. Ezt a részét a dokumentumnak általában az ablak címsorában jelenítik meg a böngészőprogramok.

A dokumentumtörzs - amit voltaképpen a WEB böngésző meg fog jeleníteni - a fájl **<BODY>** és **</BODY>** utasítások közötti része. Ezen elemek között kell elhelyezni mindent: a szöveget, hivatkozásokat, képeket, stb. (A keretek és a JavaScript kódok kivételével!)

Példa (egy tipikus HTML dokumentum felépítése):

```
<HTML>  
<HEAD>  
<TITLE>A dokumentum neve</TITLE>  
Fejléc elemek ...  
</HEAD>  
<BODY>A tulajdonképpeni dokumentumtörzs következik ..  
<H1>Ez itt egy alcím</H1>  
<P>Ez itt egy normál bekezdés </p>  
<!-- HTML comment ez nem jelenik meg a dokumentumban -->  
</BODY>  
</HTML>
```

A HTML dokumentum fejléce

A dokumentumot a fejlécelemek vezetik be, melyek kezdetét a **<HEAD>**, végét a **</HEAD>** utasítás jelzi. A fejlécelemek között legfontosabb a dokumentumcím, mely címet a **<TITLE>** és a **</TITLE>** utasítások közé kell zárni. Ezt a részét a dokumentumnak általában az ablak címsorában jeleníti meg a legtöbb böngészőprogram. A **<BASE HREF="protokoll://gépnév/elérési_út">** utasításban szereplő URL határozza meg a báziscímet, melyből a relatív címeket értelmezni kell. Az intelligens kiszolgálók korábban nem kötelező megadni.

Az **<LINK>** utasításban szereplő opciók jelzik dokumentum kapcsolatait más dokumentumokkal, stíluslappal, címszalaggal, stb.

Az **<META NAME="mező" CONTENT="érték">** utasítás jelezheti a keresőrendszerek számára a dokumentum-adatbázisba kerülő adatokat, pl. a dokumentum alkotóját, a létrehozó programot, rövid tartalmat, stb.

A fejlécelemekre jellemző, hogy a böngészőablakban nem jelennek meg! Ezért a World Wide Weben szereplő dokumentumok fejlécének csak nagyon kis hányada tartalmaz a címen kívül egyéb információkat. (HTML fejléc részletesebb leírását lásd. ASP tárgyalásán belül)

Példa (HTML dokumentum fejléc):

```
<HTML>
  <HEAD>
    <TITLE>HTML leírás - 3. lap</TITLE>
    <META NAME="Author" CONTENT="Almási Pál">
    <LINK REL="stylesheet" HREF="pelda.css"><!--ez egy stylesheet hozzárendelés lásd.
CSS-rész -->
  </HEAD>
  <BODY> ... dokumentumtörzs ...
  </BODY>
</HTML>
```

A HTML dokumentum szövegtestének felépítése

Minden HTML formátumú szövegfájl a **<BODY>** és a **</BODY>** utasításokkal közrezárt részében tartalmazza a megjelenítendő részét. (A dokumentum-kereteket kivéve!)

Ezen elemek között kell elhelyezni mindent: a szöveget, hivatkozásokat, képeket, stb.

A **<BODY BACKGROUND="fájlnev.kit" BGCOLOR="színkód" TEXT="színkód" LINK="színkód" VLINK="színkód" ALINK="színkód">** utasításban a dokumentumtörzsre vonatkozó fenti előírások is szerepelhetnek opcióként.

A **BACKGROUND="elérési_út/fájlnev.kit"** opcióval a dokumentum háttéréül szolgáló fájlt jelölhetjük ki.

Háttérszint a **BGCOLOR="színkód"** opcióval kiegészített utasítással definiálhatunk. (Amennyiben háttérmintául szolgáló fájlt - lásd fent - is megadunk, akkor a háttérszín csak nagyon ritkán fog előtűnni a dokumentumban, pl. a keretek szegélyében.)

A dokumentumban a szöveg színét a **TEXT="színkód"** opcióval jelölhetjük ki.

A **LINK="színkód"** opció a hivatkozások megjelenési színét határozza meg. A **VLINK="színkód"** pedig, a már bejárt hivatkozásokat jelölő színt határozza meg.

Természetesen egyszerre több opció is szerepelhet - tehát nem kötelező egyik sem - a **<BODY>** utasításban.

Példa

```
<BODY
  BACKGROUND="pelda.gif"
  BGCOLOR="#FF3333"
  TEXT="#000099"
  LINK="#993399"
  VLINK="#009999">
  <P> A szemléltetés kedvéért az alábbi szöveg széles szegéllyel határolt! <P>
  <TABLE BORDER=12>
    <TH><TD><H1><A HREF="#pelda">Ez itt egy hivatkozás,</A> kattints rá!
    </H1></TH></TD>
  </TABLE>
  <P> <IMG SRC="k1.gif" ALT="Görgetősáv"> <P>
  <A NAME="pelda"> Ide mutat a hivatkozás!</A>
</BODY>
```

A HTML dokumentum címszintjeinek felépítése

A HTML formátumú szövegfájlban definiálhatunk címeket, ill. alcímeket hat szint mélységig. A legfelső szintű címet a **<H1 ALIGN="hely">** és a **</H1>** utasítaspárral kell közrezárni. A második szintet a **<H2 ALIGN="hely">** és a **</H2>** utasítások határolják, és így tovább a hatodik szintig.

Minden szint más-más betűformátumban jelenik meg a dokumentumban, a böngészőprogram beállításától függően. A címek igazítását szabályozza az **ALIGN** opció, melynek lehetséges értékei: *left*, *center*, *right*. Amennyiben túl hosszú a cím, de egy sorosnak kell maradnia, akkor a **NOWRAP** opció megakadályozza a cím betördelését több sorba.

Tulajdonsága, hogy nem szkrollozódik a dokumentum többi részével ellentétben.

A címek csak a szemlélő számára keltik a tagoltság érzetét, a valóságban nem tagolják fizikailag szakaszokra a dokumentumot. Ezt a tagolást a **<DIV CLASS="osztály" ALIGN="hely">**, **</DIV>** utasításokkal lehet meghatározni, ahol a **CLASS** opció sorolja a megfelelő SGML osztályba a szakaszt, az **ALIGN** pedig a szakasz igazítási formátumát írja elő. Az automatikus tördelést itt is megakadályozza a **NOWRAP** opció, ez esetben a szakasz tördelését a **<P>** vagy a **
** utasításokkal lehet szabályozni.

Példa a címszintekre:

```
<H1 ALIGN="left">Legfelső szintű címsor</H1>
<H2 ALIGN="center">Második szintű alcímsor</H2>
<H3 ALIGN="right">Harmadik szintű alcímsor</H3>
<H4 NOWRAP>Negyedik szintű alcímsor</H4>
<H5>Ötödik szintű alcímsor</H5>
<H6>Hatodik szintű alcímsor</H6>
<DIV ALIGN="center"> Ez egy szakasz, melyben a szöveg elvileg azonos módon - középre igazítva -
kezelendő. </DIV>
```

Bekezdések a HTML dokumentumban

Minden dokumentum, így a HTML formátumú dokumentum is, alapvetően bekezdésekre tagolódik.

A HTML fájlban a bekezdések *kezdését* a **<P>** utasítás jelzi a böngészőprogram számára a végét **</P>**. A legtöbb böngészőprogram két bekezdés között egy üres sort szűr be megjelenítéskor!

A bekezdés elem hordozhat magában a bekezdés stílusát meghatározó opciókat. A bekezdés igazítását a **<P ALIGN="hely">** formájú utasítással szabályozhatjuk. Az automatikus tördelést a **NOWRAP** opció tiltja meg a böngészőprogram számára. Amennyiben tördelhetetlen szóközt igényel a szöveg, akkor az egyszerű szóköz helyett alkalmazzuk a ** ** különleges karaktert.

Amennyiben egy bekezdésen belül mindenképpen új sort szeretnénk kezdeni, akkor a **
** utasítást kell használni. (Nincs zárópárja!)

Példa bekezdésekre:

<P>Ez alapértelmezett (balra igazított) bekezdés.</p>

<P ALIGN=left>Ez balra igazított bekezdés.</p>

<P ALIGN=center>Ez középre igazított bekezdés.</p>

<P ALIGN=right>Ez jobbra igazított bekezdés.</p>

<P ALIGN=justify>Ez sorkizárt bekezdés lenne.</p>

<P>

Itt egy sortörés elem található,

**
**

**melynek hatására új sorban folytatódik a szöveg,
és nem maradt ki üres sor.</p>**

Kereszthivatkozások HTML dokumentumok között

A HTML formátum lényegét az egymásra és egymás tartalmára való hivatkozások jelentik (vagyis a hypertext lehetőség). A dokumentum bármely részéhez hivatkozást (linket) helyezhetünk el, amelyet aktivizálva, a hivatkozottal összefüggésben lévő szöveghez jutunk el. A hivatkozó utasítások megjelenési formája sokféle lehet, a célobjektumtól függően:

A legegyszerűbb esetben a hivatkozás az adott fájl egy távolabbi részére mozditja a böngészőablakot. A hivatkozás kezdetét a **** utasításnak a dokumentumban való elhelyezése jelzi. A hivatkozást a **** utasítás zárja le. Ez az elempár közrezárhat szövegrészt, képet, stb. A közrezárt részt a böngészőprogram a dokumentum többi részétől eltérően jeleníti meg (pl. aláhúzással, kerettel, ...), az egérkurzorral fölé érve a mutató alakja megváltozik. Azt a részt (praktikusan: könyvjelzőt), ahová a hivatkozás mutat a **** és a **** utasítások kell, hogy határolják.

A legtöbb esetben a egy hivatkozás egy másik fájlra/dokumentumra mutat. A hivatkozás kezdetét ekkor a **** utasítás jelzi, a hivatkozást ekkor is a **** utasításelem zárja le. Mind a protokoll, mind az elérési út elhagyható, amennyiben azonos URL-en van a kiindulási dokumentum és a hivatkozott. A hivatkozott fájlnek e példában nincs külön névvel (könyvjelzővel) jelölt része. Működés szempontjából a fentebb leírtak vonatkoznak erre a hivatkozási formára is.

A legbonyolultabb esetben a hivatkozás egy másik fájl valamely pontosan meghatározott részére mutat. A hivatkozás kezdetét a **** utasítás jelzi, és a hivatkozást szintén a **** elem zárja le. Ebben az esetben a hivatkozott fájl kell, hogy tartalmazzon egy olyan részt (könyvjelzőt), ahová a hivatkozás mutat. Ezt a részt a **** és a **** utasítások határolják.

Megjegyzés: Ha az ****, **** utasításpár képet fog közre, akkor a kép szegéllyel jelenik meg, amely szegély letiltható az **** utasításban elhelyezett **BORDER=0** opció alkalmazásával. A képekkel kapcsolatos egyéb hivatkozási lehetőségeket lásd a képeknél.

Példa hivatkozásokra:

Ennek a fájlnek a végére visz ez a hivatkozás.

A makói Almási Utcai Általános Iskola honlapjára vonatkozik e

hivatkozás.

helyére

repít e hivatkozás. Ezen leírással kapcsolatos véleményed

írd meg!

Itt az oldal vége!

Karakterformátumok a HTML dokumentumban

A HTML formátumú szövegfájlban is használhatjuk a szövegszerkesztőkben megszokott karakterformátumokat. Az alábbi táblázat a formázás kezdő és záróutasítása között a mintát is tartalmazza.

Kezdő elem	Ilyen betűformátumot eredményez	Záró elem
	Félkövér betűformátumot eredményez	
<I>	<i>Dőltbetűs formátumot eredményez</i>	</I>
<U>	<u>Aláhúzott formátumot eredményez</u>	</U>
<S>	<u>Áthúzott formátumot eredményez</u>	</S>
<TT>	Fixpontos betűket eredményez	</TT>
	<i>Kiemeli a szöveget</i>	
<CITE>	<i>Idézetekre használható</i>	</CITE>
<VAR>	<i>Változónevet jelöl</i>	</VAR>
	Ez is egy kiemelési lehetőség	
<CODE>	Kódoknál használjuk	</CODE>
<SAMPLE>	Minták jelzésére használjuk	</SAMPLE>
<KBD>	Billentyűfelirat jelzése	</KBD>
<BQ>	Idézet megjelenítése	</BQ>
<BIG>	Nagyméretű betűformátumot eredményez	</BIG>
<SMALL>	Kisméretű betűformátumot eredményez	</SMALL>
_{	Alsóindexet eredményez	}
^{	Felsőindexet eredményez	}
	A részleteket lásd lentebb	

A , utasításpárral direkt módon előírhatók a megjelenő szöveg betűinek a jellemzői. A FACE opciót nem szokás használni, mert nem valószínű, hogy minden rendszerben rendelkezésre áll pl. az ARIAL CE FÉLKÖVÉR betűtípus. A COLOR opció pontosan meghatározza a megjelenítendő szöveg színét. A SIZE opcióban egy számot megadva a betűméretet határozza meg direkt módon. (A SIZE opcióban előjeles szám is szerepelhet, ami az alapbetűtípushoz viszonyított méretet jelöl.)

Példa karakterformátumokra:

Vastag
<I>Dőlt</I>
<U>Aláhúzott</U>
<S>Áthúzott</S>
<TT>Fixpontos</TT>
Kiemelt
<CITE>Idézet</CITE>
<VAR>Változónév</VAR>
Kiemelt
<CODE>Kód</CODE>
<SAMPLE>Minta</SAMPLE>
<KBD>Billentyűfelirat</KBD>
<BQ>Idézet</BQ>
<BIG>Nagyméretű</BIG>
<SMALL>Kisméretű</SMALL>
_{Alsóindex}
^{Felsőindex}
Kicsi piros
N
ö
v
e
k
v
ö
ARIAL CE
SYMBOL

Képek elhelyezése a HTML dokumentumban

A HTML formátumú dokumentumban képeket - grafikákat - is elhelyezhetünk. Az utasítás a szöveg aktuális pozíciójába helyezi a megadott képet. Ennél azért a legegyszerűbb szövegszerkesztő program is többet nyújt. A HTML dokumentum csinosítására is vannak a képek elhelyezésének finomabb lehetőségei is. Ha ezeket mind kihasználjuk, akkor az utasítás a következőképpen fog kinézni: .

Az ALIGN opció meghatározza a kép igazításának módját, lehetséges értékei: *top*, *middle*, *bottom*, *left*, *right*.

A HSPACE a kép melletti vízszintes térközt, a VSPACE pedig a függőleges térközt (ha úgy tetszik: margókat) határozza meg.

A WIDTH a szélességét, a HEIGHT pedig a magasságát adja a képnek, az UNITS által meghatározott egységben (*pixel* vagy *en*).

Az ALT azt a szöveget adja meg, amelyet nem grafikus böngészők használata esetén meg fog jelenni a kép helyett. A grafikus böngészőkben meg ha kurzort a kép fölé mozgatjuk akkor jelenik meg a kurzor alatt.

Az USEMAP, ISMAP összetartozó opciók a kép különböző területeihez különböző hipertext hivatkozásokat rendelhetnek. (Tehát csak akkor van értelmük, ha a kép egy hivatkozás része! Ekkor a hivatkozások és utasításpárját nem kell megadni.) Ezenkívül

szorosan kapcsolódik ezen opciókhoz (az utasítást megelőzően) a következő utasításstruktúra:

<MAP NAME="jelző">

<AREA SHAPE="alak" COORDS="koordináták" HREF="hivatkozás">

...

</MAP>

amellyel egy hivatkozási térképet kell megadni. Az <AREA> utasításból természetesen több is szerepelhet. A **SHAPE** opció a *circle*, *rect*, *polygon* értékeket veheti fel, amikor *circle* (kör) esetén a **COORDS** három vesszővel elválasztott koordinátát tartalmaz (középx,középy,sugár), *rect* (téglalap) esetben négyet (balfelsőx,balfelsőy,jobbalsóx,jobbalsóy), a *polygon* (sokszög) esetén pedig minden csúcs koordinátáit meg kell adni. A <MAP NAME="jelző">, </MAP> utasításpárral körülhatárolt hivatkozási rész külön fájlban is elhelyezhető. Ekkor az **USEMAP** opció kimarad. - Vigyázat az **ISMAP** nem marad el! - Helyette a és az utasítások közé kell zárni az utasítást. (Ahol a *fájlnév.map* annak a fájlnak a neve, URL-je, amely a hivatkozásokat tartalmazza.)

Példa kép alkalmazására:

<MAP NAME="osztott">

<AREA SHAPE="circle" COORDS="130,130,50" HREF="#vege">

</MAP>

<P>

kép szövegbe beszúrva

így jelenik meg

</P>

<P>

Ugyanez jobbra igazítva és arányosan kicsinyítve...

</P>

<P>

Ugyanez balra igazítva és arányosan kicsinyítve...

</P>

<P>

Ugyanez a szöveg felső széléhez igazítva...

</P>

<P>

Ugyanez sor közére igazítva, valamint...

</P>

<P>

Ugyanez a sor alsószéléhez igazítva, valamint...

<P>

VÉGE

Különböző listafomátumok a HTML dokumentumban

A HTML formátumú szövegfájlban használhatunk listákat is, amelyek szövegszerkesztőkbeli megfelelői a felsorolások és bajuszos bekezdések.

A számozott bekezdések (felsorolások) megfelelője a számozott lista, az ún. "bajuszos" bekezdések megfelelője pedig a számozatlan lista.

A harmadik lista típus pedig a leíró lista, ahol az egyes lista elemekhez tartozhat egy hosszabb leírás is.

A számozott listát az **** és az **** utasítások közé kell zárni. A számozatlan listát pedig az ****, **** utasításpár közé. Mindkét típusú listában használhatjuk a listafejléct, melyet az **<LH>** utasítás vezet be - az **</LH>** pedig zár! Mindkét listatípusban a listák sorai az **** utasítással kezdődnek és nem kötelező lezárni.

Számozott lista esetében a kezdő sorszám közvetlenül megadható az **<OL SEQNUM="szám">** formájú kezdőutasítással. Másik lehetőség, hogy egy előzőleg definiált lista számozása folytatható az **<OL CONTINUE>** kezdőutasítás használatával. Egyébként az **** utasítás 1-től kezd a lista tagok számozását.

A számozatlan listák kezdőeleme is hordozhat formázóinformációkat. Az **<UL SRC="fájlnev.kit">** formájú utasítás például a listasort megelőző bajuszként a megadott képfájlt használja. Az **<UL DINGBAT="karakter">** a megadott bajuszkaraktert alkalmazza. Az **<UL WRAP="irány">** pedig többszlopos listák esetén az igazítás formáját határozza meg. (A **WRAP** opció a *horiz* és a *vert* értékeket veheti fel.

A számozatlan listák speciális - külön HTML utasításokkal létrehozható - fajtái a könyvtárlista és a menülista. A könyvtárlista típus a **<DIR>** utasítással kezdődik és a **</DIR>** utasításra végződik. A menülista pedig **<MENU>** és a **</MENU>** utasításokkal határolt. Ezek a listaformák a normál számozatlan listáktól mindössze annyiban különböznek, hogy a könyvtárlista tagjai 20 karakteresnél, a menülista tagjai pedig egy sorosnál nem lehetnek hosszabbak és nincs "bajuszuk".

A leíró listát a **<DL>** és a **</DL>** utasítások közé kell zárni. A lista fejléc megadása azonos az többi listatípusnál látottal. A listák egyes alkotóelemeinek kezdetét a **<DT>** utasítás jelzi, az ehhez tartozó leírás kezdetét pedig a **<DD>** utasítás határozza meg. Nincs egyik utasításnak sem záró párja, ezért a lista tag a **<DT>** elemtől a **<DD>**-ig, a hozzá tartozó leírás pedig a **<DD>** elemtől a következő **<DT>**-ig tart.

Példa listákra:

<P>Normál szöveg</P>

<P>

<LH>A számozott lista fejléce</LH>

Első sor

<LH>A beágyazott lista fejléce</LH>

Első elem

Második elem

Harmadik elem

Második sor

Harmadik sor

Negyedik sor

</P>

<P>Normál szöveg</P>

<P>

<LH>A számozatlan lista fejléce</LH>

Első sor

Második sor

<UL wrap="horiz"><LH>A beágyazott lista fejléce</LH>

Első sor

Második sor

Harmadik sor

</P>

**<P>Könyvtárlista:
**

<DIR>

Első tag

Második tag

Harmadik tag

Negyedik tag

</DIR>

</P>

**<P>Menülista:
**

<MENU>

Első menü

Második menü

</MENU>

</P>

<P>Normál szöveg</P>

<P>

<DL><LH>A leíró lista fejléce</LH>

<DT>Első sor

<DD>Az első sorhoz tartozó leírás, lehet hosszabb szöveg

is.

A leírás tördelése automatikus. Szépen igazodnak a betördelt sorok az első sor kezdőpontjához.

<DT>Második sor

<DD>A második sorhoz tartozó leírás

</DL>

</P>

A HTML formátum táblázatai

A HTML formátumnak ez az utasításcsoportja képes a legváltozatosabb szöveg-, és képmegjelenítési formák előállítására. A **<TABLE>** és a **</TABLE>** utasítások közé zárt részt tekintjük egy táblázatnak.

A táblázatnak a címét a **<CAPTION>** és a **</CAPTION>** utasítások között kell megadni. (Figyelem! Az így megadott cím nem a táblázatban, hanem előtte fog megjelenni!) A cím **<CAPTION ALIGN="hely">** formájú megadással igazítható.

A táblázat minden sora a **<TR>** utasítással kezdődik és a következő **</TR>** vagy **<TR>**-ig, ill. a táblázat végéig tart. Egy sor tartalmazhat oszlopfejléceket és adatokat. Az oszlopfejléceket a **<TH> </TH>** utasítás vezeti be és választja el egymástól. A táblázat adatcellái pedig a **<TR>**-rel megkezdett sorban egy **<TD>** utasítással kezdődnek és minden cella a következő **<TD>**-ig - ill. **</TD>** ill. a következő sor elejét jelző elemig - tart, ahol értelemszerűen új cella kezdődik. Az oszlopfejléceknek és az adatcelláknak csak a kezdőutasítása használatos - habár van lezáró utasításuk is (**</TH>**, **</TD>**) -, mert a záróutasításuk elhagyható!

A táblázat nyitóutasítása tartalmazhat a teljes táblázatra vonatkozó beállításokat: **<TABLE BORDER="szám" ALIGN="hely" COLSPEC="oszlopjellemzők" UNITS="egység" NOWRAP CELLPADDING="pszám" CELLSPACING="kszám" BGCOLOR="színkód">**

Ahol a **BORDER** opció a rácszat szélességét határozza meg. (0 esetén nincs rácszat.) Az **ALIGN** a teljes tábla elhelyezkedését határozza meg (*left, right, center* lehet). A **COLSPEC** egy oszlop igazítását és szélességét adja meg. Egy oszlopra vonatkozóan egy betű és szám egybeírva (pl.: *L12 C24 R10*), melytől a következő oszlop értékeit egy köz választja el. Az **UNITS** a számokhoz tartozó mértékegységet jelöli ki (*en, relative* - oszlopszélességhez -, *pixel*). A **NOWRAP** opció a cellák szövegének tördelését tiltja le. Végül a **BGCOLOR** a táblázat háttérszínét határozza meg.

A táblázat oszlopfejlécei nem csak a legfelső oszlopban szerepelhetnek, hanem a táblázatban bárhol (pl. sorok címeként is).

Mind az oszlopfejlécekben, mind az adatcellákban használhatók a következő formázásra való opciók:

COLSPAN="szám":

Egyesít több egymással szomszédos cellát - vízszintesen.

ROWSPAN="szám":

Egyesít több egymással alatti cellát - függőlegesen.

ALIGN="hely":

Igazítja a cellák tartalmát - vízszintesen. Lehetséges értékei: *left, center, right, justify, decimal*

VALIGN="hely":

Igazítja a cellák tartalmát - függőlegesen. Lehetséges értékei: *top, middle, bottom, baseline*

Példa táblázatok használatára:

```
<TABLE border="5" align="center">
  <CAPTION> A táblázat címe
</CAPTION>
  <TR>
    <TH colspan="2">      Az 1.-2. oszlop közös fejléce
    </TH>
    <TH colspan="3">      A 3.-4.-5. oszlop közös fejléce
    </TH>
    <TH rowspan="2">      A 6. (2 soros) oszlop fejléce
    </TH>
  </TR>
  <TR>
    <TH> Az 1. oszlop másodrendű fejléce
    </TH>
    <TH colspan="2">      A 2.-3. oszlop másodrendű fejléce
    </TH>
    <TH colspan="2">      A 4.-5. oszlop másodrendű fejléce
    </TH>
  </TR>
  <TR>
    <TH> Az első adatsor címe
    </TH>
    <TD> Az első adatcella
      <TD> Adat (indexe: C3)
      </TD>
      <TD> Adat (indexe: D3)
      </TD>
      <TD> Adat (indexe: E3)
      </TD>
      <TD> Adat (indexe: F3)
      </TD>
    </TD>
  </TR>
  <TR>
    <TH> A 2. adatsor címe
    </TH>
    <TD> Adat (indexe: B4)
    </TD>
    <TD> Adat (indexe: C4)
    </TD>
    <TD> Adat (indexe: D4)
    </TD>
    <TD> Adat (indexe: E4)
    </TD>
    <TD> Adat (indexe: F4)
    </TD>
  </TR>
  <TR>
    <TH> Függőleges igazítások
    </TH>
    <TD align="center" valign="bottom">      Le
      <TD align="center" valign="top">      Fel
    </TD>
  </TR>
```

```

        <TD align="center" valign="middle">      Középre
        </TD>
        <TD>
        </TD>
        <TD> Adat (indexe: F5)<BR>Ettől a cellától balra egy
üres cella,
        alatta pedig két üres cella összevonva
        </TD>
    </TR>
    <TR>
        <TH> Vízszintes igazítások:
        </TH>
        <TD> Alapértelmezés
        </TD>
        <TD align="left"> Balra
        </TD>
        <TD align="center">      Középre
        </TD>
        <TD align="right">      Jobbra
        </TD>
        <TD rowspan="2">
        </TD>
    </TR>
    <TR>
        <TD align="center" colspan="5"><IMG src="k12.gif" alt="Kép
a cellában">
        </TD>
    </TR>
</TABLE>

```

A HTML dokumentumablak felosztása

Egyetlen böngészőablakban több HTML dokumentum is megjeleníthető a **<FRAMESET>** és a **</FRAMESET>** utasításpár, valamint a szorosan kapcsolódó **<FRAME>** utasítás együttes használatával.

A **<FRAMESET ROWS="oszlophatárok">** kezdőutasítással osztható fel a képernyő függőlegesen, a **<FRAMESET COLS="sorhatárok">** utasítással pedig vízszintesen. Ahol az oszlop- és sorhatárok megadhatók képernyőpontban ill. százalékosan - vesszővel elválasztva -, a maradék képernyőterületre pedig a * dzsókerkarakter használatával lehet hivatkozni. Mivel vagy csak vízszintesen, vagy csak függőlegesen osztható fel a képernyő, ezért ha mindkét irányban osztott böngészőablak létrehozásához a **<FRAMESET>** elemeket egymásba kell ágyazni! Tehát például egy függőleges felosztáson belül kell vízszintesen elválasztott részekre tagolni egy oszlopot.

A fenti módon definiált területekre a **<FRAME SRC="objektum">** utasítás tölti be a megadott objektumot, mely objektum lehet egy teljes HTML fájl, annak egy meghatározott része, ill. egy kép. Az így kitöltendő keretek viselkedését szabályozza az utasítás **<FRAME NAME="név" SRC="objektum" SCROLLING="érték" MARGINWIDTH="szám" MARGINHEIGHT="szám">** alakú megadása.

Az adott keretnek nevet ad a **NAME** opció, a szkrollozást letilthatja **SCROLLING="no"** kiegészítés (ezenkívül a *yes* és az *auto* értékeket veheti fel a **SCROLLING** opció), a **MARGINWIDTH** és a **MARGINHEIGHT** pedig a kereten belüli margók szélességét szabályozza.

Például a fejlécben megadott **<BASE TARGET="név">** utasítás a **NAME="név"** opcióval elnevezett keretbe irányítja a hivatkozásokat. Egyébként az **** utasítás is ismeri a **TARGET="név"** opciót. (A **TARGET="_top"** opcióval az egész böngészőablakot elfoglalja a hivatkozott dokumentum, tehát feloldja az ablak keretekre osztását!) Ha ezek egyike sem szerepel, akkor a hivatkozás a hivatkozó objektum keretében jelenik meg, felülírva azt!

A **<FRAMESET>**, **<FRAMESET>** utasításpárral határolt területnek meg kell előznie a **<BODY>** utasítással kijelölt dokumentumtörzset! Sőt a egy **<NOFRAMES>** utasítással kell jelezni a dokumentum azon részének kezdetét, amelyet akkor kell a böngészőnek megjelenítenie csak, ha nem ismeri a keretutasításokat. És csak ez a **<NOFRAMES>**-mel kezdődő rész tartalmazhatja a **<BODY>** és a **</BODY>** utasításpárt.

Példa keretek használatára:

```
<FRAMESET ROWS=185,*>
  <FRAMESET COLS=185,*>
    <FRAME SRC=k08.gif SCROLLING=NO NAME="cimer">
    <FRAME SRC=02.htm NAME="felepites">
  </FRAMESET>
  <FRAMESET COLS=25%,*>
    <FRAME SRC=index.htm NAME="tart">
    <FRAME SRC=13.htm NAME="keret" MARGINHEIGHT=10 MARGINWIDTH=10>
  </FRAMESET>
</FRAMESET>

<NOFRAMES>
<CENTER>
<BODY BGCOLOR="#FFFF00">
<FONT COLOR="#FF3333">
<H1>Sajnos ez a böngésző nem támogatja a keretek használatát!</H1>
</FONT>
</BODY>
```

Kérdőívek a HTML dokumentumban

A HTML formátumú dokumentumban kérdőíveket is közzétehetünk, melyek feldolgozásához külön programot kell írni. (Nem HTML-alapút! Általában valami server oldali program meghívása történik pl. ASP, CGI, PHP ...)

A **<FORM METHOD="mód" ACTION="elérési_út/fájlnév.kit">** és a **</FORM>** utasítások zárják közre a kitöltendő kérdőívet/űrlapot.

Az opciókat ajánlott mindig megadni, már csak azért is, mert az **ACTION** határozza meg a feldolgozást végző programot, a **METHOD** pedig a kitöltött űrlap továbbítási módját a feldolgozó programnak. Lehetséges értékei: *GET* - az URL-ben, *POST* - adatsomagban. Az alapértelmezés a *GET*, ami bizonyos veszélyeket rejt magában, mivel túl hosszúra nyúlhat az URL. A *POST* a biztonságosabb mód.

Az **<INPUT NAME="név" TYPE="típus" ALIGN="hely">** utasítással határozható meg egy kitöltendő űrlapmező

A **NAME** természetesen a mezőnév, amely alapján a feldolgozóprogram azonosítja a bevitt adatot. A **TYPE** pedig az adattípus, melyet vár a beviteli mező. Lehetséges típusok: *TEXT* - szöveg, *PASSWORD* - jelszó (nem jelenik meg bevitelkor!), *HIDDEN* - rejtett (ez sem jelenik meg), *CHECKBOX* - kapcsoló (több is kiválasztható egyszerre), *RADIO* - kapcsoló (egyszerre csak egyet lehet kiválasztani), *RANGE* - numerikus adat, *FILE* - csatolandó fájl, *SUBMIT* - adattovábbító gomb, *RESET* - inicializáló gomb, *BUTTON* - egyéb nyomógomb. Az **<INPUT >** utasításban további opciók is szerepelhetnek, a fő opciók értékeitől függően:

- A **VALUE** kiegészítő opcióval megadott értéket veszi fel alapértelmezésként a szöveges vagy numerikus beviteli mező.
- *TEXT* típusú mező esetén egy további opció, a **SIZE="méret"** opció határozza meg a beviteli ablak szélességét, a **MAXLENGTH="érték"** pedig a bevihető maximális szöveghosszt.
- A *CHECKBOX* és a *RADIO* típusú mezők további paramétere lehet a **CHECKED** opció, mely bekapcsolja a kapcsolót - alapértelmezésként.
- *RANGE* típusú mező esetén megadható az a tartomány, melybe a bevitt értéknek bele kell esnie, a **MAX="maximum"** és a **MIN="minimum"** további opciókkal.

- A *FILE* típusú mezőben megadott fájl az **ACCEPT** kiegészítő opcióval megadott MIME módon csatolódik az elküldendő kérdőívhez. (Megjegyzés: Egy *Browse* nyomógommbal támogatott fájlkereső-ablakból lehet a fájlt kiválasztani.)
- A *SUBMIT* és a *RESET* gombokhoz tartozó kiegészítő opció a **VALUE="felirat"**, amely a gombok feliratát jelöli ki. Egyébként a *SUBMIT* gomb lenyomásának hatására küldi el az űrlapadatokat a kérdőív a feldolgozó programnak, a *RESET* gomb lenyomása pedig az alapértékekkel tölti fel a beviteli mezőket.
- A *IMAGE* típussal készíthetünk egy képből gombot. A hozzá tartozó kiegészítő opció **SRC="elérési út/kép név"** segítségével adhatjuk meg a kép elérését.

Hosszabb szöveg bevitelére alkalmas a **<TEXTAREA NAME="név" ROWS="magasság" COLS="szélesség" VALUE="szöveg">**, **</TEXTAREA>** utasításpár, amely egy beviteli ablakot nyit a **COLS**-ban megadott szélességben és a **ROWS**-ban megadott sorban. A **VALUE** az alapértelmezésként megjelenítendő szöveget adja meg.

Egy kérdésre adandó válasz egyszerű - menüből történő - kiválasztását teszi lehetővé a kérdőíven a **<SELECT NAME="név" SIZE="sor">**, **</SELECT>** utasításokkal létrehozott kiválasztásos menü, melynek menüpontjait az **<OPTION>** utasítással adhatjuk meg.

A **SIZE** opció azt határozza meg, hogy hány sorban jelenjenek meg a választható menüpontok. Megadásával szkrollozható menüt kapunk. Elhagyása esetén, ún. legördülő menüből lehet választani. A **MULTIPLE** opció esetén több menüpont is kijelölhető egyszerre. Az **<OPTION SELECTED>** formájú utasítás adja meg az alapértelmezett választást!

Példa űrlap kezelésre:

```
<CENTER><H1>Adatfelvételi lap:</H1>
  <FORM method="post" action="program.bin">
    <INPUT TYPE="reset" VALUE="Alapértelmezés">
<P>Vezetékeve:
  <INPUT NAME="vezeteknev" TYPE="text" VALUE="Kovács"
SIZE="25" MAXLENGTH="30">Keresztneve:
  <INPUT NAME="keresztnev" TYPE="text" MAXLENGTH="50">Férfi:

  <INPUT NAME="neme" TYPE="radio" CHECKED>Nő:
  <INPUT NAME="neme" TYPE="radio">Kora:
  <INPUT NAME="kor" TYPE="range" SIZE="2" MIN="10"
MAX="60"></P>
<P>Érdeklődési köre:
  Windows:
    <INPUT NAME="erdek" TYPE="checkbox" CHECKED>Win95:
    <INPUT NAME="erdek" TYPE="checkbox">LINUX:
    <INPUT NAME="erdek" TYPE="checkbox">OS/2:
    <INPUT NAME="erdek" TYPE="checkbox"></P>
<P>
  <TEXTAREA name="egyeb" cols="40"
rows="4">Közlendők:</TEXTAREA></P>
<P>Foglalkozása:
  <SELECT name="foglakozas">
    <OPTION>diák
    <OPTION>tanár
    <OPTION selected>nyugdíjas
    <OPTION>egyik sem
  </SELECT></P>
```

```
<P>Csatolandó fájl(ok):<BR>
      <INPUT NAME="fajl" TYPE="file"></P>
<P>
      <INPUT TYPE="submit" VALUE="Elküldés"></CENTER>
</FORM></P>
```

A HTML dokumentum egyéb elemei

Egy HTML formátumú szövegfájl a tartalmazhat megjegyzéseket.

A megjegyzés egyik típusa a megjelenítendő megjegyzés, a **<NOTE>** és a **</NOTE>** utasításokkal közrezárva.

A HTML dokumentumban elhelyezhetők olyan megjegyzések is, melyek sehol sem jelennek meg a dokumentum WEB-böngészővel történő megjelenítésekor. Viszont a fájl átszerkesztéskor segítségül lehetnek a módosítást végzőnek. A megjegyzéseket a **<!--** és a **-->** utasítások között kell elhelyezni.

Egy HTML formátumú szövegfájl a tartalmazhat lábjegyzeteket. Az **<FN ID="azonosító">** és a **</FN>** utasítások között szerepel a lábjegyzet szövege. Az így definiált lábjegyzetszövegre hivatkozik a szövegnek az **** és az **** utasításokkal jelölt része.

Amennyiben a megjelenítendő szöveg formátuma pontosan olyan kell, hogy legyen, mint ahogy a HTML fájlban szerepel, akkor azt az előreformázott szöveget jelző utasítások közé kell zárni. Ezen utasítások a **<PRE>** és a **</PRE>**. A közéjük zárt szöveg pontosan annyi szóközzel, pontosan annyi sorban és olyan állapotban fog a dokumentumban megjelenni, mint ahogy azt a HTML fájl tartalmazza. E dokumentumsorozat példái is így módon kerültek rögzítésre ...

A **<PRE WIDTH="szám">**, **</PRE>** utasításpár használatával egy tördeletlen szöveg az adott szélességben betördelhető.

A szövegrészeket tagolás vagy esztétikai ok miatt vízszintes vonallal el lehet választani egymástól. Legegyszerűbb esetben a **<HR>** utasítás egy vízszintes elválasztó vonalat helyez el az adott ponton, a rendelkezésre álló szélességben. Ezt a durva megjelenítést lehet azért finomítani a **<HR ALIGN="hely" WIDTH="hszám" SIZE="vszám" NOSHADE>** alakú utasítással.

Az **ALIGN** az igazítás helyét adja meg (*left, right, middle*). A **WIDTH** a vonalhosszt definiálja, a **SIZE** pedig a vonal szélességét. Mindkettőt meg lehet adni képpontban, ill. a hosszt az ablak-szélesség százalékában. A **NOSHADE** pedig térhatást (árnyékoltságot) tiltja le.

Példa az egyéb elemekre:

```
<!-- Ez itt egy értelmezést segítő megjegyzés. -->
<NOTE>Ez itt egy jegyzet</NOTE>
Ehhez a sorhoz <A HREF="#az">lábjegyzet</A> tartozik.
<HR WIDTH=50 SIZE=50>
<PRE> Ez a sor sok közt tartalmaz. Ez pedig egy új sor, pedig nem előzte meg sem
<P>, sem <BR>
</PRE>
<HR ALIGN="left" WIDTH=50%> <P> <HR NOSHADE></P><P> <FN ID="az">Íme a fenti jelzéshez
tartozó lábjegyzet</FN></P>
```

A JavaScript használata a HTML dokumentumban

Egy HTML formátumú szövegfájl a tartalmazhat JavaScript "programnyelven" megírt kódsorokat is **<SCRIPT LANGUAGE="JavaScript">** és a **</SCRIPT>** utasításokkal közrezárva. A JavaScript, mint programnyelv bemutatására nem kerül most sor. Ezért csak röviden:

A JavaScript változókat és függvényeket a dokumentum fejlécében szokás definiálni. (Vagyis a **<HEAD>** és a **</HEAD>** utasításokkal közrezárt részében a dokumentumnak. Az így definiált függvényeket lehet meghívni, a változókra lehet hivatkozni a szöveg HTML elemeiben.

Figyelem! Nem tévesztendő össze a JavaScript a JAVA programozási nyelvvel. A JAVA nyelven önálló programokat lehet írni, melyeket az **<APPLET CODE="osztály">** utasítással lehet meghívni.

JavaScriptről lásd. Külön fejezet.

Példa JavaScript alkalmazásra:

```
<HTML>
  <HEAD>
    <TITLE>HTML leírás - 17. lap</TITLE>
    <SCRIPT LANGUAGE="JavaScript">
      function Ablak()
      {
        msg=open("", "DisplayWindow", "toolbar=no,directories=no,menubar=no,"
+ "resizable=no,width=300,height=200")
        msg.document.write("<BODY BGCOLOR=#009999>");
        msg.document.write("<CENTER><H2>Új böngészőablak</H2>");
        msg.document.write("<FORM>");
        msg.document.write("<INPUT TYPE='button' VALUE=' BEZÁR ÉS VISSZA
'+onclick='window.close()'>");
        msg.document.write("</FORM></CENTER>");
        msg.document.write("</BODY>");
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <CENTER>
      <H1>JavaScript példa</H1>
      <FORM>
      <INPUT TYPE="button" VALUE=' ÚJ ABLAK MEGNYITÁSA ' onclick="Ablak()">
    </FORM>
  </CENTER>
</BODY>
</HTML>
```

A HTML speciális karakterei

Ha egy HTML formátumú szövegfájl nem csak az angol ABC alfanumerikus jeleit akarjuk használni, hanem ékezetes betűket vagy speciális jeleket is, akkor a HTML speciális jeleit kell használni. Lehetséges az ESCAPE szekvencia alapján történő jelölése és ISO-kód szerinti megadása is ezen speciális jeleknek.

Az ABC betűi és kódjaik								
Elnevezés	Jel	ESC	ISO		Elnevezés	Jel	ESC	ISO
Nagy A betű	A	A	A		Kis a betű	a	a	a
Nagy Á betű	Á	Á	Á		Kis á betű	á	á	á
Nagy B betű	B	B	B		Kis b betű	b	b	b
Nagy C betű	C	C	C		Kis c betű	c	c	c
Nagy D betű	D	D	D		Kis d betű	d	d	d
Nagy E betű	E	E	E		Kis e betű	e	e	e
Nagy É betű	É	É	É		Kis é betű	é	é	é
Nagy F betű	F	F	F		Kis f betű	f	f	f
Nagy G betű	G	G	G		Kis g betű	g	g	g
Nagy H betű	H	H	H		Kis h betű	h	h	h
Nagy I betű	I	I	I		Kis i betű	i	i	i
Nagy Í betű	Í	Í	Í		Kis í betű	í	í	í
Nagy J betű	J	J	J		Kis j betű	j	j	j
Nagy K betű	K	K	K		Kis k betű	k	k	k
Nagy L betű	L	L	L		Kis l betű	l	l	l
Nagy M betű	M	M	M		Kis m betű	m	m	m
Nagy N betű	N	N	N		Kis n betű	n	n	n
Nagy O betű	O	O	O		Kis o betű	o	o	o
Nagy Ó betű	Ó	Ó	Ó		Kis ó betű	ó	ó	ó
Nagy Ö betű	Ö	Ö	Ö		Kis ö betű	ö	ö	ö
Nagy Ő betű	Ő	Õ	Ô		Kis ő betű	ő	õ	ô
Nagy P betű	P	P	P		Kis p betű	p	p	p
Nagy Q betű	Q	Q	Q		Kis q betű	q	q	q
Nagy R betű	R	R	R		Kis r betű	r	r	r
Nagy S betű	S	S	S		Kis s betű	s	s	s
Nagy T betű	T	T	T		Kis t betű	t	t	t
Nagy U betű	U	U	U		Kis u betű	u	u	u
Nagy Ú betű	Ú	Ú	Ú		Kis ú betű	ú	ú	ú
Nagy Ü betű	Ü	Ü	Ü		Kis ü betű	ü	ü	ü
Nagy Ű betű	Ű	Û	Û		Kis ű betű	ű	û	û
Nagy V betű	V	V	V		Kis v betű	v	v	v
Nagy W betű	W	W	W		Kis w betű	w	w	w
Nagy X betű	X	X	X		Kis x betű	x	x	x
Nagy Y betű	Y	Y	Y		Kis y betű	y	y	y
Nagy Z betű	Z	Z	Z		Kis z betű	z	z	z

Néhány speciális jel és kódjaik								
Elnevezés	Jel	ESC	ISO		Elnevezés	Jel	ESC	ISO
Tabulátor						Soremelés		
	
Szókőz		 			Felkiáltójel	!	!	!
Idézőjel	"	"	"		Számjel	#	#	#
Dollárjel	\$	$	\$		Százalékjel	%	%	%
Angol és jel	&	&	&		Aposztróf	'	'	&
Bal zárójel	(((Jobb zárójel)))
Aszteriszk	*	*	*		Pluszjel	+	+	+
Vessző	,	,	,		Kötőjel	-	-	-
Pont	.	.	.		Perjel	/	/	/
Nullás	0	0	0		Egyes	1	1	1
Kettes	2	2	2		Hármas	3	3	3
Négyes	4	4	4		Ötös	5	5	5
Hatos	6	6	6		Hetes	7	7	7
Nyolcas	8	8	8		Kilences	9	9	9
Kettőspont	:	:	:		Pontosvessző	;	;	;
Kisebb jel	<	<	<		Egyenlőségjel	=	=	=
Nagyobb jel	>	>	>		Kérdőjel	?	?	?
Kukac	@	@	@		Bal zárójel	[[[
Visszaper jel	\	\	\		Jobb zárójel]]]
Hatványjel	^	^	^		Aláhúzás	_	_	_
Vissza aposztróf	`	`	`		Bal kapocs	{	{	{
Függőleges		|			Jobb kapocs	}	}	}
Tilde jel	~	~	~		Alsó aposztróf	,	‚	
Alsó idézőjel	“	„			Kereszt	†	†	
Kettős kereszt	‡	‡			Ezrelék	‰	‰	
Felső vessző	‘	‘			Felső vessző	’	’	
Kettős vessző	“	“			Kettős vessző	”	”	
Szorzás jel	•	•			Mínusz előjel	–	–	
Kivonás jel	—	—			Trade Mark	™	™	
Cent jele	¢	¢	¢		Font jele	£	£	£
Csővezeték jel	¡	¦	¦		Paragrafus jel	§	§	§
Umlaut	“	¨	¨		Copyright	©	©	©
Bal tört idézet	«	«	«		Lágy kötőjel	-	­	­
Registered TM	®	®	®		Fok jele	°	°	°
Plusz-mínusz	±	±	±		Ékezet	˘	´	´
Mikro	μ	µ	µ		Bekezdés vége	¶	¶	¶
Középen pont	·	·	·		Jobb tört idézet	»	»	»
Szorzás kereszt	×	×			Scharfes s	ß	ß	ß
Osztás jel	÷	÷						

Az alábbi kis táblázat tartalmazza a magyar ékezetes betűk Escape-szekvenciáit és ISO-kódját:

Jel	ESCAPE	ISO-kód	Jel	ESCAPE	ISO-kód
Á	Á	Á	á	á	á
É	É	É	é	é	é
Í	Í	Í	í	í	í
Ó	Ó	Ó	ó	ó	ó
Õ	Õ	Ô	õ	õ	ô
Ö	Ö	Ö	ö	ö	ö
Ú	Ú	Ú	ú	ú	ú
Ű	Û	Û	ű	û	û
Ü	Ü	Ü	ü	ü	ü

Karaktertáblák

Az Internet használatának első éveiben, az aktuális feladatok megoldására a 7 bites ASCII kódolás tökéletesen elegendő volt. A 7 bitbe (127 karakter) belefért a teljes angol ABC, a számok és még néhány jel is. Azután a hálózat kezdte átlépni a határokat, és egyre inkább szükség volt az angoltól különböző nyelvek betűinek megjelenítésére is. Eközben a hétbites rendszerekről lassan megkezdődött az áttérés a nyolc bitre (így lett az ASCII-ből ANSI), ezután kézenfekvő volt, hogy a speciális karaktereket a megjelenő 128 üres helyre lehet bekódolni. Ahány ház, annyi szokás: ki így, ki úgy töltötte ki ezt a felső tartományt. A különböző megoldásoknak köszönhetően megszülettek a kódtáblák (codepage), és elkezdődött a káosz. Még messze a DOS-os időkben járunk, amikor a táblázatrajzoló karakterek helyett néha ékezetes karakterek jelennek meg a rossz beállításoknak köszönhetően. A nyugat-európai (western) kódtáblák tartalmazzak ugyan ékezetes betűket, de a magyar hosszú ő és ű már nem fért bele. Sebaj, van hasonló: (sajnos) valószínűleg mindannyian találkoztunk már a kalapos u[^] és hullámos o[~] betűkkel. Akkoriban ez a kis csúsztatás még elviselhetőnek tűnt, manapság viszont már inkább kínos hibának tűnik a megjelenésük. A világ ugyanis azóta sokat fejlődött. A szerteágazó kódtáblákat szabványokká fogták össze, a dokumentumokba beépítették a karaktertáblákat azonosító adatokat, így egy HTML oldal forrásából, vagy egy e-mailből azonnal kiderül, hogy azt a küldője milyen karaktertáblával írta. Bizony, így van ez akkor is, ha a (főleg nem Windows platformon futó) levelezőprogramok zöme erről nem hajlandó tudomást venni. Mindenekelőtt két fontos kódtáblára hívnám fel a figyelmet: az iso-8859-1, más néven Western kódtábla a nyugat-európai karaktereket (és a hullámos/ kalapos o[~]-t és u[^]-t) tartalmazza, míg az iso-8859-2 alias Central European amely tartalmazza a magyar változatot. Hasonló hatást lehet elérni a Windows-1250 nevű kódlappal, ez azonban, mint az a nevéből is kitalálható, nem kifejezetten elterjedt Un*x és Macintosh körökben :-). Magyar szövegben, amikor csak tehetjük, használjuk az iso-8859-2-t!

Cascading Style Sheets (CSS)

Bevezetés

A CSS egy olyan stíluslap megvalósítás, amely lehetővé teszi, hogy a HTML oldalak szerzői oldalaikhoz egyedi stílust rendeljenek hozzá. A CSS egyik alapvető tulajdonsága a folyamatos stíluslap - HTML lap kapcsolat. A lapok szerzői az általuk kedvelt stílust **egyszer** rögzítik, és hozzákapcsolhatják minden általuk készített HTML laphoz. Ez a leírás tartalmazza ennek megoldási lehetőségeit. Stíluslappal (style sheet) tervezni nem nehéz, de szükséges hozzá némi alapvető HTML ismeret. Ezt szemlélteti az alábbi példa:

```
H1 {color: blue}
```

Fenti példa tartalmazza a CSS használatának alapszabályait - **egy css utasítás két részből áll:**

- a **szelektor** tartalmazza a formázandó HTML tag megnevezését (H1);
- a **deklaráció** végzi el a szelektorban meghatározott tag formázását.

Maga a deklaráció is két részre bontható: egy tulajdonságra és a hozzá tartozó értékre (szín: kék). A szelektor a tulajdonképpeni kapocs a HTML, mint leírónyelv és a stíluslap között; szinte minden HTML tag betöltheti a szelektor szerepét. Az előbb említett **szín (color)** tulajdonság csak egy a több, mint 50 közül, amelyek segítségével alakíthatjuk egy HTML oldal kinézetét. (A továbbiakban ahol **oldal**, **dokumentum** olvasható, értelemszerűen **HTML** oldalra, dokumentumra vonatkozik.)

A stíluslap csatolása

A következő példa mutatja be a kapcsolódás négy lehetséges módját:

```
<HTML>
  <HEAD>
    <TITLE>Stíluslapok</TITLE>
    <LINK REL=STYLESHEET TYPE="text/css" HREF="http://style.com/sajat"
TITLE="sajat">
    <STYLE TYPE="text/css">
      @import url(http://style.com/sajat)
    <!--
      H1 {color: blue}
    -->
    </STYLE>
  </HEAD>

  <BODY>
    <H1>A címsor kék</H1>
    <P STYLE="color: green">Az egész bekezdés zöld</P>
  </BODY>
</HTML>
```

A fenti példában látható első lehetőség a csatolásra a <LINK> tag használata; külső stíluslap behívására. Második lehetőségként a dokumentum HEAD szekciójában elhelyezett <STYLE> tag, ahol közvetlenül definiálhatók a használni kívánt stílusok, vagy az '@import' kulcsszóval külső stíluslap importálható. Az utolsóként bemutatott lehetőség pedig valamely HTML elem STYLE attribútumának használata, a dokumentum BODY szekciójában.

A böngészők általában figyelmen kívül hagyják az általuk ismeretlen elemeket. Ezért a régebbi böngészők jó esetben egyszerűen 'elmennek' a <STYLE> elem mellett. Kellemetlenebb eredménnyel jár, ha belezavarodnak tartalmába. Ez megelőzhető, ha egy standard SGML utasítással elrejtjük előlük:

```
<STYLE TYPE="text/css">
  <!--
    H1 {color: blue}
  -->
</STYLE>
```

Csoportosítás

A stíluslapok méretének csökkentése érdekében a szelektorok csoportosíthatók; vesszővel elválasztott listába szedve:

```
H1, H2, H3 {font-family: verdana}
```

Hasonló módon a deklarációk is csoportosíthatók:

```
H1
{
font-family: helvetica;
font-size: 12pt;
font-style: normal;
}
```

Néhány tulajdonság eltérő csoportosítási szintaktikát is megenged:

```
H1 {font: bold 12pt helvetica}
```

Öröklődés

Az első példában a H1 elem színbeállítását mutattuk be. A következőkben tartalmaz egy elemet is:

```
<H1>A címsor <EM>mindig</EM> fontos.</H1>
```

Ha az elemhez nincs külön szín rendelve, a kiemelt (emphasized) "mindig" szó *mindig* öröklí a tartalmazó (container) elem színét, jelen esetben a kéket. Hasonlóképpen más stílus-tulajdonságok is öröklődnek (pl.:font-family, font-size). Alapértelmezett stílustulajdonság beállításához az alkalmazni kívánt stílust olyan elemhez kell kötni, amely tartalmazza mindazokat az elemeket, amelyekre a stílust vonatkoztatni akarjuk. A HTML dokumentumokban erre a célra megfelelhet a <BODY> elem:

```
BODY
{
color: black;
background: url(hatter.gif) white;
}
```

Ez a stíluspélda a BODY szövegszínét feketére állítja be, háttérként képet rendel hozzá. Ha a kép nem érhető el, a háttérszín fehér lesz. Van azonban néhány stílustulajdonság, amely nem öröklődik (Ezek közé tartozik pl. a background), de a szülő elem háttértulajdonságát néhány *tartalmazott* (contained) elem láthatóan hagyja. (Ha egy táblázatnak nem adunk meg 'background' /háttér/ tulajdonságot, átlátszik rajta a BODY háttére.)

A tulajdonságok értéke százalékban is megadható:

```
P {font-size: 10pt}
P {line-height: 120%}
```

Ebben a példában a 'line-height' tulajdonság értéke a 'font-size' tulajdonság értékének 120% -a lesz: 12 pont.

A class szelektor

A HTML elemek stílusbeállítási rugalmasságának növelése érdekében a W3C új attribútumot vett fel a HTML-be: ez a **CLASS** (osztály). A 'BODY' minden eleme osztályba sorolható, az egyes osztályok pedig stíluslapból megcímezhetők.

```
<HTML>
<HEAD>
<TITLE>Stíluslap példa</TITLE>
<STYLE TYPE="text/css">
H1.mezei {color: #00ff00}
</STYLE>
</HEAD>
<BODY>
<H1 CLASS=mezei>Zöld, mint a rét</H1>
</BODY>
</HTML>
```

Az osztályba sorolt elemekre az öröklődés általános szabályai vonatkoznak. Öröklik a stílusértékeket a dokumentum struktúrájában felettük álló ún. *szülő* elemeiktől. Minden azonos osztályba tartozó elem megcímezhető egyszerre is, elhagyva a hozzájuk tartozó tag nevét:

```
.mezei {color: green}
```

```
/*minden mezei osztályba tartozó elem (CLASS=mezei)*/
```

Megjegyzendő, hogy szelektoronként (HTML elemenként) csak egy osztály definiálható!

Az ID szelektor

A HTML -be felvételre került az 'ID' attribútum is, amely lehetővé teszi az egyedi azonosítók felvételét a dokumentumba. Ennek a lehetőségnek különleges jelentőséget ad, hogy felhasználható stíluslap szelektorként, és megcímezhető a '#' előtaggal.

```
#z98y {letter-spacing: 0,3em}
```

```
H1#z98y {letter-spacing: 0,5em}
```

```
<P ID=z98y>Széles szöveg</P>
```

Az fenti példa első esetében a kiválasztott formázást a 'P' elemhez feeltettük meg, az 'ID' attribútumérték segítségével. A második esetben kettős feltételt támasztottunk: a formázás akkor lép érvénybe, ha a H1 elemet a '#z98y' azonosítóval (ID) látjuk el. Ezért ez már nem vonatkozik a 'P' elemre. Az ID attribútum szelektorként való használatával HTML elemekre alapozott stílustulajdonságok állíthatók be.

Összekapcsolt szelektorok

Az öröklődés szabályai mentesítik a stíluslap tervezőjét egy csomó fölösleges gépelés alól. A tulajdonságok beállítása során elég egyszer elkészíteni az alapértelmezettet, utána felsorolni a kivételeket. Ha az 'EM' elemnek a 'H1' elemen belül más színt szeretnénk meghatározni, azt a következőképp tehetjük meg:

```
H1 {color: blue}
```

```
EM {color: red}
```

Mikor a stíluslap aktív, minden kiemelt () szövegrész, akár a H1 elemen belül, akár azon kívül található, vörösre változik. Abban az esetben, ha csak egyszer akarjuk a H1 -en belül az EM elemet vörösre színezní, a CSS kódot az alábbiak szerint kell megváltoztatni:

```
H1 EM {color: red}
```

Összekapcsolt szelektorok használata esetén azok az elemek lesznek megcímezve, amelyek a felsorolásban utoljára állnak. Tehát akár kettőnél több elem is 'egymásba ágyazható' ilyen módon.

Megjegyzések

A CSS kódban elhelyezett megjegyzések hasonlóak a C programnyelvben elhelyezett megjegyzésekhez: A megjegyzések nem ágyazhatók egymásba, illetve más CSS utasításba.

```
EM {color: red} /* A kiemelt szövegrész vörös!*/
```

Látszólagos osztályok és elemek

A CSS-ben a beállítandó stílus alapesetben egy HTML elemhez van kapcsolva; ez a kapcsolat az elemnek a dokumentum-struktúrában elfoglalt helyére alapozódik. Ez az egyszerű modell a stíluslapalkalmazás viszonylag széleskörű lehetőségét nyújtja, de nem nyújt lehetőséget az összes lehetséges megjelenítés alkalmazására.

A látszólagos osztályok és elemek a HTML leírásban nem szerepelnek (ezért látszólagosak), mégis köthetőek szelektorokhoz. Tulajdonképpen a böngésző által átadott és a stíluslapon keresztül értelmezett címzési módról van szó. Ugyanúgy kell rájuk hivatkozni, mint bármely

elemre, vagy osztályra; ez a szabványos eljárás viselkedésük leírására. Pontosabban: viselkedésük tagek elméleti sorozataként írható le.

A látszólagos elemek az elemek részelemeinek megcímzésére használhatók, a látszólagos osztályok pedig lehetővé teszik a stíluslapon keresztül történő elemtípus megkülönböztetést.

Látszólagos osztályok az élőkapcsokban

A böngészők közös tulajdonsága, hogy másképp jelenítik meg a látogatott linkeket, mint a még nem látogatottakat. Ezt a tulajdonságot a CSS az <A> elem látszólagos osztályain keresztül kezelni tudja:

```
A: link {color: red}
```

```
A: visited {color: blue}
```

```
A: active {color:lime}
```

Minden 'HREF' attribútummal rendelkező <A> elem a fenti csoportból egyet és egy időben csak egyet jelöl ki. A böngészők pedig kiválaszják, hogy az adott linket -állapotától függően- milyen színnel jelenítsék meg. Állapotukat a látszólagos osztály határozza meg:

- **link** - Nem látogatott hivatkozás;
- **visited** - Már látogatott hivatkozás;
- **active** - Amelyik hivatkozás éppen ki van választva (egérkattintással).

Egy élőkapocs látszólagos osztályának formázása ugyanúgy történik, mintha az osztály külön volna definiálva. A böngészők nem követelik meg az aktuálisan megjelenített dokumentum újrabetöltését, amikor egy élőkapocs látszólagos osztálya által meghatározott változtatás esedékessé válik. (Pl.: a CSS szabványos eljárása lehetővé teszi az 'active' link 'font-size' tulajdonságának futásidejű megváltoztatását úgy, hogy az aktív dokumentumot nem kell újra betöltenie a böngészőnek, mikor az olvasó kiválaszt egy 'visited' linket.). A látszólagos osztályok nem feleltethetők meg a normál osztályoknak és fordítva; ezért az alábbi példában bemutatott stílusszabályok nem befolyásolják egymást:

```
A: link {color: red}
```

```
<A CLASS=link NAME=target5>...</A>
```

Az élőkapocs látszólagos osztályoknak nincs hatásuk az 'A' -n kívül más elemre. Ezért az elemtípus el is hagyható a szelektorból:

```
A: link {color: red}
```

```
:link {color: red}
```

Fenti két deklarációban a szelektor ugyanazt az elemet fogja kiválasztani. A látszólagos osztályok nevei kis- és nagybetűérzékenyek. A látszólagos osztályok használhatóak a kapcsolódó szelektorokban is:

```
A: link IMG {border: solid blue;}
```

A látszólagos osztályok kombinálhatók a normál osztályokkal:

```
A.external: visited {color: blue}
```

```
<A CLASS=external HREF="http://valahol.mashol.com">Külső (external)  
hivatkozás</A>
```

Ha a fenti példában levő hivatkozás látogatottá válik (visited), színe kékre változik. Megjegyzendő, hogy a normál osztályok neveinek a látszólagos osztályok neveit meg kell előznie a szelektorbán.

Tipografikai látszólagos elemek

Néhány közös megjelenítési effektus nem strukturális elemhez kapcsolható, hanem inkább a képernyőn elemeket kirajzoló tipografikai tulajdonságokhoz. A CSS -ben két ilyen tipografikai tétel címezhető meg látszólagos elemen keresztül: egy elem tartalmának első sora és az első betű.

A 'first-line' látszólagos elem

A 'first-line' látszólagos elem az első sor különleges formázásához használható:

```
<STYLE TYPE="text/css">
  P:first-line {font-variant: small-caps}
</STYLE>
```

A tagek elméleti sorozata a következőképp néz ki:

```
<P>
<P:first-line>
A szöveg első sora
</P:first-line>
kiskapitális betűkkel jelenik meg.
</P>
```

A 'first-line' látszólagos elem használata hasonló a soron belüli elemekhez, azonban figyelembe kell venni néhány megszorítást. Csak a következőkben felsorolt tulajdonságok alkalmazhatók hozzá:

- Betűtípus tulajdonságok
- Szín- és háttér tulajdonságok
- 'word-spacing' ,
- 'letter-spacing' ,
- 'text-decoration' ,
- 'vertical-align' (csak, ha a 'float' tulajdonság értéke 'none';),
- 'text-transform',
- 'line-height',
- 'clear'.

A 'first-letter' látszólagos elem

A 'first-letter' látszólagos elem gyakran előforduló használati lehetősége az iniciálé kialakítása, ami gyakran használt tipográfiai effektus. A következőkben felsorolt tulajdonságok alkalmazhatók hozzá:

- Betűtípus tulajdonságok,
- Szín- és háttér tulajdonságok,
- 'text-decoration',
- 'vertical-align' (csak, ha a 'float' tulajdonság értéke 'none';),
- 'text-transform',
- 'line-height',
- margó tulajdonságai,
- helykitöltő (padding) tulajdonságok,
- szegélytulajdonságok,
- 'float',
- 'clear'.

A következő példa bemutatja, hogyan készíthető kétsoros iniciálé:

```
<HTML>
<HEAD>
<TITLE>Lapcím</TITLE>
<STYLE TYPE="text/css">
P: {font-size: 12pt; line-height: 12pt}
P:first-letter: {font-size: 200%; float:left}
</STYLE>
```



```

</HEAD>
<BODY>
<P>
A sor első betűje kétszer akkora lesz, mint a többi.
</P>
</BODY>
</HTML>

```

A böngészőtől függ, mely karakterek tartoznak a 'first-letter' elemhez. Általában a bevezető idézőjelet is belefoglalják. A 'first-letter' látszólagos elem csak blokk szintű elemhez kapcsolható.

Látszólagos elemek a szelektorokban

Kapcsolódó szelektorok esetén a látszólagos elemeknek a szelektor utolsó elemeként kell szerepelniük:

```
BODY P:first-letter {color: purple}
```

A látszólagos elemek kombinálhatóak a szelektorban az osztályokkal is:

```
P.alap:first-letter {color: red}
```

```
<P CLASS=alap>Első bekezdés</P>
```

A fenti példában látható stílusmeghatározás az összes olyan P elem első betűjét bíborra színezi, amelynek osztálya 'alap' (class=alap). Ha látszólagos elemeket osztályokkal, vagy látszólagos osztályokkal kombinálunk, a látszólagos elemet a szelektor utolsó tagjaként kell elhelyezni. Egy szelektorban csak egy látszólagos elem lehet elhelyezve.

Látszólagos elemek többszörözése

Néhány látszólagos elem kombinálható:

```
P {color: red; font-size: 12pt}
```

```
P:first-letter {color: green; font-size: 200%}
```

```
P:first-line {color: blue}
```

A fenti példa minden 'P' elem első betűjét zöldre, a betűméretet pedig 24 pontosra állítja. Az első sor többi része kék lesz, a többi része pedig vörös.

Megjegyzendő, hogy a 'first-letter' elemet a 'first-line' elem tartalmazza. A 'first-line' elemre beállított tulajdonságokat a 'first-letter' elem örökli, de az öröklődés szabályánál a 'first-letter' elemre külön beállított tulajdonság-érték erősebb.

Rangsor

Ebben a fejezetben a terminológia egy új elemét kell bevezetnünk: a "súlyt". A stíluslap-szabályok súlya jelzi, hogy az adott szabály milyen prioritást élvez. Természetesen a nagyobb súlyú szabály erősebben érvényesül, mint az alacsonyabb súllyal rendelkező.

CSS használatával egyidejűleg egynél több stíluslap is befolyásolhatja a HTML oldal megjelenését. E tulajdonságnak két fő oka van: a modularitás és a szerző (tervező) / olvasó - egyensúly.

- **Modularitás** A stíluslap tervezője a redundancia csökkentése érdekében több stíluslap használatát kombinálhatja:

```
@import url(http://www.style.org/egyik);
@import url(http://www.style.org/masik);
H1 {color: red} /* Felülbírálja az importált stílust */
```
- **Szerző / olvasó egyensúly:** Stíluslappal a szerző és az olvasó is befolyásolhatja a dokumentum megjelenését. Ehhez mindkettjüknek ugyanazt a stílusnyelvet kell használniuk, így tükrözve a web egyik alapvető tulajdonságát: mindenki közzéteheti elképzelését. A böngészők a saját stíluslapokra hivatkozás kiválasztását szabadon lehetővé teszik.

Néha ellentét merül fel a dokumentum megjelenését meghatározó stíluslapok között. Az ellentét feloldását a stílusszabályok súlyozása teszi lehetővé. Alapértelmezésben az olvasó által felállított stílusszabályok súlya kisebb, mint a dokumentum szerzője által felállítottaké. Tehát, ha ellentét merül fel egy dokumentum szerzői és olvasói stíluslapja között, a szerzői stíluslap kerül alkalmazásra. Mind a szerzői, mind az olvasói stíluslap-szabályok felülbírálják a böngésző alapértelmezett beállításait.

Az importált stíluslapok szintén jól meghatározott rangsorban állnak egymással. A rangsor az alább meghatározott szabályok szerint dől el. Bármely szabály, amely a stíluslapban van leírva, erősebb, mint az importált stíluslap(ok)ban levő(k). Tehát, az importált stíluslapokban leírt szabályok súlya kisebb a stílusok rangsorában, mint a stíluslapban magában leírt szabályoké. Az importált stíluslapok maguk is rekurzívan importálhatnak és ezáltal felülbírálnak más stíluslapokat.

A CSS minden `@import` utasításának a stíluslap legelején kell megjelenennie, megelőzve minden más deklarációt. Ez megkönnyíti annak átláthatóságát, hogy melyik stíluslap melyik másikat bírálja felül.

'important'

A stíluslapok tervezői deklarációik súlyát megnövelhetik:

```
H1 {color: black ! important; background: white ! important }
P {font-size: 12pt ! important; font-style: italic }
```

Ebben a példában az első három deklaráció súlya a *fontos!* kiemeléssel meg van növelve, míg az utolsó deklaráció súlya normál. Egy olvasó által felállított *fontos* szabály prioritásban megelőzi a szerző normál súlyú szabályát. A szerző által felállított *fontos* szabály prioritásban megelőzi az olvasó által felállított *fontos* szabályt.

A rangsor felállítása

A deklaráció-rangsor felállításának szabályai lényegesek a CSS működésében. Egy elem/tulajdonság páros értékének meghatározásához az alábbi algoritmust kell követni:

1. A kérdéses elem/tulajdonság párosra alkalmazott összes deklaráció előkeresése. A deklaráció akkor 'alkalmazott', ha a kérdéses elem szelektorként szerepel. Ha nincs az elemhez alkalmazott deklaráció, az öröklés szabályai érvényesülnek. Ha nincs örökölt érték (pl.: a HTML elem esetében), a kezdeti értékek lesznek irányadók.
2. A deklarációk explicit súlya szerinti rendezés: az `!important` jelölésű deklarációk erősebbek, mint a jelöletlen (normál) deklarációk.
3. Eredet szerinti rendezés: A szerző stíluslapjában meghatározott szabályok mint az olvasó által meghatározottak; mindkettő erősebb a böngésző alapértelmezett beállításainál. Az importált stíluslapok eredete megegyezik annak a stíluslapnak az eredetével, ahonnan importálták.
4. Szelektor egyedisége szerinti rendezés: Az egyedileg meghatározott szelektorhoz tartozó szabály erősebb, mint az általános meghatározásban leírtak. Az egyediség megállapításához meg kell állapítani a szelektorban található ID attribútumokat (a), a CLASS attribútumokat (b), és a tag-nevek számát (c). A három szám 'összeláncolásával' (concatenating) állapítható meg az adott szelektor egyedisége. A könnyebb megértés kedvéért néhány példa:

```
LI          { ... } /* a=0 b=0 c=1 => egyediség = 1 */
UL LI      { ... } /* a=0 b=0 c=2 => egyediség = 2 */
```

OL UL LI	{...}	/*	a=0	b=0	c=3	=>	egyediség = 3	*/
LI.red	{...}	/*	a=0	b=1	c=1	=>	egyediség = 11	*/
OL UL LI.red	{...}	/*	a=0	b=1	c=3	=>	egyediség = 13	*/
#x34y	{...}	/*	a=1	b=0	c=0	=>	egyediség = 100	*/

A látszólagos elemeket és látszólagos osztályokat a számítás során normál elemekként, osztályokként kell figyelembe venni.

5. Rendezés a meghatározás sorrendje szerint: Ha két szabály ugyanakkora súllyal bír, a később meghatározott győz. Az importált stíluslapban leírt szabályok a saját lapban írtak után lesznek csak figyelembe véve.

A tulajdonság-értékek keresése megszakítható, ha az egyik kiértékelt szabály súlya egy elem/tulajdonság vonatkozásban egyértelműen nagyobb bármely más szabályénál.

Egy elem `STYLE` attribútumában történő deklarációnak ugyanolyan súlya van, mint egy - a stíluslap végén meghatározott - `ID` alapú szelektornak.

```
<STYLE TYPE="text/css">
  #x97z {color: blue}
</STYLE>
```

```
<P ID=x97z STYLE="color: red">
```

A fenti példában a `P` elem színe vörös (red) lesz. Bár mindkét deklarációs forma egyedisége megegyezik, mégis - a rangsor-meghatározás 5. pontjában írt szabály alkalmazása miatt - a `STYLE` attribútumban levő deklaráció erősebb lesz a `STYLE` elem által tartalmazott deklarációnál.

Formázásmodell

A CSS egy egyszerű, dobozszerű formázási modellt használ, ahol minden elemformázás eredménye egy, vagy több négyszögletes dobozként képzelhető el. Minden doboznak van egy



'maga', az öt körülvevő 'kitöltéssel' (padding), szegéllyel (border) és margóval (margin).

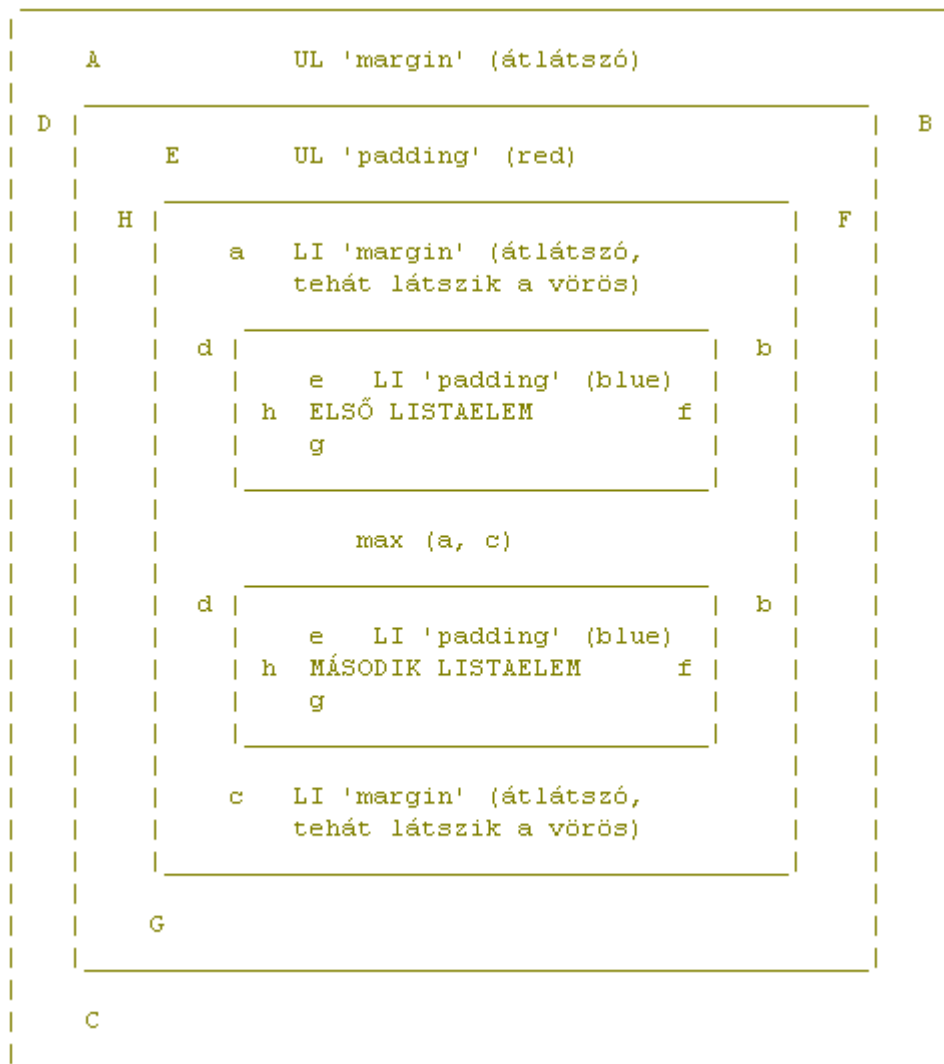
A margó, a szegély és a kitöltés mérete egyenként meghatározható a **margin**, a **border** és a **padding** tulajdonságok értékeinek beállításával. A kitöltőterület hátérszíne megegyezik az

elemével, amelyet a **background** tulajdonság határoz meg. A szegély színe és stílusa szintén a **border** tulajdonság beállításával határozható meg, míg a margó *mindig* átlátszó, így a szülő elem állandóan látható marad. A doboz szélessége az *elem*, a *kitöltés*, a *szegély* és a *margó* területének összege. A formázások szempontjából két elemtípus különböztethető meg: a >blokkszintű elem és a soron belüli elem.

Blokkszintű elemek

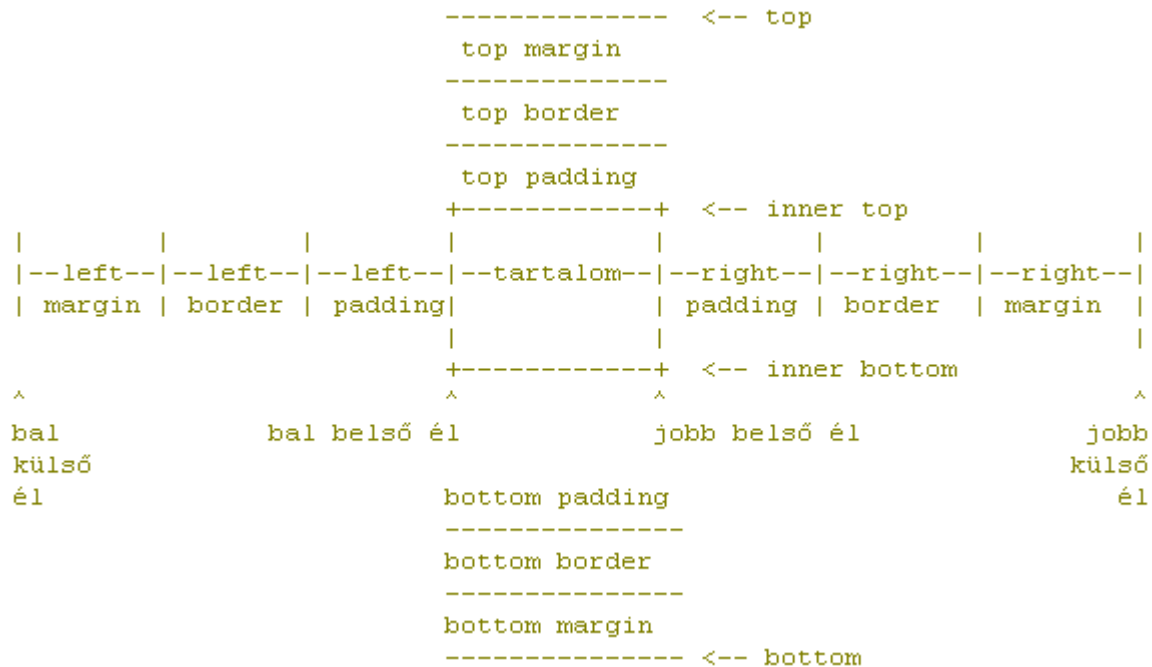
A következő példa bemutatja, hogyan formázható CSS használatával egy két gyermek-elemet tartalmazó **UL** elem. A példa egyszerűsítése okán szegélyeket nem definiálunk. Szintén ez okból használjuk értékeként az ABC kis- és nagybetűit, amelyek nem szabványos CSS meghatározások, de ebben az esetben használatuk megkönnyíti a táblázat és a példastílus összevetését.

```
<STYLE TYPE="text/css">
  UL {
    background: red;
    margin: A B C D;
    padding: E F G H;
  }
  LI {
    color: white;
    background: blue;          /* fehér szöveg kék háttéren */
    margin: a b c d;
    padding: e f g h;
  }
</STYLE>
...
<UL>
  <LI>Első elem
  <LI>Második elem
</UL>
```



Gyakorlatilag, a kitöltés és a margó tulajdonságai nem öröklődnek. De, ahogy a példa is mutatja, az elemek szülőkhöz és 'testvérekhez' (velük egyenrangúakhoz) viszonyított elhelyezkedése lényeges, mert ezeknek az elemeknek kitöltés- és margótulajdonságai hatást gyakorolnak a gyermek elemekre. Ha az előző példában lettek volna szegélyek is, azokat a kitöltés és a margó között kellett volna megjeleníteni.

A következő ábra a dobozmodell használatakor érvényes terminológiáról nyújt áttekintést:



A jelölések fordítása szándékosan maradt el, ugyanis ugyanezek a kifejezések szerepelnek a CSS nyelvtanában is. Azok kedvéért, akik számára az angol nyelv bizonyos nehézséget okoz, az alábbiakban rövid szószeretet teszünk közzé:

bottom	- alsó
left	- bal
top	- felső
right	- jobb
margin	- margó
border	- szegély
padding	- kitöltés
inner	- belső

E kis kitérő után térjünk vissza a dobozmodellhez:

A *bal külső él* a formázott elem bal széle; a hozzá tartozó kitöltéssel, szegéllyel és margóval. A bal belső él pedig magának a tartalomnak a szélét jelenti, kitöltés, szegély és margó *nélkül*. A jobb oldal ennek tükörképe. Ugyanez a szabály érvényes az elem csúcsára (top) és aljára (bottom).

Egy elem szélességén a *tartalom* szélességét értjük, vagyis a bal belső él és a jobb belső él közti távolságot. Az elem magasságán pedig a tartalom magasságát, azaz a rajzon **inner top** és **inner bottom** kifejezéssel jelölt magasságok közti távolságot.

Függőleges formázás

A nem 'lebegő' blokk szintű elemek margószélessége határozza meg az öt körülvevő doboz szélei közti minimális távolságot. Két, vagy több összevont szomszédos margó (értsd: nincs közöttük szegély, kitöltés, vagy tartalom) közösen használja a stíluslapban meghatározott legnagyobb 'margin' értéket. A legtöbb esetben a függőleges margók összevonás után vizuálisan kellemesebb látványt nyújtanak, mint amit a tervező gondolt. Az előbbi példában a két LI elem közötti margó van összevonva, és közösen használják az első LI elem '**margin-bottom**' és a második LI elem

'**margin-top**' tulajdonsága közül a nagyobbbat. Hasonlóképpen; ha az UL elem és az első LI elem közti kitöltés (padding - az E jelű konstans) 0 értékű, az UL elem és az első LI elem közti margó összevonódik.

Vízszintes formázás

A nem 'lebegő' blokkszintű elemek vízszintes pozícióját és méretét hét tulajdonság határozza meg: a 'margin-left', 'border-left', 'padding-left', 'width', 'padding-right', 'border-right', 'margin-right'. Ezen tulajdonságok értékeinek összege mindig megegyezik a szülő elem 'width' értékével. Alapértelmezésben egy elem szélességének értéke 'auto'. Ha az elem nem helyettesített elem, a szélesség értékét a böngésző számolja ki, a fent említett hét tulajdonság értékének összegzésével. Ha az elem helyettesített elem, 'width' tulajdonságának 'auto' értéke automatikusan kicserélődik az elem saját szélesség-értékére. A hét tulajdonság közül háromnak értéke állítható 'auto' -ra: 'margin-left', 'width', 'margin-right'. A 'width' tulajdonságnak van egy nemnegatív, böngésző által meghatározott minimum értéke (amely elemenként változik és mindig másik tulajdonságtól függ). Ha a 'width' értéke e határérték alá kerül, akár azért, mert így lett beállítva, vagy mert értéke 'auto' volt és az alábbi szabályok miatt vált értéke túl alacsonnyá: értéke kicserélődik a minimumként használható értékkel. Ha *pontosan egy* 'margin-left', 'width', vagy 'margin-right' értéke 'auto'; a böngésző ennek értékét úgy állítja be, hogy kiadja a fentebb felsorolt hét tulajdonsághoz tartozó maradék értéket; amennyi szükséges a szülő elem szélességének eléréséhez. Ha a tulajdonságok közül *egyik sem* 'auto'; a 'margin-right' tulajdonság értéke lesz 'auto'. Ha a háromból *egynél több* 'auto' és közülük egyik a 'width'; a többi tulajdonság ('margin-left' és/vagy 'margin-right') értéke nullára lesz állítva, és a 'width' kapja meg azt az értéket, amennyi szükséges a szülő elem szélességének eléréséhez. Abban az esetben, ha a 'margin-left', és 'margin-right' tulajdonságok vannak 'auto' -ra állítva, mindkettő egyenlő értéket kap. A függőleges margókkal ellentétben, a vízszintes margók nem összevonhatók.

Listaelemek

Azok az elemek, amelyek 'list-item' tulajdonságának értéke 'display', blokkszintű elemekként formázódnak, és listajelölő előzi meg őket. A jelölő típusát a **list-style** tulajdonság határozza meg. A **list-style** tulajdonság értéke szerint a jelölő elhelyezkedése az alábbi lehet:

```
<STYLE TYPE="text/css">
  UL          {list-style: outside}
  UL.compact {list-style: inside}
</STYLE>

<UL>
  <LI>Első listaelem tartalma
  <LI>Második listaelem tartalma
</UL>

<UL CLASS="compact">
  <LI>Első listaelem tartalma
  <LI>Második listaelem tartalma
</UL>
```

A fenti példát a böngésző így jeleníti meg:

* Az első listaelem
tartalma

* A második listaelem tartalma

* Az első listaelem tartalma

* A második listaelem tartalma

Jobbról-balra értelmezendő szövegek esetében értelemszerűen a listajelölő a doboz jobb oldalára kerül.

Lebegő elemek

A 'float' tulajdonság alkalmazásával egy elem elhelyezkedése meghatározható az elemek által általában elfoglalt helyek oldalsó vonalán kívül is; ez esetben formázásuk módja a blokkszintű elemekéhez hasonló. Példaként: ha egy kép 'float' tulajdonságát 'left' (bal) értékre állítjuk be, eredményképpen a kép addig tolódik balra, míg egy másik blokkszintű elem margóját, kitöltését, vagy szegélyét el nem éri. Jobb oldalán a lap többi tartalma fogja körbefutni. Az elem margói, szegélyei, kitöltései megmaradnak, de margói a szomszédos elem margóival *sosem* lesznek összevonva. A lebegő elemek pozicionálása a következő szabályok szerint történik

1. Egy balra igazított lebegő elem bal külső éle nem lehet balrább, mint szülő eleme bal belső éle. A jobbra igazított lebegő elemekre ugyanilyen szabály érvényes.
2. Egy balra igazított lebegő elem bal külső élének jobbra kell esnie bármelyik (a HTML forrásban) korábban leírt balra igazított lebegő elemtől, vagy tetejének kell lejjebb lennie, mint az előbb leírt aljának. A jobbra igazított lebegő elemekre ugyanilyen szabály érvényes.
3. Egy balra igazított lebegő elem jobb külső éle nem eshet jobbra bármely tőle jobbra eső, jobbra igazított lebegő elem bal külső élétől. A jobbra igazított lebegő elemekre ugyanilyen szabály érvényes.
4. A lebegő elem teteje nem lehet magasabban, mint szülő elemének belső magassága (inner top).
5. A lebegő elem teteje nem lehet magasabban, mint bármely korábbi lebegő, vagy blokkszintű elem teteje.
6. Egy lebegő elem teteje nem lehet magasabban, mint bármely 'sordoboz', amelynek tartalma a HTML forrásban megelőzi a lebegő elemet.
7. A lebegő elemet olyan magasra kell elhelyezni, amennyire csak lehet.
8. A balra igazított lebegő elemet annyira balra kell elhelyezni, amennyire csak lehet; a jobbra igazított lebegő elemet pedig annyira jobbra kell elhelyezni, amennyire csak lehet. Ez az igazítási mód előnyben részesül a többi (más) balra / jobbra igazítási eljárással szemben.

```
<STYLE TYPE="text/css">
  IMG {float: left}
  BODY, P, IMG {margin: 2em}
</STYLE>
```

```
<BODY>
  <IMG SRC="kep.gif">
  Példa szöveg, melynek nnincs más szerepe, ...
</BODY>
```

A fenti példaszöveg formázása a következőképp alakul:

max(BODY margin, P margin)				
				Példaszöveg, melynek nincs
B	P	IMG margók		más szerepe, minthogy
O				bemutassa, hogy a lebegő
D	m			elem a szülő elem szélére
Y	a	IMG		igazodik, miközben meg-
	r			tartja annak margóját,
m	g			szegélyét és kitöltését.
a	ó			Megjegyzendő, hogy a szom-
r				szédos függőleges margók
g				összevonódnak a nem-lebegő
ó				blokkszintű elemek között.

Megjegyzés: a lebegő **IMG** elemet a **P** elem tartalmazza, tehát jelen esetben az a szülő elem. Van két olyan eset, amikor egy lebegő elemek margó-, szegély-, és kitöltő területei átfedhetik egymást:

- Amikor a lebegő elemnek negatív értékű margója van: a lebegő elem negatív margója úgy lesz figyelembe véve, mint más blokkszintű elemé.
- Amikor a lebegő elem szélesebb, vagy magasabb, mint az őt tartalmazó elem.

Soron belüli elemek

Azok az elemek, amelyek nem formázhatók blokkszintű elemként; a soron belüli elemek. Egy soron belül használható több soron belüli elem is. Figyeljük meg a példát:

```
<P>Néhány <EM>kiemelt</EM> szó jelenik meg a <STRONG>sorban</STRONG>
```

A **P** elem általánosan használt blokkszintű elem, míg az **EM** és a **STRONG** soron belüli elemek. Ha a **P** elem szélessége egy egész sort kitölt, akkor abban a sorban két soron belüli elem is található lesz.

Néhány *kiemelt* szó jelenik meg a **sorban**

Ha a **P** elem szélessége több, mint egy sort foglal el, a soron belüli elem 'doboz' felosztódik a két sor között:

```
<P>Néhány <EM>kiemelt szó jelenik</EM> meg; a sorban
```

Néhány *kiemelt* szó
jelenik meg a sorban.

Ha egy soron belüli elemhez margó, szegély, kitöltés, vagy szövegdekoráció van rendelve, ezek nem lesznek hatással a sortörésre.

Helyettesített elemek

A helyettesített elem olyan elem, amely megjelenítésekor avval a tartalommal van helyettesítve, amelyre hivatkozásként mutat. Például a egy HTML kódban szereplő **IMG** elem az **SRC** attribútumában szereplő képpel lesz megjelenésekor helyettesítve. Ilyenkor a helyettesített elemek magukkal hozzák saját méreteiket is. Ha a **width** értéke **auto**, az elem szélességét megjelenítésekor saját szélessége határozza meg. Ha stíluslapon keresztül más (nem **auto**) érték van meghatározva, a helyettesített elem azzal az értékkel kerül megjelenítésre. (A méretezési eljárás a média típusától függ.) A **height** tulajdonságra hasonló szabályok vonatkoznak. Helyettesített elemek lehetnek soron belüli, vagy blokkszintű elemek is.

Sorok magassága

Minden elemnek van **line-height**, azaz sormagasság tulajdonsága, amely egy szövegsor teljes magasságát adja meg. A szövegsor teljes magasságát a szöveg és az alá-fölé elhelyezett üres helyek összessége adja meg. Példaként: ha egy szöveg 12pt magas, és a **line-height** értéke 14pt; két képpontnyi üres hely adódik hozzá a betűmérethez: egy képpont a sor fölé, egy képpont alá.

A betűméret és a sormagasság közti különbséget *vezetésnek* nevezzük. A vezetés-érték fele a *fél-vezetés*. Formázáskor minden szövegsor egy-egy *sordoboz*ként értelmezhető.

Ha a szövegsor egyes részei különböző magasság-értékeket tartalmaznak (több különböző soron belüli elem), akkor minden egyes résznek megvan a maga saját vezetés-értéke is. A sordoboz magasságát a legmagasabb rész tetejétől a legmélyebbre érő rész legaljáig mérjük. Megjegyzendő, hogy a 'legmagasabb' és a 'legmélyebb'értékek nem szükségszerűen tartoznak ugyanazon elemhez, mivel az elemek függőleges pozicionálása a **vertical-align** tulajdonság beállításával történik. Egész bekezdés formázása esetén a minden sordoboz közvetlenül az előző alá kerül. A nem helyettesített elemekhez tartozó alsó és felső kitöltések, szegélyek és margók nem befolyásolják a sordoboz magasságát. Más szóval: ha a kiválasztott kitöltéshez, vagy szegélyhez tartozó **line-height** tulajdonság túl kicsi, a szomszédos sorok szövegei rá fognak takarni.

A sorban található helyettesített elemek (pl.: képek) megnövelhetik a sordoboz magasságát, ha a helyettesített elem teteje (beleértve a hozzá tartozó kitöltést, szegélyt, margót is) a legmagasabb szövegrész fölé, illetve alja a legalacsonyabbra érő szövegrész alá ér.

Általános esetben, amikor az egész bekezdésnek egy **line-height** értéke van, és nincsenek nagyobb méretű képek, a fenti szabályleírás biztosan meghatározza, hogy a párhuzamos sorok alapvonalai pontosan a **line-height** tulajdonságban megadott értékre legyenek egymástól.

A Vászon

A Vászon (CANVAS) a böngészőfelület azon része, amelyre a dokumentum és beágyazott elemei kirajzolódnak. A vászon *nem* a dokumentum strukturáló elemeinek felel meg, és ez a dokumentum formázása során két fontos kérdést vet fel.

- A beállítás során honnan számítjuk a vászon méreteit?
- Amikor a dokumentum nem fedi le az egész vászont, az üres terület hogyan lesz kirajzolva?

Az első kérdésre adható válasz az, hogy a vászon kezdeti szélessége a böngészőablak méretén alapul, de a CSS a pontos méretbeállítást a böngészőre hagyja. Ésszerű a böngészőre hagyni a vászon méretének megváltoztatását az ablak átméretezésekor is, mert ez szintén kívül esik a CSS hatókörén.

A HTML kiterjesztései szolgáltatnak példát a második kérdés megválaszolásához: A **BODY** elem attribútumai állítják be a háttérrel; az egész vászonra vonatkozóan. A laptervezők elvárásainak támogatására a CSS-ben bevezetésre került egy, a vászon háttérére vonatkozó szabály:

Ha a **HTML** elem **background** értéke nem **transparent**, akkor az kerül használatba, egyébként pedig a **BODY** elem **background** attribútuma.

Fenti szabály használat közben a következőképp jelenhet meg:

```
<HTML STYLE="background: url(http://style.com/hatter.jpg)">  
<BODY STYLE="background: red">
```

A példában a vászont betéríti a 'hatter.jpg' nevű kép. A **BODY** elem háttérszíne pedig (amely nem biztos, hogy teljesen lefedi a vászont), vörös lesz.

CSS tulajdonságok

A stíluslapok a dokumentumok megjelenését a stílustulajdonságokhoz hozzárendelt értékekkel befolyásolják. Ez a fejezet bemutatja a definiált stílustulajdonságokat, és az azokhoz rendelhető értékeket.

A tulajdonság-érték párok jelölési rendszere

A következőkben a tulajdonságokhoz tartozó lehetséges értékeket az alábbi jelrendszer és szintaktika szerint fogjuk bemutatni:

Érték: N | NW | NE

Érték: [<hossz> | vastag | vékony]{1,4}

Érték: [<family-name>,]* <family-name>

Érték: <url>?<szín> [/<szín>]?

Érték: <url> || <szín>

A '<' és '>' jelek között szereplő szavak jelentik az érték *típusát*. A leggyakrabban használt közös típusok közé tartozik a <hossz>, a <százalék>, az <url>, a <szám> és a <szín> (length, percentage, url, number, color). A többi specializált típus (pl.: font-family, vagy border-style) leírása a hozzájuk tartozó tulajdonságnál található.

A kulcsszavak betű szerint szerepelnek, idézőjelek nélkül. A törtjelet (/) és a vesszőt (,) szintén ugyanoda és ugyanúgy kell elhelyezni, ahova és ahogyan a szabályok előírják.

Az egymás mellé írt kifejezések azt jelentik, hogy mindegyikük használható, a mutatott sorrendben. Függőleges vonal (*A|B*) jelzi az alternatívákat: a felsoroltak közül *egy* fordulhat elő. Kettős függőleges vonal (*A||B*) jelzi, hogy a felsoroltak közül vagy A, vagy B, vagy mindkettő előfordulhat, tetszőleges sorrendben. A szögletes zárójelek (*[]*) a csoportosításra utalnak. A tulajdonságok egymás mellé helyezése erősebb, mint a függőleges vonal; a kettős függőleges vonal erősebb, mint a függőleges vonal. Így az "*a b|c||d e*" kifejezés megegyezik az "*[a b]/[c]/[d e]*" kifejezéssel. Minden típust, kulcsszót, vagy zárójeles csoportosítást követhet egy, a következő módosítók közül:

- Csillag (*) jelzi, hogy az előzőekben írt típus, szó, vagy csoport 0 (nulla), vagy annál több esetben ismételhető.
- Plusz jel (+) jelzi, hogy az előzőekben írt típus, szó, vagy csoport 1, vagy több esetben.
- Kérdőjel (?) jelzi, hogy az előzőekben írt típus, szó, vagy csoport használata opcionális.
- Kapcsos zárójelbe írt számpár (*{A,B}*) jelzi, hogy az előzőekben írt típus, szó, vagy csoport legalább (*A*), de legfeljebb (*B*) számú esetben ismételhető.

Font tulajdonságok

A stíluslapok leggyakoribb használata a fonttulajdonságok beállítása. Balszerencsére, nem létezik általános érvényű és széles körűen elfogadott módszer a betűtípusok osztályozására, nincs terminológia, ugyanúgy alkalmazhatnánk a különböző betűtípus-családokra. (az 'italic' jelző általánosan használt a dőlt szöveg jelzésére, de a dőlt betűk jelezhetőek az *Oblique*, *Slanted*, *Incline*, *Cursive*, vagy *Kursiv* címkékkel is.) Ezért nem egyszerű probléma általánosan használható tulajdonságokat hozzárendelni egy meghatározott betűtípushoz. A CSS-ben a *font-family*, *font-style*, *font-variant*, *font-weight*, *font-size* és *font* tulajdonságok vannak leírva.

Betűtípusok megfeleltetése

Mivel nem létezik általános érvényű és széles körűen elfogadott módszer a betűtípusok osztályozására, a tulajdonságok és betűmegjelenítések összepárosítása fokozott óvatosságot kíván. A tulajdonságok megfeleltetése jól definiált sorrendben történik, hogy biztosítva legyen a konzisztencia a megfeleltetési művelet végeredménye és a böngésző képességei között.

1. A böngésző minden általa ismert betűtípushoz, készít (vagy megnyit) egy adatbázist, amely tartalmazza az alkalmazható CSS tulajdonságokat. A böngésző ismeri az elérhető betűtípusokat, mivel azok a helyi számítógépre vannak telepítve. Ha két betűtípus is megfelel ugyanannak a tulajdonságnak, a böngésző az egyiket figyelmen kívül hagyja.
2. Egy adott elemnél, és az elem minden karakterére vonatkozóan a böngésző összegyűjti az elemhez alkalmazható betűtípus-tulajdonságokat. A teljes tulajdonság-készlet használatához a böngésző a *font-family* tulajdonságot használja, hogy kiválasszon egy (ideiglenes) próba-fontcsaládot. A fennmaradó tulajdonságokat pedig a minden tulajdonságra leírt megfeleltetési kritérium szerint teszteli. Ha az összes fennmaradó tulajdonságot sikerült megfeleltetnie, akkor megtalálta a megfelelő font-kinézetet az adott elemhez.
3. Ha nincs a 2. lépésnek megfelelő font-kinézet a *font-family* tulajdonság értékei között, de van alternatív fontcsalád az elérhető betűkészletben, a második lépésben leírt megfeleltetési eljárás azzal folytatódik.
4. Ha van megfelelő kinézetű font, de a készlet nem tartalmazza az ábrázolandó karaktert, és van következő alternatív font-család; a második lépés ismétlődik.
5. Ha nincs megfelelő font a 2. lépésben kiválasztott betűkészlet-családban, a böngészőtől függő alapértelmezett betűkészlet-család kerül használatba, és a böngésző azon belül ismétli meg a 2. lépést, amíg végül megtalálja az előírtak legjobban megfelelőit.

A tulajdonságonkénti megfeleltetés szabályai (2. lépés) a következők:

1. Az első keresési feltétel a *font-style*. (Az '*italic*' keresési feltétel akkor megfelelő, ha vagy van a böngésző font-adatbázisában, amely megfelel a CSS '*italic*', vagy '*oblique*' kulcsszava által meghatározottaknak.)
2. A következő megfeleltetési feltétel a *font-variant*. '*normal*' az a font, amely nincs '*small-caps*'-ként megjelölve. '*small-caps*' az a font, amely (1) így van megjelölve, (2) az a font, amelyből a '*small-caps*'-et előállítják, vagy (3) az a font, amely esetében kisbetűk nagybetűkkel vannak helyettesítve. Kiskapitális betűtípust elő lehet állítani elektronikus úton is, normál fontból, a nagybetűk átméretezésével.
3. A következő megfeleltetési feltétel a '*font-weight*'; ez sosem okozhat hibát (lásd a '*font-weight*' tulajdonságot, lejjebb).
4. A '*font-size*' tulajdonság megfeleltetése a böngésző tűréshatárán történik. (Általában, a méretezhető fontok méretei a legközelebbi egész számú pixel értékére vannak kerekítve; a bittérképes fontok megjelenítésének tűrése 20% körül van.)

'font-family'

Értéke lehet: `[(<family-name>|<generic family>),]*(<family-name>|<generic family>)`

Alapértelmezés: Böngészőfüggő.

Alkalmazható: Minden elemhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

Értéke elsőbbségi listája a fontcsalád neveknek, és/vagy az általános fontcsalád-neveknek. Eltérően a legtöbb más CSS tulajdonságtól, az értékeket vesszővel kell elválasztani, így jelezve, hogy azok alternatívák.

`BODY {font-family: gill, helvetica, sans-serif}`

A felsorolásban használható típusok:

<family-name>

A választott fontcsalád neve. Fenti példában fontcsalád a 'gill' és a 'helvetica'.

<generic-family>

Fenti példában az utolsó érték az általános fontcsalád-név. A következő általános fontcsalád-nevek vannak definiálva:

- serif; (pl: Times)
- sans-serif (pl: Helvetica)
- cursive (pl: Zapf-Chancery)
- fantasy (pl: Western)
- monospace (pl: Courier)

Az általános fontcsaládok, mint végső eset jöhetnek számításba. A szóközöket is tartalmazó fontneveket idézőjelbe kell tenni:

```
BODY {font-family: "new century schoolbook", serif}
```

'font-style'

Értéke lehet:

normal | italic | oblique

Alapértelmezés: normal

Alkalmazható: Minden elemhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

A 'font-style' tulajdonság használatával lehet egy fontcsaládból kiválasztani a normál, vagy dőlt betűs megjelenést (az 'italic' és az 'oblique' is dőlt betűt eredményez).

Az értékek azokat a betűtípusokat választják ki a böngésző fontadatbázisából, amelyek az értéknek megfelelően vannak megjelölve. Ha 'italic' jelölésű font nem elérhető, helyette az 'oblique' kerül használatba. Az 'oblique' jelölésű fontokat a böngésző futásidőben is generálhatja; egy normál (álló) font megdöntésével.

```
H1, H2, H3 {font-style: italic}
```

```
H1 EM {font-style: normal}
```

Fenti példában a H1 elemen belül szereplő kiemelt (EM) szöveg normál betűkkel jelenik meg.

'font-variant'

Értéke lehet:

normal | small-caps

Alapértelmezés: normal

Alkalmazható: Minden elemhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

A fontcsaládon belüli megjelenítésváltozatok másik típusa a kiskapitális betűtípus. A kiskapitális megjelenítésnél a kibetűk hasonlítanak a nagybetűkhöz, csak méretük kisebb; arányaikban jelentéktelen a különbség.

A 'normal' érték a betűtípus normál alakját, a 'small-caps' a kiskapitális alakot választja ki. A CSS-ben elfogadható (de nem megkövetelt), hogy a kiskapitális betűk kisbetűi a normál fontkészlet nagybetűinek futásidejű átméretezésével legyenek kialakítva.

A következő példában található utasítások eredményeképpen a H3 elemekben levő szöveg kiskapitális betűvel lesz megjelenítve:

```
H3 {font-variant: small-caps}
```

'font-weight'

Értéke lehet:

normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900

Alapértelmezés: normal

Alkalmazható: Minden elemhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

A 'font-weight' tulajdonság választja ki a betűtípus súlyát (megjelenését).

Az angol eredetiben a "weight" szó szerepel, a magyarban nincs igazán megfelelő **egy** szó az itt alkalmazott jelentésre: a betűtípus vastagsága, színintenzitása. Ezért megmaradtam a szó szerinti fordításnál: súly.

Alkalmazható értékei 100 - 900 -ig terjedhetnek, ahol minden egyes szám egy súlyt jelez, amelyek mindegyike sötétebb az előzőnél. A `normal` kulcsszó a '400'-as, a `bold` a 700 -as értéknek felel meg.

```
P {font-weight: normal}      /* 400 */
```

```
H1 {font-weight: 700}       /* bold */
```

A `bolder` és `lighter` értékek a szülő elemtől örökölt súlyhoz viszonyítva választják ki az alkalmazandó súlyt.

```
STRONG {font-weight: bolder}
```

A további gyermek-elemek az így eredményként kapott értéket öröklik, nem az eredetit.

A fontoknak (font adatoknak) jellemzően van egy, vagy több tulajdonságuk, amelyeknek értékei a font "súlyát" leíró nevek. Ezeknek a neveknek nincs széleskörűen elfogadott, általános érvényű jelentésük. Feladatuk elsősorban az, hogy súly szerint megkülönböztessék a fontcsaládon belül a különböző kinézetű fontokat. Használatuk különböző fontcsaládok között meglehetősen változó. Az a font, amit **bold**nak (félkövérnek) gondolhatnánk, az leírása szerint lehet, hogy *Regular*, *Roman*, *Book*, *Medium*, *Semi*-, vagy *Demi-bold*, *Bold*, vagy *Black*, attól függően, mennyire sötét a 'normal' font az adott fontcsaládban. Mivel a neveknek nincsenek mértékadó használati előírásaik, a CSS `weight` tulajdonság-értékei egy numerikus skálán vannak megadva, ahol a 400-as, vagy `normal` érték jelenti a fontcsalád "normal" értékét. Ez a súlynév van hozzárendelve ahhoz a kinézetű fonthoz, amelynek neve általában *Book*, *Regular*, *Roman*, illetve ritkábban *Medium*.

A fontcsaládon belül más súlyok hozzárendelése a numerikus értékekhez az árnyékolási (sötétítési) sorrend megtartásával történik. A következő műveleti sorrend megmutatja, hogy egy tipikus esetben hogyan történik a súlyok számértékhez rendelése.

- Ha a fontcsalád már használ kilencértékű numerikus skálát, a hozzárendelés közvetlenül történik.
- Ha a fontcsaládban a `medium` érték és a *Book*, *Regular*, *Roman*, *Medium* értékek más súlyt jelentenek, a `medium` értékhez az 500 lesz hozzárendelve.
- A `bold`ként jelölt font leggyakrabban a 700-as értéket kapja.
- Ha a fontcsaládban kevesebb, mint kilenc súlyérték van, a "lyukak" kitöltésére a következő algoritmus használatos: Ha az '500'-as érték hozzárendelés nélküli, ahhoz a fonthoz lesz hozzárendelve, amelyikhez a '400'. Ha a '600', '700', '800', vagy '900'-as értékek valamelyike marad hozzárendelés nélkül, ahhoz a fonthoz lesznek hozzárendelve, amelyiket a következő sötétebbnek megfelelő kulcsszó kijelöl. Ha a '300', '200', vagy '100'-as értékek valamelyike marad hozzárendelés nélkül, ahhoz a fonthoz lesznek hozzárendelve, amelyiket a következő világosabbnak megfelelő kulcsszó kijelöl.

A következő példa mutatja ennek végrehajtását. A 'Család 1' fontcsaládban négy súlyt feltételezünk, a világosabbtól a sötétebb felé: *Regular*, *Medium*, *Bold*, *Heavy*. A 'Család 2' fontcsaládban 6 súly van: *Book*, *Medium*, *Bold*, *Heavy*, *Black*, *ExtraBlack*. Figyeljük meg, hogy a 'Család 2 ExtraBlack' fonthoz nem rendelünk értéket.

Elérhető kinézetek	Hozzárendelt érték	Kitöltött "lyukak"
Család 1 Regular	400	100, 200, 300
Család 1 Medium	500	
Család 1 Bold	700	600
Család 1 Heavy	800	
Család 2 Book	400	100, 200, 300
Család 2 Medium	500	
Család 2 Bold	700	600
Család 2 Heavy	800	
Család 2 Black	900	
Család 2 ExtraBlack	(nincs)	

Mivel a **bolder** és **lighter** relatív kulcsszavak célja a *családon belül* a fontok sötétebben, vagy világosabban történő ábrázolása, és mert a fontcsaládoknak nincs valamennyi súlyértékhez fontja hozzárendelve, a **bolder** kulcsszó hatására a fontcsalád következő sötétebb, a **lighter** kulcsszó hatására a fontcsalád következő világosabb tagja kerül használatba. A pontosság kedvéért a **bolder** és **lighter** relatív kulcsszavak jelentése a következő:

- A **bolder** azt a súlyt választja ki, amelyik sötétebb, mint az eredeti örökölt érték. Ha nincs megfelelő, ez egyszerűen a következő numerikus értéket jelenti (és a font változatlan marad). Ha az örökölt érték '900' volt, az eredményül kapott súly is '900' lesz.
- A **lighter** hasonlóképpen jár el, csak ellenkező irányban. A következő világosabb súlyértékhez tartozó kulcsszót választja ki; ha nincs megfelelő, a következő világosabb számértéket (és a megjelenített font változatlan marad).

Nincs garantálva, hogy minden **font-weight** érték hatására az alkalmazott font sötétebb lesz; a fontcsaládok egy részében csak normál és félkövér típus van, míg más fontcsaládok nyolc különböző súlyértékkel rendelkeznek. Nincs garantálva az sem, hogy a böngészők helyesen rendelik egymáshoz az ugyanahhoz a fontcsaládhoz tartozó fontokat a megfelelő súlyértékekkel. Az egyetlen garantálható következmény az, hogy egy adott értéknél a font nem lesz kevésbé sötét, mint egy világosabb értéknél.

'font-size'

Értéke lehet: <abszolút méret> | <relatív méret> | <hossz> | <százalék>

Alapértelmezés: medium

Alkalmazható: Minden elemhez

Öröklődik: Igen

Százalékos értékek: Viszonyítási alap a szülő elem fontmérete

<abszolút méret>

Az abszolút méret egy indexet jelent a böngésző által nyilvántartott fontméret-táblázaton.

Lehetséges értékei:

[xx-small | x-small | small | medium | large | x-large | xx-large].

A szomszédos indexek közötti különbség megjelenítéskori javasolt értéke 1,5-szörös; ha a 'medium' 10 pontos méretet jelent, a 'large' 15 pontos lesz. A különböző médiatípusokhoz különböző méretezési faktorok szükségesek. A fontméret-táblázat a különböző fontcsaládok esetén különbözhet egymástól.

<relatív méret>

A relatív méret a fontméret-táblázat és a szülő elem valós fontmérete közötti arányt veszi figyelembe. Lehetséges értékei:

[larger | smaller]

Ha a szülő elem fontmérete 'medium' volt, a larger érték az aktuális elem fontméretének értékét large -ra állítja. Ha a szülő elem fontméretét a böngésző fontméret-táblázata nem tartalmazza, a böngésző helyettesítheti a kívánt méretet a sorban következővel. A hossz és százalékos értékek nem férhetnek hozzá a fontméret-táblázathoz az elem aktuális fontméretének kiszámításakor. Negatív értékek használata nem megengedett. A többi tulajdonságnál, az 'em' és 'ex' méretértékek az aktuális elem fontméretére hivatkoznak. A font-size tulajdonság esetén azonban ezek a mértékegységek a szülő elem fontméretét veszik alapul. Példák:

```
P {font-size: 12pt;}
BLOCKQUOTE {font-size: larger}
EM {font-size: 150%}
EM {font-size: 1,5em}
```

Ha a javasolt 1,5-szörös méretezési faktor van használatban, az utolsó három deklaráció - eredményét tekintve- megegyezik.

'font'

Értéke lehet:

```
[<font-style> || <font-variant> || <font-weight>]<font-size> [/<line-height>]<font-family>
```

Alapértelmezés: Nincs definiálva

Alkalmazható: Minden elemhez

Öröklődik: Igen

Százalékos értékek: Értelmezhető a font-size és a line-height tulajdonságokhoz.

A font tulajdonság használata gyors elérési lehetőséget biztosít afont-style, font-variant, font-weight, font-size, line-height, font-family tulajdonságok beállításához. Használata a hagyományos rövidutas tipográfiai jelrendszer szabályain alapul.

A lehetséges és alapértékek beállításához lásd az előző bekezdésekben leírt szabályokat. Azon tulajdonságok esetében, ahol érték nincs beállítva, azok alapértelmezett értékei kerülnek használatba.

```
P { font: 12pt/14pt sans-serif }
P { font: 80% sans-serif }
P { font: x-large/110% "new century schoolbook", serif }
P { font: bold italic large Palatino, serif }
P { font: normal small-caps 120%/120% fantasy }
```

A második szabályban a százalékos méretérték a szülő elem fontméretét veszi alapul. A harmadik szabály sormagasságra vonatkozó százalékos értéke magát az aktuális elemet veszi viszonyítási alapul. Az első három szabályban a 'font-style', 'font-variant', 'font-weight' tulajdonságok nincsenek explicit módon említve, ezért mindhárom tulajdonság alapértelmezett értékét ('normal') veszi fel. A negyedik szabály a 'font-weight' tulajdonságot 'bold'-ra, a 'font-style' tulajdonságot 'italic'-ra állítja, a 'font-variant' pedig implicit módon kapja meg a 'normal' értéket. Az ötödik szabály állítja be a 'font-variant' (small-caps), a font-size (a szülő elem betűméretének 120%-a), a 'line-height' (a fontméret 120%-a) és a 'font-family' (fantasy) tulajdonságokat.

Szín- és háttértulajdonságok

Ezek a tulajdonságok írják le egy elem (gyakran előtérzínként [foreground color] említett) szín és háttértulajdonságait. A háttér az a felület, amelyre a tartalom kirajzolásra kerül. Ez lehet háttér szín, vagy háttér kép. Szintén ezek a tulajdonságok használhatóak néhány más fontos megjelenítési lehetőség beállításához: háttér-kép helyzete, legyen-e és hányszor ismételve, fix legyen-e, vagy gördíthető a vászonhoz képest, stb...

A `color` tulajdonság normál módon öröklődik. A háttér tulajdonságai nem öröklődnek, de az egyes szülő elemek hátterei áttűnnek az alapértelmezett háttérértéken, mivel a `background-color` alapértelmezett értéke a `transparent`.

'color'

Értéke lehet: szín

Alapértelmezés: Böngészőfüggő.

Alkalmazható: Minden elemhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

Ez a tulajdonság írja le egy elemhez tartozó szöveg színét; gyakran előtérszínként (foreground color) történik róla említés. Egy szín értékének meghatározása többféle módon történhet:

```
EM { color: red }
```

```
EM { color: rgb(255,0,0) }
```

```
EM { color: #FF0000 }
```

Fenti három deklaráció mindegyike az EM elem szövegéhez a vörös színt rendeli hozzá. Az első esetben a szín *nevét* írtuk le, a második és harmadik esetben a szín *RGB kódját*; először *decimálisan*, az utolsó esetben *hexadecimálisan*.

'background-color'

Értéke lehet: <szín> | transparent

Alapértelmezés: transparent

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ez a tulajdonság állítja be egy elem háttérszínét.

```
H1 { background-color: red }
```

```
H1 { background-color: rgb(255,0,0) }
```

```
H1 { background-color: #FF0000 }
```

'background-image'

Értéke lehet: <url> | none

Alapértelmezés: none

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ez a tulajdonság állítja be egy elem háttérként látható képet. Egy háttér-kép beállításakor beállítható háttér-szín is, amely akkor látható, ha a kép nem elérhető. Ha a kép megjeleníthető, a háttérszínt el fogja takarni.

```
BODY { background-image: url(hatter.gif) }
```

```
P { background-image: none }
```

'background-repeat'

Értéke lehet: repeat |

repeat-x | repeat-y |

no-repeat

Alapértelmezés: repeat

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ha van háttér-kép beállítva, a 'background-repeat' tulajdonsággal állítható be, hogy legyen-e és hányszor legyen ismételve a kiválasztott kép.

A repeat érték azt jeleni, hogy a kép vízszintesen és függőlegesen egyaránt ismétlődni fog. A repeat-x (repeat-y)érték felelős a kép vízszintes (függőleges) ismétlődéséért; így egy kisebb méretű háttérképből a képernyő egyik oldalától a másikig érő sáv állítható elő.

```
BODY
{
    background: red url(hatter.gif);
    background-repeat: repeat-y;
}
```

Fenti példában a háttérkép függőlegesen ismétlődik.

'background-attachment'

Értéke lehet: scroll | fixed

Alapértelmezés: scroll

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ha van beállított háttérkép, a 'background-attachment' értéke határozza meg, hogy fixen marad-e, vagy a tartalommal együtt gördíthető.

```
BODY
{
    background: red url(hatter.gif);
    background-repeat: repeat-y;
    background-attachment: fixed;
}
```

A böngészők néha a fixed értéket is scroll-nak értelmezik. A W3C ajánlása azt tartalmazza, hogy a böngészők a fixed értéket értelmezzék helyesen, legalább a HTML és a BODY elemek vonatkozásában, mivel a weblapok szerzőinek nincs arra lehetőségük, hogy külön háttérrel válasszanak csak a gyengébb képességű böngészők számára.

'background-position'

Értéke lehet: [<százalékos> | <hossz>]{1,2}[top | center | bottom][left | center | right]

Alapértelmezés: 0% 0%

Alkalmazható: Blokkszintű- és helyettesített elemekhez

Öröklődik: Nem

Százalékos értékek: Viszonyítási alap a formázott elem

Ha van beállított háttér-kép, a 'background-position' értéke határozza meg annak betöltődéskor elfoglalt helyzetét. A 0% 0% értékpár használatával a kép az elemet körbefogó doboz bal felső sarkába kerül. (Nem a margót, szegélyt, kitöltést tartalmazó dobozról van szó.) A 100% 100% értékpár a kép jobb alsó sarkát azelem jobb alsó sarkába helyezi. A 14% 84% értékpár a képet az elemen vízszintesen a 14%-ot jelentő ponttól függőlegesen a 84%-ot jelentő pontig helyezi el.

A 2cm 2cm értékpár a képet az elem bal felső sarkától 2 cm-re balra és 2 cm-re lefelé helyezi el. Ha csak egy százalékos, vagy hosszúságérték van megadva, az csak a vízszintes pozíciót állítja be, a függőleges pozíció értéke 50% lesz. Két érték megadása esetén a vízszintes pozícióra vonatkozó értéket kell először megadni. A hosszúság- és százalékos értékek kombinációja megengedett (50% 2cm). Megengedett a negatív értékek használata is.

A háttérkép helyzetének beállításához használhatók kulcsszó-értékek is. A kulcsszavak **nem kombinálhatók** a hosszúság- és százalékos értékekkel. A használható kulcsszó-kombinációk és értelmezésük:

Kulcsszavak	Értelmezésük
'top left' és 'left top'	0% 0%
'top', 'top center', 'center top'	50% 50%
'right top', 'top right'	100% 0%
'left', 'left center', 'center left'	0% 50%
'center', 'center center'	50% 50%
'right', 'right center', 'center right'	100% 50%
'bottom left', 'left bottom'	0% 100%
'bottom', 'bottom center', 'center bottom'	50% 100%
'bottom right', 'right bottom'	100% 100%

Példák:

```
BODY {background: url(hatter.gif) right top}
BODY {background: url(hatter.gif) top center}
BODY {background: url(hatter.gif) center}
BODY {background: url(hatter.gif) bottom}
```

Ha a háttér-kép rögzítve van a vászonhoz, az elem elhelyezéséhez a méretek viszonyítása nem az elemhez, hanem a vászonhoz történik.

'background'

Értéke lehet: <background-color> || <background-image> || <background-repeat> || <background-attachment> || <background-position>

Alapértelmezés: Nincs definiálva

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Használható a <background-position> tulajdonsághoz.

A *background* egy ún. gyors tulajdonság, amellyel egy helyről állítható be az összes háttérre vonatkozó tulajdonság. Használható értékei megegyeznek az egyedi háttér-tulajdonságok **összes** értékeivel.

```
BODY {background: red}
P {background: url(hatter.jpg) gray 50% repeat fixed}
```

A *background* tulajdonság mindig beállítja az összes egyedi tulajdonságot. Az első példában csak a háttér-szín lett beállítva, a többi tulajdonság alapértelmezett értékeit kapja. A második példában az összes egyedi tulajdonság kapott kezdőértéket.

Szöveg tulajdonságok

'word-spacing'

Értéke lehet: normal | hossz

Alapértelmezés: normal

Alkalmazható: Minden elemhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

A megadott hosszúságérték jelzi, hogy mennyivel növelendő meg a szavak közötti alapértelmezett szóköz értéke. Értéke lehet negatív, de lehetnek megvalósítás-függő korlátozások. A böngésző szabadon választhatja meg a megfelelő szóközérték-kiszámítási algoritmust. A

szóköz mértéke függhet a szöveg igazításától is (ez a text-align tulajdonság értékeivel állítható be).

```
H1 { word-spacing: 1em }
```

Fenti példában a H1 elemen belül a szóköz értéke 1em értékkel megnövelt. A böngészők a word-spacing bármely értékét értelmezhetik normal-ként.

'letter-spacing'

Értéke lehet: normal | hossz

Alapértelmezés: normal

Alkalmazható: Minden elemhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

A megadott hosszúságérték jelzi, hogy mennyivel növelendő meg a betűk közötti alapértelmezett köz értéke. Értéke lehet negatív, de lehetnek megvalósítás-függő korlátozások. A böngésző szabadon választhatja meg a megfelelő betűközérték-kiszámítási algoritmust. A betűköz mértéke függhet a szöveg igazításától is (ez a text-align tulajdonság értékeivel állítható be).

```
BLOCKQUOTE { letter-spacing: 0,1em }
```

Fenti példában a BLOCKQUOTE elemen belül a betűköz értéke 0,1em értékkel megnövelt. Ha a beállított érték a normal, a böngésző szabadon állíthatja be a betűközt, a szöveg igazításának megfelelően. Ez nem történik meg, ha a letter-spacing tulajdonság értékét explicit módon adjuk meg:

```
BLOCKQUOTE { letter-spacing: 0 }
```

```
BLOCKQUOTE { letter-spacing: 0cm }
```

A böngészők a letter-spacing bármely értékét értelmezhetik normal-ként.

'text-decoration'

Értéke lehet: none | [underline | overline | line-through | blink]

Alapértelmezés: none

Alkalmazható: Minden elemhez

Öröklődik: Nem, de lásd a részletezést alább

Százalékos értékek: Nincs értelmezve

Ez a tulajdonság írja le egy elem szövegének megjelenítésekor alkalmazható dekorációkat (nincs, aláhúzott, fölhúzott, áthúzott, villogó). Ha az elem üres (), vagy nem tartalmaz szöveget (IMG), a tulajdonság nem eredményez látható hatást. A blink érték hatására a szöveg villog.

A szövegdekoráció színeit a color tulajdonság értékeivel kell beállítani.

Ez a tulajdonság nem öröklődik, de az elemek összeillenek szülő elemeikkel. Tehát; ha egy elem aláhúzott, az aláhúzás meg fog jelenni a gyermek elemnél is. Az aláhúzás színe ugyanaz marad akkor is, ha a leszármazott elem color tulajdonságának értéke más.

```
A:link, A:active, A:visited { text-decoration: underline }
```

A fenti példa az összes link szövegét (link minden A elem, amelynek HREF attribútuma van). A böngészőknek fel **kell** ismerniük a blink kulcsszót, de nem kötelesek támogatni a villogtatást.

'vertical-align'

Értéke lehet: baseline | sub | super | top | text-top | middle | bottom | text-bottom | <százalékos>

Alapértelmezés: baseline

Alkalmazható: Soron belüli elemekhez

Öröklődik: Nem

Százalékos értékek: Viszonyítási alap az elem line-height tulajdonsága

Ez a tulajdonság az elem függőleges pozicionálását befolyásolja. Értékkészletének egy csoportja a szülő elemhez viszonylik:

- `baseline` - az elem alapvonalára igazít (vagy az aljára, ha a szülő elemnek nincs alapvonala);
- `middle` - az elem függőleges középpontjára igazít (általában kép esetén);
- `sub` - az elem alsó indexe;
- `super` - az elem felső indexe;
- `text-top` - az elem tetejéhez igazít, a szülő elem betűinek tetejéhez;
- `text-bottom` - az elem aljához igazít, a szülő elem betűinek aljához.

Az értékkészlet másik csoportja a formázott sorhoz viszonylik:

- `top` - az elem tetejéhez igazít, a sor legmagasabb elemével egy vonalba;
- `bottom` - az elem aljához igazít, a sor legmélyebben levő elemével egy vonalba;

A százalékos értékek viszonyítási alapja az elem line-height tulajdonsága. Ez a tulajdonság felel az elem alapvonalának (`baseline`) hollétéért (vagy az elem aljának elhelyezkedéséért, ha az elemnek nincs alapvonala). Negatív értékek megengedettek (a -100% lesüllyeszti az elemet annyira, hogy az elem alapvonala a következő sor alapvonalához ér. Ez az olyan elemek függőleges igazításának is különösen pontos beállítását teszi lehetővé, amelyeknek nincs alapvonaluk [képek]).

'text-transform'

Értéke lehet: `capitalize|uppercase|lowercase|none`

Alapértelmezés: `none`

Alkalmazható: Minden elemhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

- `capitalize` - Minden szó első karakterét nagybetűssé alakítja;
- `uppercase` - Az elem összes betűjét nagybetűssé alakítja;
- `lowercase` - Az elem összes betűjét kisbetűssé alakítja;
- `none` - Hatástalanítja az örökölt értéket.

Az aktuális transzfomáció minden esetben nyelvfüggő.

```
H1 { text-transform: uppercase }
```

A H1 elembe levő szöveget nagybetűssé alakítja.

A böngészők figyelmen kívül hagyhatják a `text-transform` tulajdonságot, ha azok a karakterek, amelyikre alkalmazni kellene, nem a *Latin-1* karakterkészletből valók.

'text-align'

Értéke lehet: `left|right|center|justify`

Alapértelmezés: Böngészőfüggő

Alkalmazható: Blokkszintű elemekhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

Ez a tulajdonság írja le a szöveg igazítását az elemen belül. Az aktuális igazítási algoritmust a böngésző nyelvfüggően alkalmazza.

```
DIV.center { text-align: center }
```

Mivel a `text-align` öröklődik, a DIV elemen belüli összes `CLASS="center"` attribútummal ellátott blokkszintű elem középre lesz igazítva. Megjegyzendő, hogy az igazítás

viszonyítása nem a vászonhoz, hanem az elemhez történik. Ha a `justify` értéket a böngésző nem támogatja, általában a `left` értékkel helyettesíti.

'text-indent'

Értéke lehet: hossz | százalékos

Alapértelmezés: 0 (nulla)

Alkalmazható: Blokkszintű elemekhez

Öröklődik: Igen

Százalékos értékek: Viszonyítási alap a szülő elem szélessége

Ez a tulajdonság határozza meg a formázott sor behúzását. Értéke negatív is lehet, de lehetnek megvalósításbeli korlátozások. Behúzás nem helyezhető el olyan elem belsejébe, ahol valamilyen törés (pl.: BR) már van.

```
P {text-indent: 3em }
```

'line-height'

Értéke lehet: normal | szám | hossz | százalékos

Alapértelmezés: normal

Alkalmazható: Minden elemhez

Öröklődik: Igen

Százalékos értékek: Viszonyítási alap az elem fontmérete

Ez a tulajdonság állítja be a két szomszédos sor alapvonalának távolságát. Ha numerikus érték van meghatározva, a sormagasságot a fontméret és a szám szorzata adja meg. Ez a megoldás a százalékos érték megadástól az öröklődésben különbözik: számérték megadásakor a leszármazott elemek magát a **számot** öröklik, nem az eredményt. Negatív értékek nem alkalmazhatóak. A következő példában szereplő három deklaráció eredménye ugyanaz a sormagasság:

```
DIV { line-height: 1,2; font-size: 10pt }      /* számérték */
DIV { line-height: 1,2em; font-size: 10pt }    /* hossz      */
DIV { line-height: 120%; font-size: 10pt }     /* százalékos */
```

A `normal` érték a `line-height` tulajdonság értékét az alkalmazott fonthoz képest ésszerű méretre állítja be.

Doboz-tulajdonságok

A doboz-tulajdonságok az elemeket reprezentáló dobozok méretét, területét és helyzetét állítják be. A margó-tulajdonságok állítják be az elemhez tartozó margókat. A `margin` tulajdonság minden oldalra vonatkozik, míg a többi margó-tulajdonság csak csak a saját margóikra van hatással. A kitöltés-tulajdonságok írják le, mekkora hely legyen a szegély és az elem tartalma között. A `padding` tulajdonság minden oldalra vonatkozik, míg a többi kitöltés-tulajdonság csak csak a saját oldalon lévő kitöltésre van hatással. A szegély-tulajdonságok állítják be az elemhez tartozó szegélyeket. Minden elemnek négy szegélye van, minden oldalon egy-egy, amelyek szélességükkel, színükkel, stílusukkal vannak leírva. A `width` és `height` tulajdonságok állítják be a doboz méretét; a `float` és `clear` tulajdonságok az elem pozícióját változtathatják.

'margin-top'

Értéke lehet: hossz | százalékos | auto

Alapértelmezés: 0

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Viszonyítási alap a legközelebbi blokkszintű szülőelem

Ez a tulajdonság állítja be az elem felső margóját:

```
H1 { margin-top: 2em }
```

Negatív érték használata engedélyezett, de lehetnek megvalósításbeli korlátozások.

'margin-right'

Értéke lehet: hossz | százalékos | auto

Alapértelmezés: 0

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Viszonyítási alap a legközelebbi blokk szintű szülőelem

Ez a tulajdonság állítja be az elem jobb oldali margóját:

```
H1 { margin-right: 2em }
```

Negatív érték használata engedélyezett, de lehetnek megvalósításbeli korlátozások.

'margin-bottom'

Értéke lehet: hossz | százalékos | auto

Alapértelmezés: 0

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Viszonyítási alap a legközelebbi blokk szintű szülőelem

Ez a tulajdonság állítja be az elem alsó margóját:

```
H1 { margin-bottom: 2em }
```

Negatív érték használata engedélyezett, de lehetnek megvalósításbeli korlátozások.

'margin-left'

Értéke lehet: hossz | százalékos | auto

Alapértelmezés: 0

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Viszonyítási alap a legközelebbi blokk szintű szülőelem

Ez a tulajdonság állítja be az bal oldali margóját:

```
H1 { margin-left: 2em }
```

Negatív érték használata engedélyezett, de lehetnek megvalósításbeli korlátozások.

'margin'

Értéke lehet: [hossz | százalékos | auto]{1,4}

Alapértelmezés: Nincs definiálva

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Viszonyítási alap a legközelebbi blokk szintű szülőelem

A margin tulajdonság egy 'gyorstulajdonság', amelynek segítségével egy deklaráción belül állítható be a margin-top, margin-right, margin-bottom és a margin-left tulajdonság értéke. Ha értéként négy számértéket adunk meg, annak értelmezési sorrendje mindig a felső => jobb alsó => bal. Ha csak egy érték van megadva, az mind a négy oldalra vonatkozik, ha kettő, vagy három: a szemben lévő értékek meg fogják egyezni.

```
BODY { margin: 2em }
```

```
BODY { margin: 1em 2em }
```

```
BODY { margin: 1em 2em 3em }
```

Fenti példák közül a legelső -hatását tekintve- a következővel megegyezik:

```
BODY {  
    margin-top: 1em;  
    margin-right: 2em;
```

```
margin-bottom: 3em;
margin-left: 2 em;    /* megegyezik a szembeni oldal értékével */
}
```

Negatív margó-értékek engedélyezettek, de lehetnek megvalósításbeli korlátozások.

'padding-top'

Értéke lehet: hossz | százalékos

Alapértelmezés: 0

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Viszonyítási alap a legközelebbi blokk szintű szülőelem

Ez a tulajdonság állítja be egy elem felső kitöltését.

```
BLOCKQUOTE { padding-top: 0,3em }
```

Negatív kitöltés-érték nem megengedett.

'padding-right'

Értéke lehet: hossz | százalékos

Alapértelmezés: 0

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Viszonyítási alap a legközelebbi blokk szintű szülőelem.

Ez a tulajdonság állítja be egy elem jobb oldali kitöltését

```
BLOCKQUOTE { padding-right: 0,3em }
```

Negatív kitöltés-érték nem megengedett.

'padding-bottom'

Értéke lehet: hossz | százalékos

Alapértelmezés: 0

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Viszonyítási alap a legközelebbi blokk szintű szülőelem

Ez a tulajdonság állítja be egy elem alsó kitöltését

```
BLOCKQUOTE { padding-bottom: 0,3em }
```

Negatív kitöltés-érték nem megengedett.

'padding-left'

Értéke lehet: hossz | százalékos

Alapértelmezés: 0

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Viszonyítási alap a legközelebbi blokk szintű szülőelem

Ez a tulajdonság állítja be egy elem bal oldali kitöltését

```
BLOCKQUOTE { padding-left: 0,3em }
```

Negatív kitöltés-érték nem megengedett.

'padding'

Értéke lehet: [hossz | százalékos]{1,4}

Alapértelmezés: Nincs definiálva

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Viszonyítási alap a legközelebbi blokk szintű szülőelem

A padding tulajdonság egy 'gyorstulajdonság', amelynek segítségével egy deklaráción belül állítható be a padding-top, padding-right, padding-bottom és a padding-left

tulajdonság értéke. Ha értéként négy számértéket adunk meg, annak értelemezési sorrendje mindig a felső => jobb alsó => bal. Ha csak egy érték van megadva, az mind a négy oldalra vonatkozik, ha kettő, vagy három: a szemben lévő értékek meg fognak egyezni.

A kitöltőfelület megadása a `background` tulajdonsággal:

```
H1 {  
    background: white;  
    padding: 1em 2em;  
}
```

Fenti példa `1em` értékű kitöltést állít be függőlegesen (fent és lent), és `2em` értékű kitöltést állít be vízszintesen (jobb- és baloldalt). Az `em` érték magyarul a *-szorosnak* felel meg. Viszonyítási alapja az elem fontmérete. Az `1em` (egyszeres) megegyezik a használt font magasságával.

A kitöltési értékek nem lehetnek negatívak.

'border-top-width'

Értéke lehet: `thin` | `medium` | `thick` | `hossz`

Alapértelmezés: `medium`

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ez a tulajdonság állítja be egy elem felső szegélyét. A kulcsszavakhoz tartozó tényleges értékek böngészőfüggők, de a következő sorrend mindenképpen érvényben marad: `thin` (vékony) « `medium` (közepes) « `thick` (vastag)

```
H1 { border: solid thick red }
```

```
P { border: solid thick blue }
```

A fenti példa szerint a `H1` és a `P` elemek ugyanolyan szegélyt kapnak, függetlenül a fontmérettől.

Relatív szélesség eléréséhez az `em` használható:

```
H1 { border: solid 0,5em }
```

A szegélyszélességek nem kaphatnak negatív értéket.

'border-right-width'

Értéke lehet: `thin` | `medium` | `thick` | `hossz`

Alapértelmezés: `medium`

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ez a tulajdonság állítja be egy elem jobb oldali szegélyét. Használata megegyezik a `border-top-width` tulajdonság használatával.

'border-bottom-width'

Értéke lehet: `thin` | `medium` | `thick` | `hossz`

Alapértelmezés: `medium`

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ez a tulajdonság állítja be egy elem alsó szegélyét. Használata megegyezik a `border-top-width` tulajdonság használatával.

'border-left-width'

Értéke lehet: `thin` | `medium` | `thick` | `hossz`

Alapértelmezés: `medium`

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ez a tulajdonság állítja be egy elem bal oldali szegélyét. Használata megegyezik a border-top-width tulajdonság használatával.

'border-width'

Értéke lehet: [thin | medium | thick | hossz]{1,4}

Alapértelmezés: Nincs definiálva

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

A border-width tulajdonság egy 'gyorstulajdonság', amelynek segítségével egy deklaráción belül állítható be a border-width-top, border-width-right, border-width-bottom és a border-width-left tulajdonság értéke.

Négy értéke lehet, a következő értelmezésben:

- *Egy megadott érték:* - Mind a négy szegély a megadott értéket kapja;
- *Két megadott érték:* - Az első érték a felső és alsó szegélyekre, a második érték a bal és a jobb szegélyre vonatkozik;
- *Három megadott érték:* - Az első érték a felső, a második a jobb és bal, a harmadik az alsó szegélyre vonatkozik;
- *Négy megadott érték:* - Felső, jobb, alsó és bal oldali szegélyekre vonatkozik, a felsorolás sorrendjében.

A következő példában a megjegyzésben írt értékek jelzik a szegélyek vastagságát:

```
H1 {border-width: thin } /* thin thin thin thin */
H1 {border-width: thin thick } /* thin thick thin thick */
H1 {border-width: thin thick medium } /* thin thick medium thin */
H1 {border-width: thin thick medium thin } /* thin thick medium thin */
```

A szegélyek szélessége nem vehet fel negatív értéket.

'border-color'

Értéke lehet: szín{1,4}

Alapértelmezés: A color tulajdonság értéke

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

A border-color tulajdonság a négy szegély színét állítja be. Négy értéke lehet, az értékek a különböző oldalak színeit állítják be, ahogyan azt a border-width tulajdonságnál leírtuk.

Ha nincs színérték meghatározva, helyét az elem color tulajdonsága veszi át.

```
P {
  color: black;
  background: white;
  border: solid
}
```

Fenti példában az elem szegélye vékony, fekete vonal lesz.

'border-style'

Értéke lehet:

none | dotted | dashed | solid | double | groove | ridge | inset | outset

Alapértelmezés: none

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

A `border-style` tulajdonság állítja be a négy szegély stílusát. Négy értéke lehet, az értékek a különböző oldalak színeit állítják be, ahogyan azt a `border-width` tulajdonságnál leírtuk.

```
#b14 { border-style: solid dotted }
```

Fenti példa szerint a vízszintes szegélyek stílusa `solid`, a függőleges szegélyeké pedig `dotted` (pontvonal) lesz. Mivel a szegélystílusok alapértelmezés szerinti beállítása `none`, külön beállított értékek nélkül az elemeknek nem lesz látható szegélye.

A szegélystílusok a következőket jelentik:

- `none` - Nincs kirajzolt szegély (tekintet nélkül a `border-width` tulajdonság értékére);
- `dotted` - A szegély pontozott vonallal lesz kirajzolva az elem hátterére;
- `dashed` - A szegély szaggatott vonallal lesz kirajzolva az elem hátterére;
- `solid` - A szegély folytonos vonallal lesz kirajzolva az elem hátterére;
- `double` - A szegély kettős vonallal lesz kirajzolva az elem hátterére. A két vonal összes szélessége és a köztük levő köz megegyezik a `border-width` értékével;
- `groove` - A szegély 3D süllyesztett hatást keltve lesz kirajzolva;
- `ridge` - A szegély 3D domború hatást keltve lesz kirajzolva;
- `inset` - A szegély 3D beékel stílust keltve lesz kirajzolva;
- `outset` - A szegély 3D kiemelt hatást keltve lesz kirajzolva;

'border-top'

Értéke lehet: `border-top-width||border-style||szín`

Alapértelmezés: Nincs meghatározva

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ez a 'gyorstulajdonság' egy elem felső szegélye szélességének, stílusának és színének beállítására szolgál.

```
H1 { border-top: thick solid red }
```

Fenti példa a `H1` elem felső szegélynek vastagságát, vonaltípusát és színét állítja be. A felsorolásból kihagyott értékek alapértelmezésükkel lesznek helyettesítve.

```
H1 { border-top: thick solid }
```

Mivel ebben a példában a szín nem szerepel, a szegély színe megegyezik az elem szegélyének színével.

'border-right'

Értéke lehet: `border-top-width||border-style||szín`

Alapértelmezés: Nincs meghatározva

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ez a 'gyorstulajdonság' egy elem jobb oldali szegélye szélességének, stílusának és színének beállítására szolgál. Használata megegyezik a `border-top` tulajdonságéval.

'border-bottom'

Értéke lehet: `border-top-width||border-style||szín`

Alapértelmezés: Nincs meghatározva

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ez a 'gyorstulajdonság' egy elem alsó szegélye szélességének, stílusának és színének beállítására szolgál. Használata megegyezik a border-top tulajdonságéval.

'border-left'

Értéke lehet: border-top-width||border-style||szín

Alapértelmezés: Nincs meghatározva

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ez a 'gyorstulajdonság' egy elem bal oldali szegélye szélességének, stílusának és színének beállítására szolgál. Használata megegyezik a border-top tulajdonságéval.

'border'

Értéke lehet: border-width||border-style||szín

Alapértelmezés: Nincs meghatározva

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

A border tulajdonság egy 'gyorstulajdonság', amelynek segítségével egy deklaráción belül állítható be a border-top, border-right, border-bottom és a border-left tulajdonság értéke.

A következő példában mutatott első deklaráció eredménye megegyezik a második deklaráció eredményével:

```
P { border: solid red }
```

```
P {  
  border-top: solid red  
  border-right: solid red  
  border-bottom: solid red  
  border-left: solid red  
}
```

Eltérően a margin és padding tulajdonságoktól, a border tulajdonság nem tud különböző értékeket beállítani a különböző oldalakhoz. Ehhez a többi szegélytulajdonságot kell használni.

'width'

Értéke lehet: hossz | százalékos | auto

Alapértelmezés: auto

Alkalmazható: Blokkszintű elemekhez és helyettesített elemekhez

Öröklődik: Nem

Százalékos értékek: Viszonyítási alap a szülő elem szélessége

Ez a tulajdonság általában szöveg-elemekhez használatos, de igen hasznos a helyettesített elemeknél (pl.: képek) is. Ezen a módon megadható egy elem szélessége. Az elemek méretezésénél az oldalak aránya változatlan marad, ha a height tulajdonság értéke auto.

```
IMG.icon { width: 100px }
```

Ha a width és a height tulajdonság értéke is auto; az elem saját eredeti méretével jelenik meg. Negatív értékek nem alkalmazhatók. A width tulajdonság és a margó, valamint a kitöltés közötti kapcsolat leírásáért lásd a formázásmodell fejezetet.

'height'

Értéke lehet: hossz | auto

Alapértelmezés: auto

Alkalmazható: Blokkszintű elemekhez és helyettesített elemekhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ez a tulajdonság általában szöveg-elemekhez használatos, de igen hasznos a helyettesített elemeknél (pl.: képek) is. Ezen a módon megadható egy elem szélessége. Az elemek méretezésénél az oldalak aránya változatlan marad, ha a width tulajdonság értéke auto.

```
IMG.icon { height: 100px }
```

Ha a width és a height tulajdonság értéke is auto; az elem saját eredeti méretével jelenik meg. Negatív értékek nem alkalmazhatók. Ha az elem, amelyre alkalmazzuk nem helyettesített elem, a böngészők figyelmen kívül hagyhatják a height tulajdonságot (auto értékkel helyettesítik).

'float'

Értéke lehet: left | right | none

Alapértelmezés: none

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ha none értékét alkalmazzuk, az elem ott jelenik meg, ahol a szövegben az őt beillesztő kódsor található. Ha értéke left (right); az elem a bal- (jobb-) oldalon fog megjelenni, és a szöveg a jobb (bal) oldalán fog körbefutni. Ha értéke left, vagy (right); az elem blokkszintűként viselkedik.

```
IMG.icon {  
    float: left;  
    margin-left: 0;  
}
```

Fenti példa az összes 'icon' osztályba tartozó IMG elemet a szülő elem bal oldalára helyezi el. A float tulajdonságot leggyakrabban soron belüli képekhez használjuk, de jól alkalmazható szöveges elemekhez is.

'clear'

Értéke lehet: none | left | right | both

Alapértelmezés: none

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ez a tulajdonság határozza meg, hogy elem melyik oldalán engedélyezi lebegő elemek megjelenését. Pontosabban: E tulajdonság értékei sorolják fel azokat az oldalakat, ahol lebegő elemek nem jelenhetnek meg. Ha a clear értéke leftre van beállítva, az elem, amelyre alkalmazzuk, a bal oldalán levő bármely lebegő elem alá kerül.

```
H1 { clear: left }
```

Osztályozó tulajdonságok

Ezek a tulajdonságok sorolják az elemeket kategóriákba, ezenkívül beállítják megjelenítési tulajdonságaikat is. A felsorolás-tulajdonságok írják le, a listaelemek formázási előírásait. A felsorolás-tulajdonságok minden elemhez alkalmazhatóak és normál módon öröklődnek a

leszármazási fastruktúrán. Mindamellet megjelénítési hatást csak a listaelemekre gyakorolnak. A HTML-ben tipikusan ide tartoznak a LI elemek.

'display'

Értéke lehet: block | inline | list-item | none

Alapértelmezés: block

Alkalmazható: Minden elemhez

Öröklődik: Nem

Százalékos értékek: Nincs értelmezve

Ez a tulajdonság határozza meg, hoy egy elem legyen-e, és hogyan legyen ábrázolva a megjelenítőn (a megjelenítő lehet képernyő, nyomtatási termék, stb...).

Egy block értékkel rendelkező elem új dobozt kap. A doboz elhelyezését és a szomszédos dobozokhoz viszonyított helyzetét a formázásmodell határozza meg. Általában, a H1, vagy P jellegű elemek block-típusúak. A list-item érték hasonlít a blockhoz, kivéve, ha listajelölő is van hozzáadva. A HTML-ben tipikusan ilyen (listajelölővel ellátott) elem a LI.

Egy inline értékkel rendelkező elem soron belüli dobozt kap, ugyanabban a sorban, mint az őt megelőző tartalom. A doboz méretezése formázott tartalmának mérete szerint történik. Ha tartalma szöveg, sorokra osztható, és minden sornak külön doboza lesz. A margó-, szegély-, és kitöltés-tulajdonságok használatosak az inline elemekhez, a sortörésnél nincs látható hatásuk.

A none érték használata kikapcsolja az elem megjelenítését, beleértve leszármazott elemeit és az őt körülvevő dobozt is.

```
P { display: block }
EM { display: inline }
LI { display: list-item }
IMG { display: none }
```

A legutolsó definíció kikapcsolja a képek megjelenítését.

A display tulajdonság alapértelmezett értéke a block; de minden böngésző rendelkezik alapértelmezett értékekkel minden HTML elemre vonatkozóan, amely a HTML specifikációban írt javasolt megjelenítésen alapul.

A böngészők figyelmen kívül hagyhatják a display tulajdonságot, a stíluslap szerzője által definiált helyett használhatják saját alapértelmezett értékeiket is.

'white-space'

Értéke lehet: normal | pre | nowrap

Alapértelmezés: normal

Alkalmazható: Blokkszintű elemekhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

Az angol 'whitespace' kifejezésre nem találtam egy magyar szót; körülírva: a szavak, mondatok közötti *üres közt* jelenti, ami nem azonos a szóközökkel. A továbbiakban -egyéb megegyezés híján- közként fogom említeni.

Ez a tulajdonság írja le, hogyan legyenek értelmezve a közök egy elemen belül: normal módon (amikor a közök egy szóközzé vannak tömörítve), pre-ként (úgy viselkedjen, mint a PRE a HTML-ben), vagy nowrap módon (a sortörés BR eleméhez hasonló módon):

```
PRE { white-space: pre }
P   { white-space: normal }
```

A white-space tulajdonság alapértelmezése a normal, de a böngészőknek általában minden HTML elemhez van alapértelmezett értékük, a HTML specifikációban leírt elemkirajzolási előírások szerint.

A böngészők figyelmen kívül hagyhatják a display tulajdonságot, a stíluslap szerzője által definiált helyett használhatják saját alapértelmezett értékeiket is.

'list-style-type'

Értéke lehet: disc|circle|square|decimal|lower-roman| upper-roman|lower-alpha|upper-alpha|none

Alapértelmezés: disc

Alkalmazható: display: list-item tulajdonság-érték párral rendelkező elemekhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

E tulajdonság használatával határozható meg a listajelölő megjelenése, ha a list-style-image tulajdonság értéke none, vagy a kép, amelyre a benne szereplő hivatkozás mutat, nem elérhető.

```
OL { list-style-type: decimal } /* 1 2 3 4 stb... */
```

```
OL { list-style-type: lower-alpha } /* a b c d stb... */
```

```
OL { list-style-type: lower-roman } /* i ii iii stb... */
```

'list-style-image'

Értéke lehet: <url>|none

Alapértelmezés: none

Alkalmazható: display: list-item tulajdonság-érték párral rendelkező elemekhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

Ez a tulajdonság állítja be azt a képet, ami listaelem-jelölőként alkalmazható. Ha a kép elérhető, helyettesíteni fogja a list-style-type tulajdonságban beállított jelölőt.

```
UL { list-style-image: url(images/styleimage.gif) }
```

'list-style-position'

Értéke lehet: inside|outside

Alapértelmezés: outside

Alkalmazható: display: list-item tulajdonság-érték párral rendelkező elemekhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

A list-style-position értéke határozza meg a lista-jelölő pozícióját a tartalomhoz képest.

'list-style'

Értéke lehet: [disc|circle|square|decimal|lower-roman|upper-roman|lower-alpha|upper-alpha|none] || [inside|outside] || [url|none]

Alapértelmezés: Nincs definiálva

Alkalmazható: display: list-item tulajdonság-érték párral rendelkező elemekhez

Öröklődik: Igen

Százalékos értékek: Nincs értelmezve

A list-style tulajdonság egy 'gyorstulajdonság', amelynek segítségével a list-style-type, a list-style-image és a list-style-position tulajdonságok értékei állíthatók be.

```
UL { list-style: upper-roman inside }
```

```
UL UL { list-style: circle outside }
LI.square { list-style: square }
```

A LI elemre közvetlenül beállított list-style tulajdonság nem várt eredményt hozhat. Tekintsünk egy példát:

```
<STYLE TYPE="text/css">
  OL.alpha LI {list-style: lower-alpha }
  UL LI      {list-style: disc }
</STYLE>
<BODY>
  <OL CLASS="alpha">
    <LI>1. szint
    <UL>
      <LI>2. szint
    </UL>
  </OL>
</BODY>
```

Mivel a fenti példában az első definíciónak nagyobb az egyedisége, felül fogja bíráltni a második definíció előírásait és csak a lower-alpha felsorolás-stílus fog érvényesülni. Ezért csak a felsorolás jellegű elemeknél ajánlott beállítani a list-style tulajdonságot.

```
OL.alpha { list-style: lower-alpha }
UL       { list-style: disc }
```

Fenti példában az OL és UL elemekre beállított értékek -hála az öröklődési szabályoknak- vonatkozni fognak a LI elemekre is.

Egységek

Hosszúság egységek

A hosszúságértékek meghatározott formátuma egy opcionális +, vagy – jelből (alapértelmezett a +), az azt közvetlenül követő számból és a számot közvetlenül követő egység-azonosítóból (annak kétbetűs rövidítéséből) áll. A 0 (nulla) szám után az egység-azonosító opcionális. Néhány tulajdonság használatánál engedélyezett a negatív érték használata, de ez bonyolíthatja a formázásmodellt és lehetnek megvalósításbeli korlátozásai is. Ha egy negatív érték nincs támogatva, az a legközelebbi támogatott értékkel lesz helyettesítve. A hosszúság-egységeknek két típusa használható: a *relatív* és az *abszolút*. A relatív egységek e méretet egy másik hosszúság-tulajdonsághoz viszonyítva adják meg. Azok a stíluslapok, amelyek a méreteket relatívan adják meg, könnyebben skálázhatók egyik médiatípusról a másikra (pl.: számítógép képernyőről nyomtatóra). A százalékos egységek (lásd alább) hasonló előnyöket kínálnak fel.

A következő relatív egységek használhatók:

```
H1{ margin: 0,5em } /* -szoros, az elem fontméret-magasságához képest */
H1{ margin: 1ex }   /* x-magasság, az 'x' bezű magasságához képest */
P { font-size: 12px }/* pixelben, a vászonhoz képest */
```

Az em és ex értékek az elem fontméretéhez viszonyulnak. E szabály alól az egyetlen kivétel a font-size tulajdonság, ahol az em és ex értékek a szülő elem fontméretéhez viszonyítandók.

A pixel egységek (ahogy a példa mutatja) a vászon (leggyakrabban számítógép-képernyő) felbontásához viszonyulnak. Ha a kimeneti eszköz pixelsűrűsége nagyban különbözik a számítógép képernyőjétől, a megjelenítő eszköz¹ átszámolja a pixelértékeket. A javasolt *referencia pixel* mérete egy pixel látható szöge az olvasó ember karhosszúságában levő 90 dpi pixelsűrűségű eszközön. Egy átlagos karhosszúságot 71 cm-nek (28 inch) véve, egy pixel 0,0227° alatt látszik.

A gyermek elemek nem a relatív, hanem a számított értéket öröklik:

```
BODY {
    font-size: 12pt;
    text-indent: 3em;          /* értsd: 36pt */
}
H1 { font-size: 15pt }
```

Fenti példában a H1 elemek text-indent tulajdonságának értéke 36pt lesz, nem pedig 45pt.

Az abszolút hosszúságegységeket csak akkor érdemes használni, ha a kiviteli eszköz fizikai tulajdonságai ismertek. A következő abszolút egységek támogatottak:

```
H1 { margin: 0,5in }      /* inch; 1 inch = 2.54 cm */
H2 { line-height: 3cm } /* centiméter */
H3 { word-spacing: 4mm } /* milliméter */
H4 { font-size: 12pt }   /* pont; 1 pt = 1/72 inch */
H4 { font-size: 1pc }    /* pica; 1 pc = 12 pt */
```

Abban az esetben, ha a meghatározott méret nem támogatható, a böngészők megpróbálják hozzávetőlegesen megközelíteni.

Százalékos egységek

A százalékos értékek meghatározott formátuma egy opcionális +, vagy – jelből (alapértelmezett a +), az azt közvetlenül követő számból és a számot közvetlenül követő % jelből áll. A százalékos egységek mindig valamely más egységre vonatkoznak, ez leggyakrabban hosszúság-egység. Minden tulajdonságnál, amelyeknél százalékos egységek alkalmazhatók, meg van határozva, hogy a százalékos egység mire hivatkozik. A hivatkozási alap leggyakrabban az adott elem fontmérete.

```
P { line-height: 120% } /* Az elemnél alkalmazott 'font-size' 120%-a */
```

Minden örökölt CSS tulajdonságnál, ha értéke százalékosan van megadva, a leszármazott elemek a számított értéket öröklik, nem a százalékosat.

Színjelölések

A színek meghatározása történhet a szín nevével, vagy numerikusan, a szín RGB kódjával. A javasolt színmegnevezések a következők: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white és yellow. Ez a 16 szín található a Windows VGA palettáján, RGB értékük nincs meghatározva ebben a specifikációban.

```
BODY { color: black; background: white }
H1 { color: maroon }
H2 { color: olive }
```

Az RGB modell számszerű színmeghatározásokat használ. A következő példák ugyanazt a színt eredményezik.

```
EM { color: #f00 }          /* #rgb */
EM { color: #ff0000 }       /* #rrggbb */
EM { color: rgb(255,0,0) }   /* egész számok: 0 - 255 */
EM { color: rgb(100%,0%,0%) } /* százalékos : 0% - 100% */
```

Az RGB értékek hexadecimális formátuma: egy # karakter, amelyet közvetlenül követ vagy három, vagy hat hexadecimális karakter. A háromszámjegyes RGB kifejezések (#rgb) hatszámjegyes formába (#rrggbb) a számjegyek ismétlésével, nem 0-k (nullák) hozzáadásával konvertálhatók. (Tehát a #b0 kifejtve #bb00) Ez biztosítja, hogy a fehér (ffffff) meghatározható legyen rövid formában is (fff), függetlenül a megjelenítő eszköz színmélységétől. Az RGB értékek funkcionális formátuma: az rgb karkterlánc, amelyet közvetlenül követ a három

színérték vesszővel elválasztott felsorolása (vagy három egész szám 0 - 255 értékhatáron belül, vagy három százalékos kifejezés 0% - 100% értékhatáron belül). A megadott értékhatárokon kívüli számértékek nem értelmezhetők, csonkítva lesznek. A következő három deklaráció értelmét tekintve megegyezik.

```
EM { color: rgb(255,0,0) } /* egész számok: 0 - 255 */
EM { color: rgb(300,0,0) } /* csonkítva 255 -re */
EM { color: rgb(110%,0%,0%) } /* csonkítva 100% -ra */
```

URL

Az URL rövidítés a Uniform Resource Locator kifejezést takarja, amelynek magyar megfelelője: Egységes Erőforrás Helymeghatározás.

```
BODY { background: url(images/hatter.jpg) }
```

Az URL kifejezés formája: `url` kulcsszó, amelyet közvetlenül követ zárójelben (()) opcionálisan egyszeres, vagy kétszeres idézőjelek (' , ") közé zárva maga az URL. Relatív URL megadásakor az elérési utat nem a dokumentumhoz kell viszonyítani, hanem a stíluslaphoz.

Összhang

Egy böngésző, amely egy dokumentum megjelenítéséhez CSS-t használ, akkor felel meg a CSS specifikáció által támasztott követelményeknek, ha:

- Elér el minden hivatkozott stíluslapot és specifikációjuk szerint értelmezi őket;
- A deklarációkat a rangsor fejezetben leírt rangsor szerint rendezi;
- A CSS funkcionalitását a megjelenítő eszköz korlátai közé tudja illeszteni.

Egy böngésző akkor felel meg a CSS specifikációban megfogalmazott stíluslap-követelményeknek, ha:

- kimenete érvényes CSS stíluslap.

Egy böngésző, amely egy dokumentum megjelenítéséhez CSS-t használ, és kimenete stíluslap, akkor felel meg a CSS specifikációnak ha a fentiekben említett *mindkét* követelménycsoportot kielégíti.

Egy böngésző sem tudja teljesíteni a CSS valamennyi lehetséges funkcióját: a böngészők akkor felelnek meg a CSS támasztotta követelményeknek, ha az alapvető funkciókat képesek teljesíteni. Az alapvető funkciók a teljes CSS specifikációból állnak, kivéve azokat a részeket, amelyek kifejezetten kivételként szerepelnek. Azokat a funkciókat, amelyek nem tartoznak a CSS alapvető funkciói közé, továbbfejlesztett (advanced) funkcióknak nevezzük.

Példák a megjelenítő eszköz korlátaira: Korlátozott erőforrások (fontok, színek), korlátozott felbontás (a margók nem jelennek meg helyesen). Ezekben az esetekben a böngésző csak megközelíti a stíluslap előírásait. Vannak böngészők, amelyek lehetővé teszik a felhasználó számára a megjelenítés megválasztását.

Előre-kompatibilis elemzés

Ez a leírás a CSS 1-et mutatja be. Előre láthatóan a későbbi verziók több új tulajdonságot vezetnek be. Ez a fejezet azt mutatja be, hogy mit tesznek a böngészők, ha olyan deklarációval találkoznak, amelyek a CSS jelen kiadásában nem érvényesek.

- Az ismeretlen tulajdonságot tartalmazó deklarációkat figyelmen kívül hagyják. Például; ha a stíluslap a

```
H1 { color: red; rotation: 70deg }
```

deklarációt tartalmazza, a böngésző úgy veszi figyelembe, mintha csak a

```
H1 { color: red }
```

deklarációt tartalmazta volna.

- Az érvénytelen értékeket, vagy *érvénytelen részeket tartalmazó értékeket* a böngésző szintén figyelmen kívül hagyja, a következők szerint:

```
IMG { float: left } /* CSS-nek megfelelő */
```

```
IMG { float: left top } /* a 'top' nem értéke a 'float'-nak */
```

```
IMG { background: "red" } /* a kulcsszavak nem kerülhetnek idézőjelbe */
```

```
IMG { border-width: 3 } /* hiányzik a mértékegység */
```

Fenti példában a böngésző csak az első deklarációt értelmezi, így a stíluslap *tényleges tartalma*:

```
IMG { float: left }
```

```
IMG { }
```

```
IMG { }
```

```
IMG { }
```

- Figyelmen kívül hagyja a böngésző az érvénytelen *at* (@) kulcsszavakat is, minden egyébbel, ami követi, egészen a következő pontosvesszőig (;), vagy kapcsos zárójel-párig ({ }). Példaképp tételezzük fel, hogy a stíluslap a következőket tartalmazza:

```
@three-dee
```

```
{
  @background-lightning:
  {
    azimuth: 30deg;
    elevation: 190deg
  }

  H1 { color: red }
}
```

```
H1 { color: blue }
```

Mivel a *@three-dee* a CSS szerint érvénytelen, így az egész deklaráció a harmadik jobb kapcsos zárójelig (}) bezárólag érvénytelen. A böngésző egyszerűen kihagyja, a stíluslap értelmezhető része a következő lesz:

```
H1 { color: blue }
```

Részletesebben:

Egy CSS stíluslap, bármely verziójában készült is, utasítások sorozatát tartalmazza. Az utasításoknak két fajtája van: az *at előírások* és az *előíráskészletek*. Az utasítások körül lehetnek közők (whitespace) is (tabulátor-, szóköz-, ujsor-karakterek).

A CSS utasítások gyakran a HTML dokumentumba vannak beágyazva; megvan a lehetőség arra, hogy ezeket az utasításokat elrejtjük a régebbi böngészők elől. Erre a célra a HTML szabvány 'megjegyzés' (comment) jele szolgál: `<-- Megjegyzés -->` - a két jel közé írandó a CSS utasítássor.

Az *at*-előírások egy azonosítóként szereplő *at*-kulcsszóval kezdődnek, amely előtt közvetlenül egy *at*, vagyis @ karakter áll (pl.:@import, @page). Az azonosító tartalmazhat betűket, számjegyeket és más karaktereket (lásd alább). Egy *at*-előírás tartalma az első pontosvesszőig (;), vagy a következő blokkig tart. Ha egy böngésző olyan *at*-előírással találkozik, amely nem az @import kulcsszóval kezdődik, figyelmen kívül hagyja az egész *at*-előírást és az elemzést a

következő utasítással folytatja. Szintén figyelmen kívül hagyja az @import kulcsszót is, ha az nem a stíluslap legelején található.

```
@import "stilus.css"
H1 { color: blue }
@import "masikstilus.css"
```

Fenti példában a második @import utasítás a CSS1 szerint érvénytelen. A CSS értelmező kihagyja az egész at-előírást, a stíluslapot a következőképpen értelmezi:

```
@import "stilus.css"
H1 { color: blue }
```

Egy blokk nyitó kapcsos zárójellel ({) kezdődik és a neki megfelelő záró kapcsos zárójelig (}) tart. Köztük bármely egyedi karakter előfordulhat, a zárójelek (()), kapcsos zárójelek ({ }) és szögletes zárójelek ([]) kivételével, amelyek a blokkon belül csak párban fordulhatnak elő. A fent említett karakterek közé zárt utasítások egymásba ágyazhatók. Az egyszeres (') és dupla (") idézőjelek szintén párban fordulhatnak elő; a közéjük zárt karakterlánc szöveggé lesz értelmezve. A következőkben bemutatunk egy példát a blokk értelmezésére. Figyeljük meg, hogy az idézőjelek között szereplő záró kapcsos zárójel *nem párja* a nyitó kapcsos zárójelnek, valamint a második egyszeres idézőjel (') egy 'függő' karakter, nem párja a nyitó idézőjelnek:

```
{ causta: "}" + ({7} * '\')
```

Egy előíráskészlet egy szelektorból és a hozzá tartozó deklarációkból áll. A szelektorként értelmezendő karakterlánc az első nyitó kapcsos zárójelig ({) tart (de azt nem foglalja magába). Azt az előíráskészletet, amely nem érvényes CSS szelektorról kezdődik, a böngészők figyelmen kívül hagyják. Tételezzük fel, hogy egy stíluslap a következőképp néz ki:

```
H1 { color: blue }
P[align], UL { color: red; font-size: large }
P EM { font-weight: bold }
```

Fenti példa második sora érvénytelen CSS szelektort tartalmaz, a böngésző a következőképp fogja értelmezni:

```
H1 { color: blue }
P EM { font-weight: bold }
```

Egy deklarációs blokk egy nyitó kapcsos zárójellel ({) kezdődik és a hozzá tartozó záró kapcsos zárójelig (}) tart. Köztük 0 (nulla), vagy több, pontosvesszővel (;) elválasztott deklaráció állhat.

Egy deklaráció egy tulajdonságnévből, kettőspontból (:) és egy tulajdoság-értékből áll. Mindegyik körül lehet köz (whitespace). A tulajdonságnév egy (a korábban leírtaknak megfelelő) azonosító. A tulajdonság-érékben bármely karakter szerepelhet, de a zárójelek (()), kapcsos zárójelek ({ }), szögletes zárójelek ([]), egyszeres (') és dupla (") idézőjelek csak párban fordulhatnak elő. A zárójelek, szögletes zárójelek és kapcsos zárójelek egymásba ágyazhatók. Az idézőjelek között található karakterek szöveggé lesznek értelmezve.

Annak érdekében, hogy a jövőben meglevő tulajdonságokhoz új tulajdonságokat és értékeket lehessen hozzáadni, a böngészőknek figyelmen kívül kell hagyniuk bármely érvénytelen tulajdonságnevet, vagy tulajdonság-értéket. Valamennyi CSS1 tulajdonság csak akkor fogadható el érvényesnek, ha megfelel a nyelvtani és szemantikai előírásoknak. Példaként lássuk a következő stíluslap előírást:

```
H1 { color: red; font-style: 12pt }
P { color: blue; font-vendor: any; font-variant: small-caps }
EM EM { font-style: normal }
```

Az első sor második deklarációjában a '12pt' érvénytelen érték. A második sor második deklarációjában a 'font-vendor' definiálatlan tulajdonság. A CSS értelmező ezeket a deklarációkat kihagyja, így a stíluslapot a következőképp fogja értelmezni:

```
H1 { color: red }
P { color: blue; font-variant: small-caps }
EM EM { font-style: normal }
```

Megjegyzések bárhova beilleszthetők, ahol közök (whitespace) előfordulhatnak. A CSS definiál ezenkívül még helyeket, ahol közök előfordulhatnak és megjegyzések beírhatók. A következő szabályok mindig betartandók:

- Minden CSS stíluslap kis-nagybetű érzéketlen, kivéve a stíluslap azon részeit, amelyeket nem a CSS vezérel (pl.: a fontcsalád nevek és az url-ek kis-, és nagybetű érzékenyek. A CLASS és ID attribútumok a HTML felügyelete alatt állnak.)
- A CSS1-ben a szelektorok (elemnevek, osztályok és ID-k) csak a következő karaktereket tartalmazhatják: a-z, A-Z, 0-9, az Unicode karaktereket 161-255 -ig, valamint a kötőjelet (-); nem kezdődhetnek kötőjellel, vagy számjeggyel; tartalmazhatnak 'függő' karaktereket, és bármely numerikus kóddal jelölt Unicode karaktert (lásd a következő pontot).
- A 'balper' jel (\) után következő legfeljebb négy hexadecimális számjegy (0..9A..F) egy Unicode karaktert jelent.
- Bármely karakter - hexadecimális számot kivéve - megfosztható speciális jelentésétől, ha elé egy 'balper' jelet helyezünk. Példa: "\" - szöveg, amely egy dupla idézőjelet tartalmaz.

JavaScript

Mi is az a JavaScript?

A JavaScript egy objektum alapú programozási nyelv, melyet a Netscape fejlesztett ki eredetileg LiveScript néven. A LiveScript és a Java ötvözéséből alakult ki a JavaScript, melyet először a Netscape Navigator 2.0-ban implementáltak. Sokan nevezik a nyelvet objektum orientáltnak, ami azonban nem teljesen igaz, hiszen olyan alapvető tulajdonságok, mint például az öröklődés, hiányoznak belőle. A JavaScript a Java appletektől és a Java programoktól eltérően futásidőben kerül értelmezésre. Tehát a böngésző letölti az oldalt, megjeleníti a tartalmát, majd értelmezi a JavaScript (vagy más nyelven írt scriptek) sorait. Már most fontos leszögezni, hogy a JavaScript nem Java! Szintaktikájában és felépítésében ugyan nagyon hasonlít a nevét adó programozási nyelvre, azonban lehetőségei sokkal korlátozottabbak.

JavaScript és a böngészők

Mivel a JavaScript interpretált nyelv, a programunk csak az oldal betöltésekor fog lefutni, addig a HTML kód sorai között pihen. Ennek a megoldásnak az az előnye, hogy a hibás programsorokat könnyedén tudjuk javítani, hátránya viszont az, hogy a fáradtságos munkával megírt scriptünkhöz bárki hozzáférhet, aki megtekinti oldalunkat. Ha egy programot a böngésző futtat, annak vannak pozitív és negatív tulajdonságai is. Előnyként értékelhető az, hogy a scriptünk platformfüggetlen lesz, tehát ugyanúgy fog futni egy Macintosh gép böngészőjében, mint a sokat dicsért Windows-os környezetben. Hátrányai közé sorolható viszont az, hogy a script hibátlan futtatása - a szabványok többféle értelmezésének köszönhetően - erősen a használt böngésző típusának függvénye. Jelentős eltérések fedezhetők fel már csak Windows-os böngészők JavaScript implementációinak összehasonlítása során is. Nézzünk csak meg egy Internet Explorer 5.0-ra optimalizált oldalt Netscape, vagy Opera böngészőkkel! Jó esetben csak néhány funkció nem működik az oldalon, máskor azonban az egész weblap működésképtelenné válhat. További

problémák merülhetnek fel, ha oldalunk látogatója olyan böngészőt használ, mely nem támogatja, vagy nem ismeri a JavaScriptet, bár ennek valószínűsége manapság igen kicsi. E böngészők ugyanis hajlamosak arra, hogy a számukra értelmezhetetlen utasításokat egyszerűen megjelenítik, mintha az a weblap szövegének része lenne, elcsúfítva és feltárva ezzel gondosan elkészített oldalunkat. Hogy ezt elkerüljük az utasításainkat a HTML kód megjegyzései közé kell beillesztenünk: a "<!--" és "-->" szekvenciák közé. Az újabb böngészőket nem zavarja ez a kis csalafintaság, a HTML szabványt pontosan követő régebbi változatok pedig nem zavarodnak össze tőle, mert egyszerűen kommentként értelmezik a valójában script-kódot tartalmazó oldalrészeket. A fentiek miatt nagyon fontos, hogy szem előtt tartsuk a következő dolgokat: scripteket csak korlátozott mértékben alkalmazzunk, és lehetőleg úgy, hogy azt több böngésző segítségével is kipróbáljuk. A másik dolog, hogy ha amennyiben lehetőség van rá, alkalmazzunk statikus HTML megoldásokat a feladatok ellátására és szerver-oldali ellenőrzéseket a bemenő adatok validálására arra az esetre, ha a JavaScript kódunk ezt a feladatot a böngésző vezríója és/vagy beállításai miatt nem képes elvégezni. Hogy egy kicsit érthetőbb legyen: ha script segítségével automatikusan átirányítunk egy oldalt egy másikra (később leírjuk hogyan), akkor tegyünk az oldalra egy linket is, amelyik a másik oldalra vezet, azon böngészők számára, akik nem értik a JavaScriptet. A problémák egy része ugyan a böngésző azonosításával (szintén később) elkerülhető, azonban jobb a scriptek használatával csinálni banni.

JavaScript beágyazása a HTML dokumentumba

Miután megismertük a nyelv történetét és sajátosságait, lássuk az első programot, melynek feladata mindössze annyi lesz, hogy egy kis felbukkanó üzenetablakban megjeleníti a "Hello World!" szöveget.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
alert("Hello World!");
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

A példa egyszerű, de ugyanakkor nagyon szemléletes (remélhetőleg). A HTML dokumentumunk fejrészébe illesztettünk be egy programsort a <SCRIPT> tagok közé, így a böngészővel tudattuk, hogy a HTML kódot más nyelven írt scripttel szakítjuk meg. Mivel többféle scriptnyelv is létezik (pl.: VBScript), és ezek közül többet is használhatunk egy dokumentumon belül, a böngészőnek megmondhatjuk, hogy mely scripteket hogyan kell értelmeznie. Ehhez a SCRIPT tag LANGUAGE attribútumát kell beállítanunk. Ha például egyszerűen a JavaScript szót írjuk ide be, az a böngésző számára a JavaScript 1.0-s verzióját jelenti. Ha olyan szolgáltatásokat is használni szeretnénk, melyek ebben a verzióban még nem szerepelnek, az attribútumnak adjuk értékül a JavaScript1.2-t, és máris új funkciókkal bővülnek scriptünk lehetőségei. Térjünk vissza egy pillanatra a programhoz, mely tulajdonképpen egyetlen sorból áll, mely elvégzi a kitűzött feladatot, azaz megjeleníti üzenetünket. Ehhez az ALERT() metódust (ha jobban tetszik függvényt, de legyünk objektum alapúak), használtuk. Az alert eljárás használata rendkívül egyszerű a zárójelek között kell megadnunk a megjelenítendő szöveget. Ha szövegünk statikus, azaz nem változik, idézőjelek között kell beillesztenünk a függvénybe. Az alert segítségével a változók értékeit is megjeleníthetjük, de erről majd később lesz szó.

Legfontosabb események

Egy HTML oldal életében is vannak események, például amikor valaki végre betölti lapunkat, amikor kattint rajta, vagy amikor elhagyja azt. Ezeket az eseményeket, vagy ha jobban tetszik felhasználói interakciókat scriptünk képes lekezelní beépített eseménykezelői segítségével. A legfontosabb eseményeket foglaljuk össze az alábbiakban.

Esemény neve Eseménykezelő Mikor következik be

load	onLoad	a HTML oldal betöltésekor
click	onClick	az objektumra történő kattintáskor
change	onChange	ha az űrlap mezőjének értéke megváltozik
mouseover	onMouseOver	az egérmutató az objektum fölött van

A felsorolás természetesen nem teljes, azonban néhány alapvető ismeretet szerezhetünk belőle. Az első, amit fontos tudni, hogy értelemszerűen nem minden objektumhoz rendelhetünk hozzá minden eseményt (a későbbiekben egy nagyobb táblázat segít majd ebben eligazodni). A második fontos tudnivaló az eseménykezelők helyes leírásának módja, a JavaScript ugyanis - akárcsak az igazi Java - különbséget tesz a kis és nagybetűk között. Ennek megfelelően az eseménykezelők neveit mindig kis kezdőbetűvel, a szóösszetételeknél azonban nagybetűvel kell írni, így lesz az onMouseover-ből onMouseOver.

Használjuk amit tudunk, 2. példa

A következő példában tudjuk is alkalmazni a most megszerzett ismereteinket. Mindössze annyi a dolgunk, hogy egy gombra való kattintáskor írjuk ki a "Rámkattintottál!" szöveget. Ehhez létre kell hoznunk egy HTML formot, amibe beágyazhatjuk a gombot. A gombra való kattintáskor tehát bekövetkezik egy click esemény, mely meghívja az onClick eseménykezelőt, melynek hatására lefut a scriptünk, és kiírja a szöveget. A forráskód tehát a következő:

```
<HTML>
<BODY>
<FORM>
<INPUT TYPE="BUTTON" NAME="GOMB" VALUE="Kattints ide!"
onClick="alert('Rámkattintottál!') ">
</FORM>
</BODY>
</HTML>
```

Scriptünk írása közben használhatunk szimpla és dupla idézőjeleket is, és kombinálhatjuk is őket, de mindig ügyeljünk a sorrendre. Ha egy szöveget szimpla idézőjellel kezdtünk, akkor azzal is zárjuk be, egyértelműen kifejezve ezzel a szöveg és az utasítás határait.

Link a Script-re

Ennek a fejezetnek két fontos tanulsága lesz. Az első, hogy a JavaScriptünket nemcsak események hatására hívhatjuk meg, hanem szabványos HTML linkek segítségével is, a második pedig az, hogy ahány böngésző, annyi szokás. Mit is jelent ez? Azt már tudjuk, hogy a Netscape-ben jelent meg először a JavaScript, és mint látványos és jól használható scriptnyelv hamarosan el is terjedt, és több böngészőbe is beépítették. A böngészők fejlesztői azonban nem egységesen implementálták a nyelvet, és ez érdekes megoldásokhoz vezetett. IE 4.0 alatt tesztelve oldalunkat semmi baj nem származik abból, ha egy képhez rendeljük hozzá az onClick eseményt - ami a JavaScript szabályai szerint súlyos hibának számít - sőt, oldalunk még megfelelően is fog működni. Ha azonban ezt az oldalt a Netscape valamely kiadásával tekintjük meg jobb esetben nem történik semmi, ha a képre kattintunk, rosszabb esetben hibaüzenetet is kapunk. Akkor vajon mi a teendő ha egy képhez az onClick eseménykezelőt szeretnénk rendelni, és szeretnénk

oldalunkat lehetőleg Netscape és Opera böngészőkkel is élvezhetővé tenni? Ekkor használhatjuk a következő megoldást: képhez ugyan nem, de linkhez már rendelhetünk Click eseményt. Tehát a képet, mint szabványos linket hozzuk létre viszont href attribútumának a következő értéket adjuk: href="javascript:void(utasítások, függvények)". A következőképpen:

```
<HTML>
<BODY>
<a href="javascript:void(alert('Működik!'))">link, ami akár egy kép is
ehet</a>
</BODY>
</HTML>
```

Tehát létrehoztunk egy linket - ami nem csak szöveg, hanem akár egy kép is lehet - aminek href tagjában megadtuk, hogy a link valójában nem egy másik HTML oldalra mutat, hanem egy JavaScript utasításra, ami esetünkben az alert függvény meghívása. A függvény értéket adhat vissza, és a visszaadott érték a képernyőn jelenne meg, amit viszont a VOID() megakadályoz. Ezt a megoldást csak akkor kell alkalmaznunk, ha a meghívott eljárásnak van visszaadott értéke, tehát esetünkben nem.

Csoportosítsunk, avagy a függvények

A függvényeket teljesen szabadon definiálhatjuk a következő szintaktikával:

```
function fuggveny(valtozo) {
utasitasok
...
}
```

Az ellenőrzést és az adatok elküldését elvégezhetjük egy függvény segítségével, így az eseménykezelő meghívása után csak a függvényünk nevét kell írunk. Ennek a függvénynek átadjuk az űrlap megfelelő mezőjének értékét és kezdődhet is a vizsgálat. Az ilyen függvényeket a HTML dokumentum elején szokás definiálni, elkerülve azt, hogy előbb hívjuk meg a függvényt, mint ahogy a leírása szerepelne. A legcélszerűbb ha függvényeinket még a fejrészben deklaráljuk, így ezek a problémák nem fordulhatnak elő. A függvény írása során egész csomó változóval dolgozhatunk, így fontos hogy ismerjük azok hatáskörét. A változók érvényességi köre attól függ, hogy hol deklaráltuk őket. Ha a függvényeken kívül, akkor az egész dokumentumra érvényes, ha függvényen belül, akkor csak az adott függvényen belül érhetjük el őket. A függvények használatát mutatja be a következő egyszerű példa, mely egy figyelmeztető ablakban írja ki, hogy az OK és a Mégse lehetőségek közül melyiket választottuk.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function dontes() {
if (confirm('Mit választasz?'))
alert('OK');
else
alert('Mégse');
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<input type="button" value="Válassz!" onClick="dontes()">
</FORM>
</BODY>
</HTML>
```


A fenti script értelmezéséhez néhány alapvető ismeret azért szükséges. A CONFIRM() metódus egy választó-ablakot jelenít meg a képernyőn, ahol az OK és a Mégsem lehetőségek közül választhatunk. Ha az OK-t választjuk az eljárás igaz értékkel tér vissza, ha a Mégse gombra kattintunk hamis a visszatérési érték. Ennek megfelelően a függvényben lévő feltétel teljesül, illetve nem teljesül, tehát a választásunknak megfelelő figyelmeztető ablak jelenik meg a képernyőn.

Objektumok, dokumentumok

Tudjuk, hogy a JavaScript objektum alapú, és nem objektum orientált nyelv, hiszen az objektumok egy szinten helyezkednek el, objektumai között mégis szoros, alá és fölérendeltségi viszony figyelhető meg. Az egész HTML dokumentumunk ilyen objektumokból épül fel, melyek alapkontextusát a window objektum adja. Az objektumoknak további objektumai, tulajdonságai és metódusai is lehetnek, amelyekre a pont-operátor segítségével hivatkozhatunk. Lássunk egy egyszerű példát: ha a HTML oldalunk URL-jét akarjuk megtudni, tehát azt az Internet-címet, amin keresztül a lapot elérjük, a document objektum location objektumának értékét kell kiíratnunk: document.location. Az objektumok metódusait is hasonló módon érhetjük el: a dokumentum név mezőjére való fókuszáláshoz például a mezőhöz tartozó FOCUS() metódust kell meghívunk: document.form.név.focus(). Az objektumokat nagyon sokféle módon elérhetjük programozás során, ami megkönnyíti a munkát, viszont nagyon zavaró is lehet, ha valaki nincs tisztában az objektumok hierarchiájával, ezért jól jöhet a következő segítség. A teljes HTML dokumentum objektumok sokaságából épül fel, és a JavaScript segítségével szinte minden egyes objektumnak meg tudjuk változtatni az összes tulajdonságát - hiszen erre találták ki. Építsünk fel egy HTML űrlapot, mely telis-tele van különböző beviteli mezőkkel rádiógombokkal, és checkbox-okkal. Ennek a dokumentumnak a fő objektuma a document objektum. Ennek további objektumai a formok, melyek további objektumokként tartalmazzák a beviteli mezőket. A beviteli mezőknek pedig további metódusai és tulajdonságai lehetnek. Így tehát egy beviteli mező értékét a következőképpen érhetjük el: document.form.mezo.value. Az objektumoknak nem kötelező, de lehet nevet adni. Az objektumok ekkor háromféle módon érhetők el. Az előbbi példánál maradva: a beviteli mező értéke a következőképpen érhető el:

- document.form.mezo.value
- document.forms[0].elements[0].value
- document.forms['form'].elements['mezo'].value

A fenti elérési módozatokat kombinálhatjuk is. Sokat segíthet azonban, ha tudjuk, hogy a dokumentum elemei a lap tetejétől kezdve kerülnek beszámózásra, tehát a document.forms[0] a dokumentum tetejéhez legközelebb eső HTML formot jelenti. Nézzünk egy példát: Az oldal tulajdonságainak módosítása lesz a feladat. Létrehozunk egy formot egy beviteli mezővel, amelynek kiírjuk, majd megváltoztatjuk az értékét. Ehhez két függvényt fogunk használni, egyet a kiíráshoz és egyet a megváltoztatáshoz.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function kiiras(mezo) {
alert(mezo);
}
function valtoztatas() {
document.form.szoveg.value="Megváltozott!";
}
</SCRIPT>
</HEAD>
```

```

<BODY>
<FORM name="form">
<input type="text" name="szoveg">
<input type="button" value="Kiir"
onClick="kiiras(document.form.szoveg.value)">
<input type="button" value="Valtoztat" onClick="valtoztatas()">
</FORM>
</BODY>
</HTML>

```

Ahhoz hogy a fent leírt feladatot el tudjuk végezni a KIIRAS() függvény meg kell hogy kapja a beviteli mező értékét meghívásakor, mely jelen esetben a document.form.szoveg.value tulajdonságon keresztül érhető el. A VALTOZTATAS() eljárásnak ezzel szemben nincs szüksége semmilyen paraméterre, hiszen feladata csak annyi, hogy a mező értékét megváltoztassa. A fenti példa ugyan nagyon egyszerű, de vegyük észre, hogy ezzel a módszerrel, tehát egy egyszerű értékadással, módosíthatjuk minden objektum tulajdonságát, dinamikusan változtatva ezzel oldalunk megjelenését. Kicserélhetjük a háttérszínét, de akár a betűk színét is megváltoztathatjuk. Ha a beviteli mezőnek nem adunk nevet, akkor a változtatáskor a fentiekben leírt módon járhatunk el. Ha tehát az VALTOZTATAS() függvényt kicseréljük a következőre, minden ugyanúgy fog működni.

```

function valtoztatas() {
document.forms[0].elements[0].value="Megváltozott!";
}

```

Ha hibáztunk

Mi történik, ha hibát vétünk a program írása közben? A többi programozási nyelvben ilyenkor a fordítótól figyelmeztetést vagy hibaüzenetet kapunk. JavaScript esetében a fordítást és a futtatást is egy böngésző végzi, így a hibajelzéseket innen kell várnunk, néhány esetben sajnos hiába. Azért hiába, mert a böngészők "természetesen" ebben is különböznek. Ha a Netscape hibát jelez ("Javascript error!" Felirat jelenik meg az állapot sorban), akkor a címsorba kell beírunk a "javascript:" sort, és máris rámutat a hiba okára. Ez a funkció akkor is jól használható, ha csak ellenőrizni szeretnénk egy-egy utasítást, mivel azokat itt is tesztelhetjük. Írjuk be a címsorba, hogy "javascript:alert('Hiba!')" és máris megjelenik a figyelmeztető ablak. Ez a funkció rendkívül egyszerűen és hatékonyan használható.

Az Explorer szolidabban (egy kis sárga háromszög a status sor sarkában) jelzi a hibánkat, és a hiba okának felderítésében sem jár el a Netscape gondosságával.

Események

Ahhoz, hogy hatékonyan használhassuk a JavaScript nyelvet, szükségünk lesz egész csomó esemény kezelésére, hiszen a felhasználói beavatkozások, tevékenységek irányítják lapunk működését. Az események lekezeléséhez - mint már tudjuk - eseménykezelőket kell használnunk. Ezek egy kis részét már ismerjük, de a programozás során sokkal több esemény megkülönböztetésére lesz szükségünk. Amit fontos még tudni, hogy nem minden esemény rendelhető hozzá minden objektumhoz, elemhez. Hogy tisztán lássuk az események - eseménykezelők - objektumok viszonyát, nézzük meg alaposan az alábbi táblázatot. események - eseménykezelők – objektumok

Esemény	eseménykezelő	objektum	Akkor következik be...
Blur	onBlur	window, frame, form	elemei ha a fókusz egy másik objektumra kerül
Click	onClick	gomb, radio-gomb, checkbox, link	ha az objektumra kattintunk
Change	onChange	text, textarea, select	ha megváltozik az adott mező értéke
Focus	onFocus	window, frame, form	elemei ha a fókusz az elemre kerül
Keydown	onKeyDown	document, image, link, text, textarea	ha egy billentyű lenyomásra kerül
Keyup	onKeyUp	document, image, link, text, textarea	ha egy billentyű felengedésre kerül
Load	onLoad	document	amikor a HTML oldal betöltődik
Mouseout	onMouseOut	area, link	ha az egér elhagyja az objektumot
Mouseover	onMouseOver	area, link	ha az egér az elem fölé kerül
Select	onSelect	text, textarea	ha a mező valamely részét kiválasztjuk
Submit	onSubmit	form	ha a submit gombra kattintunk

Ha esetleg a fenti táblázat értelmezéséhez íme egy kis magyarázat: az első oszlopban szerepel az esemény neve, a másodikban a hozzá tartozó eseménykezelő, a harmadik oszlop tartalmazza azon objektumok listáját, melyekhez az eseménykezelő hozzárendelhető, és végül a negyedik oszlopban láthatjuk, hogy az adott esemény mikor következik be. Nézzünk egy gyakorlati példát:

Változó képek

Ha a HTML kódban tag-be berakjuk a NAME="kep" attribútumot, akkor képünkre a következő módon hivatkozhatunk: document.kep, a forrásfájlt pedig a document.kep.src tulajdonságon keresztül érhetjük el, illetve változtathatjuk meg. A feladat rendkívül egyszerű lesz: változzon meg a kép, ha a felhasználó fölé viszi az egeret, nyerje vissza eredeti állapotát, ha elvitte onnan, és legyen más akkor is, ha rákattintottunk. Lássuk a programot:

```
<html>
<body>
<a href="javascript:alert('Lenyomva is más a kép, de sajnos hamar
isszaugrik')">

</a>
</body>
</html>
```

Mi történik, ha a cserélendő képek túl nagyok, vagy a felhasználó sáv szélessége túl kicsi? Nagyon egyszerű, a képek kisebb nagyobb késéssel fognak megjelenni, hiszen letöltésük csak az első hivatkozás pillanatában fog elkezdődni (azaz pl. a 2.jpg akkor, amikor először mozgatjuk az egeret a kép fölé), és a kicsi sáv szélesség miatt akár jó néhány másodpercre is beletelhet mire a böngészőnek sikerül a képet letöltenie, és végre megjelenítheti azt. Ez időnként elég zavaró lehet. Azonban erre is van megoldás.

Előöltött képek

Létre kell hoznunk új objektumokat, mégpedig olyanokat, amelyek képek, de nem jelennek meg az oldal betöltődéskor a dokumentum felületén. Ezt az Image objektum használatával érhetjük el. Lássuk először a forráskódot, majd sorról sorra a magyarázatot.

```
<html>
<script>
    elseKep = new Image();
    elseKep.src = "1.jpg";
    masodikKep = new Image();
    masodikKep.src = "2.jpg";
    function kepMutat(forras) {
        if (forras==1) document.kep.src= elseKep.src;
        else document.kep.src= masodikKep.src;
    }
</script>
<body>
<a href="javascript:alert('Előre töltött képekkel...')">

</body>
</html>
```

A HTML kód remélem már mindenkinek világos: létrehozunk egy képet - ami feltétlenül link kell, hogy legyen -, amely különböző események hatására különböző függvényeket hív meg. A hangsúly itt a függvényeken, illetve a JavaScript kódon van. A script törzsében - tehát a függvény definícióján kívül - létrehozunk két új objektumot, melyek így a függvényből és azon kívülről is elérhetőek lesznek. Ezek az objektumok Image, azaz kép objektumok, melyeket a következőképpen hoztunk létre: elseKep = new Image(). Tehát az elseKep egy új (new) kép objektum lesz (Image), melynek, a következő sorban, az src attribútumában megadjuk a képet tartalmazó fájlunk nevét. Hasonlóan hozzuk létre a második képünket is. Ezzel a módszerrel a képek már az oldal letöltődése közben letöltésre kerülnek a memóriába vagy a böngésző helyi gyorsítótárába, így a képcserékor csak meg kell jeleníteni őket, nem kell azok letöltésére várnunk. Nincs más dolgunk, mint írni egy függvényt, ami a képek attribútumainak változtatásait elvégzi. Erre a feladatra szolgál a kepMutat eljárás, ami a paraméterétől függően az elseKep vagy a masodikKep objektumokban tárolt képeket jeleníti meg, egy feltételtől függően. A fenti módszerek alkalmazásával - akár többtucat képet is használhatunk - nagyon látványossá tehetünk már egy egyszerű oldalt is, azonban itt sem érdemes túlzásokba esni, és túldíszíteni a lapunkat.

Időzítések

A JavaScript lehetőséget ad az utasítások időzítésére. Ez az időzítés tulajdonképpen azt jelenti, hogy az adott utasítás csak bizonyos idő múlva fog végrehajtódni. Erre a funkcióra sokszor szükségünk lehet, ha valamit késleltetni szeretnénk, például a felhasználónak időt szeretnénk adni egy szöveg elolvasására, mielőtt új oldalt töltenénk be. Az ilyen esetekre találták ki a setTimeout() metódust, melynek paraméterei a következők: setTimeout("utasítás",idő). Az eljárás a paraméterül kapott utasítást - melyet mindig idézőjelek közé kell tennünk - csak adott idő múlva hajtja végre. Ha tehát azt szeretnénk, hogy 5 másodperc múlva ugorjunk egy másik oldalra, akkor a következő utasítást kell használnunk: setTimeout("location.href='uj_oldal.html'",5000), ugyanis az idő paraméterül szolgáló számokat milliszekundumokban (azaz ezred másodpercekben) kell megadnunk.

Feladatunk az lesz, hogy egy beviteli mezőben bizonyos késleltetéssel írjunk ki egy szöveget, mintha valaki éppen most gépelné be azt. Lássuk először a forráskódot majd a hozzá tartozó magyarázatot.

```
<html>
<script>
function kiiras() {
    var kiirandoSzoveg="Ezt szépen lassan jelenítjük meg...";
    var kiirtSzoveg=document.form.mezo.value;
    var kiirandoHossz=kiirandoSzoveg.length;
    var kiirtHossz=kiirtSzoveg.length;
    if (kiirtHossz<kiirandoHossz) {
        document.form.mezo.value=kiirandoSzoveg.substring(0,kiirtHossz+1);
        setTimeout("kiiras()",300);
    }
}
</script>
<body onLoad="kiiras()">
<form name='form'>
<input type='text' name='mezo' size='35'>
</form>
</body>
</html>
```

A HTML kódban mindössze egy beviteli mezőt definiálunk a hozzá tartozó form-mal, és az oldal betöltődésekor meghívjuk a kiiras eljárást. Nézzük akkor a függvényünket sorról sorra. Az első sorban definiálunk egy változót, melyben elhelyezzük a kiírandó szöveget. A második sorban létrehozunk még egy változót, melyben a már kiírt szöveget tároljuk el. Mivel szükségünk lesz mind a kiírandó, mind a kiírt szöveg hosszára, ezért a következő utasítások segítségével ezt a két értéket tároljuk el egy-egy változóban. Következik egy feltétel, mely eldönti, hogy folytassuk-e a kiíratást, vagy befejeződhet scriptünk futása, azaz már kiírtuk a kívánt szöveget. Ekkor a kiírt karakterek számát növeljük a substring metódus segítségével, majd újra meghívjuk függvényünket, egy kis késleltetés beiktatásával.

Azt hiszem, nem árt, ha a feltétel igaz ágában szereplő két sort kicsi tovább elemezzük. A kiirandoSzoveg változónak használjuk a substring metódusát, melynek két paramétere van. Az első paraméter mondja meg, hogy hányadik karaktertől kezdve, a második pedig hogy hányadik karakterig írjuk ki a stringet. Tehát ha stringünk a"valami" szóból áll, akkor annak string.substring(0,2) metódusa a "va" szótagot adja vissza értékül. Mivel esetünkben a második paraméter mindig egyel növekszik, így minden lépésben egyel több karakter jelenik meg a kiírandó stringből a képernyőn. Most következik a késleltetés, mely 0,3 másodperc múlva újra meghívja a kiiras() eljárást, és így újabb betű jelenhet meg a beviteli mezőben.

Ezt a módszert, amikor a függvény saját magát hívja meg rekurciónak nevezzük, és ilyen, vagy ehhez hasonló problémák (faktoriális kiszámítása) nagyon jó hasznát vehetjük.

Status sor

A status sorban állandó és változó információk is megjelenhetnek, éppúgy, mint egy beviteli mezőben. A két mezőbe való írás módja sem tér el jelentősen egymástól, így kézenfekvő, hogy az előbbi példában használt szövegkiíró scriptet ültessük át a status sorra. A kód csak annyiban változik, hogy nem kell formot deklarálnunk, és a beviteli mező helyett a scriptünk kimenete a status sor lesz.

```
<html>
<script>
```

```

function kiiras() {
    var kiirandoSzoveg="Így készül az internetes futófény, már csak le
kellene törölni...";
    var kiirtSzoveg=window.status;
    var kiirandoHossz=kiirandoSzoveg.length;
    var kiirtHossz=kiirtSzoveg.length;
    if (kiirtHossz<kiirandoHossz) {
        window.status=kiirandoSzoveg.substring(0,kiirtHossz+1);
        setTimeout("kiiras()",100);
    }
}
</script>
<body onLoad="kiiras()">
Ne ide nézz, hanem egy kicsit lejjebb...
</body>
</html>

```

A kódban történt változtatások szerintem maguktól értetődőek, ezért nem is szeretném túlmagyarázni a dolgot. A scriptünkkel - remélhetőleg - csak egyetlen gond van, mégpedig az, hogy ugyan hajlandó kiírni a szöveget, azonban futása ekkor be is fejeződik, és a status sor újra unalmassá válik. Erre a problémára nyújt gyógyírt a következő példa.

SWITCH használat

Nézzünk egy olyan programot, amely eltávolítja egy szövegből az ékezeteket:

```

<html>
<script>
function check() {
var
utolsoKarakter=document.form.text.value.substring(document.form.text.value.length-1);
var ujKarakter="";
switch (utolsoKarakter) {
case "é" : ujKarakter="e"; break;
case "É" : ujKarakter="E"; break;
case "á" : ujKarakter="a"; break;
case "Á" : ujKarakter="A"; break;
case "í" : ujKarakter="i"; break;
case "Í" : ujKarakter="I"; break;
case "ű" : ujKarakter="u"; break;
case "Ű" : ujKarakter="U"; break;
case "ú" : ujKarakter="u"; break;
case "Ú" : ujKarakter="U"; break;
case "ő" : ujKarakter="o"; break;
case "Ő" : ujKarakter="O"; break;
case "ó" : ujKarakter="o"; break;
case "Ó" : ujKarakter="O"; break;
case "ü" : ujKarakter="u"; break;
case "Ü" : ujKarakter="U"; break;
case "ö" : ujKarakter="o"; break;
case "Ö" : ujKarakter="O"; break;
}
if (ujKarakter)
document.form.text.value=document.form.text.value.substring(0,document.form.text.value.length-1) + ujKarakter;
}
</script>
<body>

```

```

Ide lehet bepötyögni a szöveget, amiből majd kiszedjük az ékezetes
betűket:<br>
<form name="form">
<textarea name="text" rows="10" cols="30" onKeyUp="check()"></textarea>
</form>
</body>
</html>

```

Létrehoztunk egy form-ot egy beviteli mezővel, és megadtuk, hogy minden egyes billentyű lenyomása után fusson le az általunk megírt ellenőrző függvény. Metódusunk nem túl bonyolult mindössze egy változó definíciót, egy switch-case szerkezetet, és egy feltételt tartalmaz. Az első lépésben beolvassuk a beviteli mezőben lévő sztringet, és levágjuk róla az utolsó karaktert, hiszen elég ezt vizsgálnunk. Ezek után el kell döntenünk, hogy mit tegyünk ezzel a karakterrel. Ehhez egy switch-case szerkezetet használhatunk, mely a következőképpen működik. A switch a paraméterként megkapott változó, switch(valtozo), értékétől függően különböző utasításokat hajt végre. Ezeket az értékeket és a hozzájuk tartozó utasításokat a case "érték" : utasítás szintaktikával definiálhatjuk. A switch sajátossága, hogy minden egyes ilyen sor után, ha nem akarjuk, hogy a következő is végrehajtsódjon, egy break utasítást kell írunk. Ennek megfelelően a case "é" : ujKarakter="e"; break; sor abban az esetben, ha az utolsó leütött karakter "é" volt az ujKarakter változónak az "e" karaktert adja értékül, és befejezi a vizsgálatot. Mivel ezt minden ékezetes betűre elvégeztük, így mindegyik helyére saját ékezet nélküli párja helyettesítődik majd be. Ehhez azonban szükség van még egy lépésre: meg kell vizsgálnunk, hogy megváltoztattuk-e a karaktert, azaz az ujKarakter változóba került-e valami. Ha a feltétel igaznak bizonyul, akkor a beviteli mezőben lévő stringhez a függvény által visszaadott karaktert fűzzük, a feltétel hamis volta esetén nem történik semmi.

A múlt hónapban megpróbáltam gyakorlati példákon bemutatni, hogyan is használhatjuk a JavaScript kódokat a weblapunk kialakításakor. Most ezt a hagyományt követve még gyakorlatibb példákat szeretnék bemutatni, amik valós helyzetben is használhatók, és valódi problémákat oldanak meg. Az előző részben már volt egy kis program, az ékeztelenítő rutin, mely ilyen feladatot oldott meg, ennek most egy továbbfejlesztett változatával is megismerkedhetünk.

For ciklus használata

Az előző verziónak két, viszonylag nagy, hibája van. Az egyik, hogy csak az utolsó karaktert vizsgálja, így ha valaki utólag szúr be valamit a szövegbe, akkor abban már nem cseréli le az ékezeteket. A másik hiba, hogy ha kivág/beilleszt módszerrel nagyobb szöveget másolunk a beviteli mezőbe, akkor az előbb említett okokból szintén elmarad a kívánt hatás. Mindkét probléma orvosolható a következő módszerrel. Minden egyes bevitelkor, egy *for* ciklus segítségével, az egész szöveget leellenőrizzük. Következzen most a javított változat, és egy kis magyarázat hozzá:

```

<html>
<script>

function check()
{
    var vizsgaSzoveg=document.form.text.value;

    for (var i=1; i<=vizsgaSzoveg.length; i++)
    {
        var vizsgaKarakter=vizsgaSzoveg.substring(i-1,i);
        var ujKarakter="";
        switch (vizsgaKarakter) {
            case "é" : ujKarakter="e"; break;
            case "É" : ujKarakter="E"; break;

```

```

        case "á" : ujKarakter="a"; break;
        case "Á" : ujKarakter="A"; break;
        case "í" : ujKarakter="i"; break;
        case "Í" : ujKarakter="I"; break;
        case "û" : ujKarakter="u"; break;
        case "Û" : ujKarakter="U"; break;
        case "ú" : ujKarakter="u"; break;
        case "Ú" : ujKarakter="U"; break;
        case "ö" : ujKarakter="o"; break;
        case "Ö" : ujKarakter="O"; break;
        case "ó" : ujKarakter="o"; break;
        case "Ó" : ujKarakter="O"; break;
        case "ü" : ujKarakter="u"; break;
        case "Ü" : ujKarakter="U"; break;
        case "ö" : ujKarakter="o"; break;
        case "Ö" : ujKarakter="O"; break;
    }
    if (ujKarakter)
    {
        vizsgaSzoveg=vizsgaSzoveg.substring(0,i-1) +
        ujKarakter + vizsgaSzoveg.substring(i);
        document.form.text.value=vizsgaSzoveg;
    }
}
</script>
<body>
Ide lehet bepötyögni a szöveget, amiből majd kiszedjük az ékezetes
betűket:<br>
<form name="form">
<textarea name="text" rows="10" cols="30" onKeyUp="check()"></textarea>
</form>
</body>
</html>

```

Akkor lássuk mi is változott. A HTML *form*, illetve az ékezetek kiszűrését végző *switch-case* szerkezet, teljes egészében ugyan az, mint az előző programban. Változott viszont a függvényünk szerkezete, és néhány utasítása. Az első sorban egy egyszerű értékadással kiküszöböljük, a változónevek hosszú, és fáradságos begépelését. Ezek után egy *for* ciklus segítségével végigmegyünk az egész *string*-en, és egyesével levizsgáljuk a karaktereket. A karakterek eléréséhez a *substring* metódust használjuk a megfelelő paraméterezéssel. Megadjuk neki a kezdő és vég indexet, amelyek közötti részre kíváncsiak vagyunk. A kiválasztás után következik a már ismert vizsgálat.

Ellenőrzések

A valódi internetes oldalak kialakításakor általában szükség van adatok bevitelére a felhasználó részéről, azonban ezeket az adatok ellenőrizni kell a feldolgozásuk előtt. Mivel itt kliens oldali programozásról írunk, így az adatok feldolgozásával nem, csak az ellenőrzésükkel foglalkozunk. Ehhez szükség van HTML ismeretekre is, amit gyorsan átveszünk, hogy érthető legyen a probléma, és a megoldás is.

Már eddig is hoztunk létre *form*-okat (magyarra fordítva talán űrlapokat), de a valódi jelentőségükről még nem, vagy csak alig esett szó. Ezek az űrlapok biztosítják a felhasználó és a szerver közötti adatcserét. Az így kitöltött adatokat elküldjük a szerverre, ahol valamilyen CGI

(Common Gateway Interface) program feldolgozza azt. Az űrlap különböző beviteli elemekből áll, ezekből mutatok be most röviden néhányat:

típus	Tulajdonságok
text	Sortörések nélküli szöveg bevitelére alkalmas mező.
textarea	Többsoros szöveg bevitelekor használjuk
select	Választólista, több lehetséges, rögzített érték közül választhatunk.
button	Nyomógomb, ami a felhasználói interakciókat hivatott lekezeln.
submit	Nyomógomb, ami a <i>form</i> adatainak elküldésére szolgál.
hidden	Speciális rejtett mező, amiben a szerver oldali program számára fontos adatokat kezelhetünk.

Form néhány eleme, és tulajdonságaik

A példák során csak a *text*, és a *submit*, illetve a *button* objektumokkal fogunk megismerkedni, mivel a feladatok nagy része ezek segítségével jól bemutatatható. A későbbiekben igyekszem majd a többi elem bemutatására is.

Egy egyszerű *form* a következő elemekből áll: beviteli mezők és *submit* vagy *button* nyomógomb. Általában a *submit* gombot használjuk az adatok elküldésére, de kerülő úton a *button* gombbal is megoldható ez a feladat.

Lássunk akkor egy *form*-ot, melynek csak annyi a feladata, hogy két mező tartalmát elküldi a szervernek (a `<body>` és `<html>` tagokat most elhagyjuk):

```
<form name="form" action="feldolgoz.cgi" method="post">
név: <input type="text" name="nev"><br>
e-mail cím: <input type="text" name="email"><br>
<input type="submit" name="gomb" value="Elküld">
</form>
```

A *form*-nak körülbelül ezek azok a paraméterei amiket mindenképpen érdemes megadni. A *name* értelemszerűen az űrlap nevét jelenti, amivel hivatkozhatunk rá. Az *action* jelenti annak a CGI programnak a nevét, ami a feldolgozást végzi majd. A *method* paraméterben adhatjuk meg, hogy milyen módon küldjük az adatokat. A *post* érték azt jelenti, hogy az adatokat egy "csomagként" kapja meg a szerver oldali program.

Amit érdemes megfigyelni az a *submit* gomb paraméterezése. A *name* magától értetődően az elem nevét jelenti, a *value* pedig azt a szöveget tartalmazza, ami a gomb felirataként megjelenik.

Ha tehát a gombra kattintunk, a mezők tartalma a *feldolgoz.cgi* programnak adódik át.

A küldés előtt, mint már említettem, ellenőrizni kell az adatokat, és itt használhatjuk fel a *JavaScript*-et.

"Üres szöveg"

A feladatunk az lenne, hogy ellenőrizzük le küldés előtt, hogy valamelyik mező tartalmaz-e üres adatokat. Ha hiányos információt adna meg valaki, akkor figyelmeztessük, és állítsuk le az adatok küldését. Ehhez az előző *form*-ot kicsit át kell alakítanunk, és írni kell hozzá egy ellenőrző függvényt:

```
<html>
<head>
<script>
    function ellenoriz()
    {
```

```

        if (document.form.nev.value=="")
        {
            alert("Nem adtál meg nevet!");
            return false;
        }
        if (document.form.email.value=="")
        {
            alert("Nem adtál meg e-mail címet!");
            return false;
        }

        return true;
    }
</script>
</head>
<body>
<form name="form" action="feldolgoz.cgi" method="post">
név: <input type="text" name="nev"><br>
e-mail cím: <input type="text" name="email"><br>
<input type="submit" name="gomb" value="Elküld" onClick="return
ellenoriz()">
</form>
</body>
</html>

```

Lássuk hogyan is működik a dolog. Először is meg kell mondanunk a böngészőnek, hogy az *Elküld* gombra történő kattintáskor hívja meg az *ellenoriz()* függvényt, és figyelje is hogy milyen értékkel tér vissza. A *return* kulcsszóval mondhatjuk meg, hogy a visszatérési értéket figyelembe véve küldjük el a *form*-ot, vagy állítsuk le a folyamatot. Ha tehát a meghívott függvény *true*, azaz igaz értékkel tér vissza, a folyamat tovább fut, ha azonban *false*, azaz hamis értékkel tér vissza a küldés megszakad. Nincs más dolgunk tehát, csak megírni úgy a függvényt, hogy ezeket az értékeket adja vissza. Egy egyszerű *if* szerkezettel, pontosabban kettővel, vizsgáljuk a mezők tartalmát. A vizsgálat történhet a benne lévő tartalom, vagy a tartalom hossza szerint, én az előbbit választottam. Tehát ha a mező üres, akkor egy üzenetet küldünk a felhasználó felé, majd hamis értékkel térünk vissza. Ha mindkét vizsgálaton túljutottunk, az azt jelenti, hogy nem volt hiba, és visszaadhatjuk az igaz értéket.

Itt szeretném megemlíteni, hogy az ellenőrzést más helyen is lehet végezni, a másik eljárás egyébként a *form onSubmit* eseményét használni.

Az előbbi pár oldaban próbáltam meg bemutatni a JavaScript alapvető tulajdonságait. A JavaScriptben lévő objektumok metódusainak pontos leírása a JavaScript referenciákban telejs mértékben megtalálható.

Az XML

Bevezetés

Az XML a World Wide Web Konzorcium (W3C) ajánlása, amely kompatibilis egy sokkal régebbi, SGML (Standard Generalized Markup Language = szabványos általánosított jelölőnyelv) nevű ISO 8879/1986 szabvánnyal. Az SGML megfelelés azt jelenti, hogy minden XML dokumentum egyben SGML dokumentum is, de fordítva már nem igaz, azaz van olyan SGML leírás, ami nem XML megfelelő.

Az XML (Extensible Markup Language = bővíthető jelölőnyelv) adatleíró nyelv létrehozását a nyílt rendszerek térhódítása és az Internet tette szükségessé. A Java nyelv és az XML között van egy érdekes hasonlóság: a Java a futtatható formában hordozható programok nyelve, míg az XML a hordozható adat készítésének az eszköze.

Az XML dokumentumok (hasonlóan a Java forráskódhoz) unicode alapú szöveges karaktersorozatok, ahol az alapértelmezés az UTF-8 (tömörített unicode), de a dokumentum elején ettől eltérő kódolási eljárást is választhatunk.

Az XML formátumú dokumentumok fokozatosan kezdik leváltani a jelenleg széles körben elterjedt ASCII alapú szövegfájl alkalmazásokat (példa: interface-ek, konfigurációs fájl-ok). Ennek a tendenciának az az oka, hogy az ASCII fájl-ok semmilyen leíró, kezelő információt (meta adatot) nem tartalmaznak saját magukról (még az az alapvető tulajdonságuk sem őrződik meg, hogy melyik kódlapot használta a fájl készítője). Ezzel szemben az XML alapú dokumentumok a benne tárolt adatok vagy hivatkozások (szöveg, kép, ...) értékein felül rendelkeznek plusz információkkal is, ugyanakkor leírásuk – hasonlóan a HTML nyelvhez – továbbra is rendelkezik azzal a kellemes tulajdonsággal, hogy szövegalapú. Az XML dokumentumokban az adatok értékein túl olyan további címkéket és hivatkozásokat helyezhetünk el, amik utalnak az adat természetére, a dokumentum szerkezeti és tartalmi felépítésére, továbbá ezek az információk felhasználhatóak a dokumentum érvényességének vizsgálatához is.

Egy XML dokumentum nem tartalmaz utalást arra nézve, hogy egy alkalmazás (például Internet Explorer 5, továbbiakban: IE5), hogyan jelenítse meg, de vannak kiegészítő W3C ajánlások, amik ezt a hiányosságot megoldják.

Az XML úgy lett tervezve, hogy egy XML adatfolyamot leíró, illetve érvényességét meghatározó formális nyelvtant könnyű legyen definiálni, így az XML adatfolyam szintaxisvezérelt elemzése egyszerű. Ez utóbbi tulajdonság az, ami miatt az XML ma szinte ez egyetlen széles körben elterjedt, általános eszköz arra, hogy adatainkat hordozható formában tároljuk és továbbítsuk. Az XML ebben a megközelítésben a különféle jelölőnyelvek készítését leíró nyelv (meta nyelv). Amikor az XML segítségével megalkotunk egy új, konkrét jelölő nyelvet, akkor ennek az eredményét XML alkalmazásnak nevezzük.

Az első XML dokumentumunk létrehozása

Annak érdekében, hogy az eddig leírtak ne legyenek túlságosan elvontak, nézzünk egy konkrét példát! Legyen egy WEBADATOK adatbázisunk, amiben a VKONYV vendégkönyv tábla felépítése a következő:

- NEV VARCHAR(50)
- EMAIL VARCHAR(50)
- DATUM DATE
- SZOVEG VARCHAR(500)

Vizsgálódásunk pillanatában a VKONYV tábla a következő 2 bejegyzést tartalmazta:

X Y xy@freemail.hu 2002.01.31 Üdvözltem a WEB mesternek!

Z V valaki@mailbox.hu 2002.04.30 Tetszett a site :)

Az XML jellegzetessége, hogy az adatokat hierarchikus szerkezetben képes ábrázolni (reprezentálni), így egy XML adatfolyam egyben mindig egy fát is definiál. A VKONYV tábla tartalmát a következő XML formában állíthatjuk elő:

```
<?xml version="1.0" encoding="WINDOWS-1250" ?>
<VKONYV>
  <VENDEG sorszam="1">
    <NEV>X Y</NEV>
    <EMAIL>xy@freemail.hu</EMAIL>
```

```

<DATUM>2002.01.31</DATUM>
<SZOVEG>Üdvözetem a WEB mesternek!</SZOVEG>
</VENDEG>
<VENDEG sorszam="2">
<NEV>Z V</NEV>
<EMAIL>valaki@mailbox.hu</EMAIL>
<DATUM>2002.04.30</DATUM>
<SZOVEG>Tetszett a site :) </SZOVEG>
</VENDEG>
</VKONYV>

```

A fenti XML adatfolyamot vizsgálva láthatjuk, hogy eddig 2 vendég írt a vendégkönyvbe. Itt az ideje, hogy elgondolkodjunk azon, hogy miért éppen ebben a formában hoztuk létre a VKONYV tábla tartalmának XML-beli exportját, hiszen ezt megtehetjük volna például így is:

```

<?xml version="1.0" encoding="WINDOWS-1250" ?>
<VKONYV>
<VENDEG sorszam="1" NEV="X Y" DATUM="2002.01.31">
<EMAIL>xy@freemail.hu</EMAIL>
<SZOVEG>Üdvözetem a WEB mesternek!</SZOVEG>
</VENDEG>
<VENDEG sorszam="2" NEV="Z V" DATUM="2002.04.30">
<EMAIL>valaki@mailbox.hu</EMAIL>
<SZOVEG>Tetszett a site :) </SZOVEG>
</VENDEG>
</VKONYV>

```

Ez a többféle lehetőség annak a következménye, hogy az XML ajánlás az adatok értelmezését (szemantikáját) és a használható tag-ek (megjelölő címkék: VKONYV, NEV, ...) körét nem határozza meg, így annak kialakítása tervezési megfontolásokon alapul. A második megoldás jobban épít a tag-ek tulajdonság (attribútum) megadási lehetőségeire, aminek következtében a leíró tag-ek száma csökkent. A fentiek alapján belátható, hogy a VKONYV tábla tartalmát sokféle XML adatfolyamként elő lehet állítani. A tervezés során ki kell választani azt az XML formátumot, amit az adott feladat szempontjából jónak tartunk, majd el kell készíteni hozzá azokat a nyelvtani szabályokat, amik egyértelműen rögzítik azt, hogy például a továbbiakban mi a VKONYV első változatú XML adatfolyamát szeretnénk használni. A nyelvtani szabályok megfogalmazására a DTD (Document Type Definition = a dokument típus definíció) nyelv szolgál, amit a későbbiekben kerül ismertetésre. Példából meggyőződhattünk arról is, hogy az XML leírás valóban nem tartalmaz olyan nyelvi elemet, ami a dokumentum kinézetét határozná meg, hiszen az csak a dokumentum szerkezetét, értelmezését és konkrét adatait tartalmazta. Az első XML dokumentumot egy vkonyv.xml fájl-ban tárolva az Internet Explorer 5.0 alapértelmezett esetként egy fa formában jeleníti meg, ahol a +/- jelekkel lehet a fa ágait kinyitni vagy becsukni. A fájl első sorában a "version=1.0" megadása kötelező, a kódolás megadása választható, de ha nem adjuk meg, akkor az alapértelmezett érték UTF-8 lesz.

XML alkalmazások

Az előzőekben már említésre került, hogy az XML segítségével megalkotott jelölőnyelveket XML alkalmazásoknak nevezzük. Alaposan körbenézve az egyes applikációk (Linux, Windows programok) között látható, hogy az XML szinte már alapvető konfigurációs és adattároló eszközzé vált. A Linux GNOME gnumeric, AbiWord a dokumentum tárolási formájául az XML-t választotta. Linux alatt az egyik legjobb grafikus elemkészlet a QT, amihez létezik egy QT Designer (a QT elemkészlet és a designer a Trolltech alkotása). A QT Designer-ben – a Delphi vagy Visual Basic-hoz hasonlóan - vizuálisan lehet felhasználói interface-eket készíteni, majd azokat szabványos *.ui fájl-okba menteni. Amikor a QT Designer használatával egy "Teszt Form" címsorú és "frmTeszt" nevű formba egy "OK" feliratú, "btnOK" nevű gombot

tervezünk, akkor a következő XML fájl jön létre:

```
<!DOCTYPE UI><UI>
<class>frmTeszt</class>
<widget>
<class>QWidget</class>
<property stdset="1">
<name>name</name>
<cstring>frmTeszt</cstring>
</property>
<property stdset="1">
<name>geometry</name>
<rect>
<x>0</x>
<y>0</y>
<width>342</width>
<height>155</height>
</rect>
</property>
<property stdset="1">
<name>caption</name>
<string>Teszt Form</string>
</property>
<widget>
<class>QPushButton</class>
<property stdset="1">
<name>name</name>
<cstring>btnOK</cstring>
</property>
<property stdset="1">
<name>geometry</name>
<rect>
<x>20</x>
<y>20</y>
<width>104</width>
<height>28</height>
</rect>
</property>
<property stdset="1">
<name>text</name>
<string>OK</string>
</property>
</widget>
</widget>
</UI>
```

A fenti két példán kívül most csak megemlítek néhány további, elterjedt XML alkalmazást:

- .NET projekt a Microsofttól, teljesen XML alapú fejlesztői eszköz, és ebből kiindulva az új Windowsokból például az eddigi registry kezelés helyére is egy XML alapú mechanizmus lép.
- XHTML nyelv: A HTML nyelvet írja le, illetve kissé módosítja annak érdekében, hogy a HTML XML megfelelő legyen.
- XSQL nyelv: Adatbázis kezelő műveletek XML megfelelő leírására szolgáló jelölő nyelv
- MathML (Mathematics Markup Language)

A szabványosított XML alkalmazások köre egyre szélesebb, amit folyamatosan figyelemmel követhetünk a W3C honlapon. Az egyik legfontosabb XML alkalmazás az XSL (Extended Style Language = bővített stíluslap leíró nyelv), amivel – a CSS alternatívájaként – az XML dokumentumok képi megjelenítésének kinézetét lehet meghatározni. Az XML általános információ leíró képessége miatt egyre gyakrabban találkozunk XML alapú konfigurációs fájl-

okkal is. Példaként mindenkinek ajánlom a Jakarta-Tomcat web.xml nevű fájlt tanulmányozásra, ami a WEB hely aktuális kialakításának konfigurációját tartalmazza.

Az XML adatfolyam felépítése (XML fájl-ok)

Az XML adatfolyamot formai és nyelvtani szempontból is vizsgálhatjuk. Ebben a pontban a formai elemzés szempontjait tekintjük át. Egy XML dokumentumot jól formázottnak (well formed) nevezünk, ha a következőben részletezett XML formai feltételeinek eleget tesz.

Elemek (Elements)

Az elemek a legalapvetőbb részei az XML dokumentumoknak. A `<NEV>X Y</NEV>` részlet egy konkrét elem. Egy XML elem általában a következő részeket tartalmazza:

- Az elemet bevezető `<ElemCimke>` tag, amit csúcsos zárójelek között kell megadni.
- Az elemhez tartozó adat (Pl. `X Y`). Nem minden elem tartalmaz adatot.
- Az elemet záró `</ElemCimke>` tag.

A nyitó és záró tag-ek feladata az, hogy megjelöljék és elnevezzék az általuk közrefogott adatot. Ezzel az adataink természete és tartalma is nyomonkövethetővé válik, emiatt fontos, hogy a címke nevek kifejezők legyenek. Lényeges kiemelni, hogy a hagyományos XML meghatározás nem ad olyan precíz adatábrázoló és kezelő eszközt a kezünkbe, mint a programozásban megismert típus fogalma, azonban már létezik olyan W3C ajánlás (XML Schema definíció), ami ennek a szigorú követelménynek is eleget tesz. Időnként előfordul, hogy egy tag nem fog közre semmilyen adatot. Erre példa az, ha valakinek nincs e-mail címe: `<EMAIL></EMAIL>`. Ezt a terjengős leírást az XML-ben a következő módon rövidíthetjük: `<EMAIL/>`. A tag nevekre a következő szabályokat kell betartani:

- Csak betűvel vagy aláhúzás jellel kezdődhet
- Tartalmazhat betűt, számjegyet, pontot, kötőjelet, aláhúzás jelet
- A kis és nagybetűk különbözőnek számítanak

Megjegyzés

Az XML dokumentumban a HTML nyelvben megszokott megjegyzést használhatjuk.

```
<!--  
Ez egy XML megjegyzés, ami lexikailag olyan mint a HTML comment  
-->
```

Ezeket a részeket az XML feldolgozó (elemző, érvényesség vizsgáló, megjelenítő) alkalmazások figyelmen kívül hagyják.

Tulajdonságok (attributes)

Az XML elemek tetszőleges számú tulajdonsággal is rendelkezhetnek. A VKONYV XML második változatából tekintsük a következő részletet:

```
<VENDEG sorszam="2" NEV="X V" DATUM="2002.04.30">
```

Itt három tulajdonságot határoztunk, adtunk meg a VENDEG elem részére: sorszam, NEV, DATUM. A tulajdonságok általában így néznek ki:

```
<element nev1="érték1" nev2="érték2" ...>
```

...

```
</element>
```

Egyed hivatkozások (entity references)

Az egyedek olyanok, mint a maróhelyettesítések. Minden egyednek egy egyedi névvel kell rendelkeznie. Legyen az "EgyedNev" egy egyednév, ekkor ennek a meghívása a következő: "&EgyedNev;", azaz "&" jellel kezdődik és ";"-vel fejeződik be. Ha megnézzük az AbiWord által készített XML dokumentum "ó" részletét, akkor ott ez egy egyedhivatkozás az "ó" betűre.

Léteznek előre definiált egyedek, mint például a "gt" nevű. Az ">" hatása olyan, mintha az XML adatfolyamba egyből a ">" jelet írtuk volna. Az egyedek fogalma lehetővé teszi azt is, hogy bármilyen unicode karaktert helyezzünk el a szövegbe. Ekkor az egyednév "#szám" vagy "#xszám" alakú, ahol az első a decimális, a második a hexadecimális megadási mód. A kis "ó" hexa unicode kódja 0x0151. Ezt mint egyedet a következő módon hívhatjuk meg: "ő". Az XML ajánlás lehetővé teszi, hogy saját egyedeket is létrehozzunk. Ez az <!ENTITY myEgyed "Érték"> szintaxis-sal lehetséges, amit "&myEgyed;"-vel hívhatunk meg és az a hatása, mintha közvetlenül az "Érték" karakterfüzért írtuk volna le. Az egyedek definiálása a jelölő nyelv nyelvtani szabályait meghatározó DTD leírás része, ezért erre ott még visszatérünk.

Feldolgozási utasítások (processing intructions)

A feldolgozási utasítások nem részei az XML-nek. Szerepük a fordítóprogramoknak küldött direktívákhoz hasonlítható. Ezek, a megjegyzésekhez hasonlóan, nem részei a dokumentum szöveges tartalmának, de követelmény az XML feldolgozókkal szemben, hogy továbbadják ezeket az utasításokat az alkalmazások számára. Megadási módjuk: <?target name?>. Egy alkalmazás csak azokat az utasításokat veszi figyelembe, amelynek a célját felismeri, az összes többit figyelmen kívül hagyja. A célt (target) követő adatok az alkalmazásnak szólnak, megadásuk opcionális.

A CDATA szakaszok

Az ebben a szakaszban lévő CDATA, azaz karakteres adat interpretálás nélkül átadódik az XML adatfolyamot fogadó alkalmazásnak. A szakasz szintaxisa a következő:

<![CDATA[Itt van a tetszoleges string]]>. A megadott karakterfüzér a "]]" kivételével bármi lehet.

Névterek

Mi van akkor, ha több ország iskoláiból kapunk tanulmányi eredményeket (természetesen XML formátumban), és ezeket szeretnénk egy nagy XML dokumentumba foglalni? Azonban az egyik országban az 1-esnek örülnek a gyerekek, míg nálunk az 5-ösnek. Ha egyszerűen összefűznénk az adatokat, akkor lehetetlen lenne összehasonlítani a különböző eredményeket. Valahogyan definiálni kellene az összesített XML dokumentumban a számok kétféle értelmezését. Ezután minden tételnél megjelöljük, hogy melyik tétel melyik értelmezéshez kapcsolódik, így később egyértelműen kibányászhatók az érdemjegyek. Az ilyen jellegű problémák feloldására vezették be a névtér (namespace) fogalmát az XML szabványban. A dokumentum elején meghatározzuk a leírni kívánt típusokat, és a tagoknál megjelöljük, hogy azok melyikhez tartoznak. Névtér használata nélkül így nézne ki az összevont érdemjegy táblázat:

```
<Osztályzatok>
<oszt>4</oszt>
<oszt>2</oszt>
</Osztályzatok>
```

Melyiknek örülne egy gyerek? Az attól függ, hogy honnan kerültek bele az adatok. Tegyük egyértelművé az adatok értelmezését:

```
<Osztályzatok
xmlns:AzÖtösaJó="http://iskola.hu/Ot"
xmlns:AzEgyesaJó="http://iskola.hu/Egy" >
<AzÖtösaJó:oszt>4</AzÖtösaJó:oszt>
<AzÖtösaJó:oszt>5</AzÖtösaJó:oszt>
<AzEgyesaJó:oszt>2</AzEgyesaJó:oszt>
</Osztályzatok>
```

Mit láthatunk itt? Készítettünk két névteret. Az egyiknek “AzÖtösaJó”, a másiknak “AzEgyesaJó” a neve. A nevek tetszőlegesen választhatók, lehetne Jancsi és Juliska is, de nem árt, ha a választott névtér neve utal a funkciójára. Az xmlns:névtérnév="valami egyedi" segítségével definiálhatunk névteret egy taghoz. A “valami egyedi”-nek az egész világon egyedinek kell lenni, így elvileg a világ összes XML dokumentumát konfliktus nélkül össze lehetne fűzni, mert a “valami egyedi” egyértelműen azonosítja a forrást, így nem lehetnek névütközések. A névtérnév egy álnév (alias) a hosszú, egyedi névtér névre. Így a tagokban könnyebb rájuk hivatkozni. A névtér öröklődik, így a gyermek tagok (mint az oszt) is használhatják a szülőben (Osztályzatok), nagyszülőben, dédszülőben, satöbbi definiált névteret. A névterek egyedi elnevezésére két módszer adódik kézenfekvően: használjunk valamilyen URL-t, vagy alkalmazzuk a Windows-os GUID-ot (Globally Unique Identifier). Mivel a W3C konzorcium, az XML gazdája nem csak a Microsoft által befolyásolt szervezet, ezért a legtöbb névtér deklarációban URL-t látunk, és nem GUID-ot, mint egyedi névtér azonosítót. Amikor egy program feldolgozza az így összesített adatokat, akkor minden egyes tag feldolgozása során megnézi, hogy az adott tag melyik névtérhez tartozik. A névtereket tudnia kell előre ahhoz, hogy tudja a dokumentumban található számok, mint osztályzatok értelmét. Ha az adatok zöme egyféle értelmezésű, így csak egy kis százaléka kellene, hogy valami más névtérbe tartozzon, akkor érdemes kihasználni az alapértelmezett névtér (default namespace) lehetőségét, hogy rövidebb dokumentumot kapjunk. Ha az előbbi példánk bejövő adathalmazában a hazai (az ötös a legjobb jegy) értelmezés a leggyakoribb, akkor tegyük azt az alapértelmezett névtérre, így csak azokat az adatokat kell megjelölni, amelyeket nem így kell értelmezni. Az alapértelmezett névteret úgy deklaráljuk, hogy nem adunk álnevet a kívánt névtérazonosítónak:

```
<Osztályzatok
xmlns="http://iskola.hu/Ot"
xmlns:AzEgyesaJó="http://iskola.hu/egy" >
<oszt>4</oszt>
<oszt>5</oszt>
<AzEgyesaJó:oszt>2</AzEgyesaJó:oszt>
</Osztályzatok>
```

Kivételek persze mindig vannak. A névtéröröklődési láncot megszakíthatjuk bármely ponton úgy, hogy egy üres névtérdefiníciót alkalmazunk a gyermekelemre, és attól a szinttől lefelé már nem lesz érvényes az alapértelmezett névtér:

```
<Osztályzatok
xmlns=http://iskola.hu/Ot >
<oszt xmlns="">
<o1/>
<o2/>
</oszt>
<oszt>5</oszt>
```


</Osztályzatok>

A példában a "http://iskola.hu/Ot" alapértelmezett névtérrel kapcsoljuk ki az első <oszt> elemre, és az ő 2 gyermekére (<o1/> és <o2/>). Fontos, hogy az alapértelmezett névtér nem vonatkozik az attribútumokra, csak a tagokra, azaz az <oszt megj="Okos gyerek"> esetén a megj attribútum az nem tartozik az alapértelmezett névtérhez (semmilyen névtérhez sem tartozik). Ha azt szeretnénk, hogy ahhoz tartozzon, akkor minden egyes attribútum elé is ki kell írni a névtér előtagot: <oszt AzEgyesJó:mej="Okos gyerek">. Ezzel még sokszor fogunk találkozni az adatbázis-XML kapcsolatok leírásánál.

Az XML adatfolyam érvényességének ellenőrzése

Az előző részben azt vizsgáltuk, hogy egy XML dokumentum mikor jól formázott. A helyes formázottság tényét nagyon könnyen ellenőrizhetjük a különféle XML feldolgozó programokkal, például az IE5 böngészővel. A vkonyv.xml jól formázott, mert az IE5 elemezni tudta, amit a fa szerkezet megjelenítésével is visszaigazolt. A jól formázottság azonban még nem jelenti azt, hogy az XML dokumentum helyes, hiszen ha mi a VKONYV adathalmazt az első formátumú XML felépítésben szeretnénk látni, akkor például a második formátum nem érvényes, ezért azt az elemző programok vissza fogják utasítani. A mai XML elemzők általában a jól formázottság mellett lehetővé teszik az érvényesség vizsgálatát is. Az XML egyik legnagyobb erőssége, hogy saját tag neveket használhatunk, azonban a VKONYV példája arra is rámutat, hogy szükség van egy nyelvtani szabályokat meghatározó specifikációra is, amit az XML elemző program az érvényesség ellenőrzéséhez tud felhasználni. Az XML elemek egy fa szerkezetet építenek fel, aminek természetesen mindig van egy gyökér eleme. A VKONYV XML karaktersorozatban a VKONYV elem a gyökér. Ez az a pont, ahonnan az XML dokumentum nyelvtani elemzése indul (a formális nyelvtanok START szimbólumának felel meg). A DTD nyelven megfogalmazott nyelvtani szabályok egy karaktersorozatot alkotnak, tárolásukra két lehetőség van:

- Egy fájl-ban tároljuk (Pl.: vkonyv.dtd)
- Az XML dokumentum elején tároljuk

Tegyük fel, hogy valaki elkészítette nekünk (hiszen mi még nem ismerjük az elkészítés módját) a VKONYV XML adatfolyam első változatának, mint XML dokumentumtípusnak a DTD-ben megfogalmazott nyelvtanát és ezt a vkonyv.dtd fájl-ban átadta nekünk. Ekkor az XML dokumentumunk első sora után a következő sort lehet beszúrni:

```
<?xml version="1.0" encoding="WINDOWS-1250" ?>
<!DOCTYPE VKONYV SYSTEM "vkonyv.dtd">
<VKONYV>
<VENDEG sorszam="1">
...
</VKONYV>
```

A beszúrt sor a "<DOCTYPE " résszel kezdődik, majd a következő szó az XML dokumentum gyökéreleme. A harmadik szó "SYSTEM" vagy "PUBLIC" lehet. A sor utolsó része megadja azt, hogy melyik fájl-ban tárolódnak a nyelvtani szabályok. Amennyiben nem akarjuk a nyelvtant külön fájl-ban tárolni (pl.-ul azért mert nagyon egyszerű), akkor az eredeti XML adatfolyam elejét így kell kiegészíteni:

```
<?xml version="1.0" encoding="WINDOWS-1250" ?>
<!DOCTYPE VKONYV [
Ide jön az a karaktersorozat, amit egyébként
a vkonyv.dtd-ben tárolnánk
]>
<VKONYV>
<VENDEG sorszam="1">
...
</VKONYV>
```

A beágyazott DTD szabályokat tehát a fent látható módon “[...]” zárójelek között adhatjuk meg. Ennyi előzetes után röviden tekintsük át azt, hogyan lehet a DTD szabályrendszereket megfogalmazni.

Elem típus deklaráció (Element Type Declaration)

Az elem típus deklarációk azonosítják az elemek neveit és tartalmukat. A DTD-ben ez az azonosítás nem éri el a programozási nyelvek típusfogalmának megfelelő részletezettség szintet. A szemléletesség kedvéért nézzük meg a VKONYV XML adatfolyam első verziója <VKONYV> elemének típus deklarációját:

```
<!ELEMENT VKONYV (VENDEG)* >
```

A fenti sor szavakkal elmondva azt jelenti, hogy a VKONYV elem VENDEG típusú elemek sorozata, ahol a “*” azt jelöli, hogy 0 vagy több tagja lehet ennek a sorozatnak (a mi vkonyv.xml fájl-unkban most 2 hosszú ez a sorozat). A számosság kifejezésére a “*”-on kívül még a “+” (1 vagy több elem) és a “?” (0 vagy 1 elem) jelek szerepelhetnek. Amikor egy név mellett nincsenek írásjelek, akkor pontosan egyszer kell előfordulnia. Most nézzük meg, hogy a <VENDEG> elemre milyen nyelvtani szabályok állapíthatók meg!

```
<!ELEMENT VENDEG (NEV, EMAIL?, DATUM, SZOVEG)>
```

Szavakban ez azt jelenti, hogy egy VENDEG elem a NEV, EMAIL, DATUM, SZOVEG elemek ebben a sorrendben (és nem más sorrendben!) vett szerkezetéből áll. Itt most egy fontos dolgot lehet észrevenni, mert ez a nyelvtani szabály már kizárja azt, hogy a második típusú VKONYV XML folyam érvényes legyen, hiszen ott a VENDEG elem csak az EMAIL és a SZOVEG elemeket tartalmazhatja. Az elem neveken felül van egy fenntartott speciális szimbólum a #PCDATA, ami a karakteres adatot jelöli. Ez teszi lehetővé, hogy például a NEV elem nyelvtani szabályát megadjuk:

```
<!ELEMENT NEV (#PCDATA)>, azaz a NEV elem egy elemzett karakterfüzér.
```

Már láttuk, hogy léteznek üres elemek is, azaz olyan tag-ek, amikhez nincs adat (tartalom) rendelve. Amennyiben a <KEDVES_VENDEG/> egy üres tag, akkor ehhez így adható meg a nyelvtani szabály:

```
<!ELEMENT KEDVES_VENDEG EMPTY>
```

Itt az EMPTY a kulcsszó, ami azt jelzi az elemző programnak, hogy a KEDVES_VENDEG egy üres elem, így nincs ennek az elemnek további szerkezeti felépítése. Ebben a példában ez valószínűleg egy jelzőként szerepel. Nézzünk egy másik üres elemet is: <NEM_KEDVES_VENDEG/>. Ekkor elképzelhető, hogy vendégkönyvünkben fel akarjuk tüntetni ezt a logikai tulajdonságot a következő XML szerkezet segítségével:

```
<MINOSITES> <MINOSITES>  
<KEDVES_VENDEG/> vagy <NEM_KEDVES_VENDEG/>  
</MINOSITES> </MINOSITES>
```

Itt a <MINOSITES> elem nyelvtani szabálya választható elemekből áll, amit a tartalom meghatározáskor a pipe karakterrel fejezünk ki:

```
<!ELEMENT MINOSITES (KEDVES_VENDEG | NEM_KEDVES_VENDEG)>
```

A tulajdonságlista (attributum lista) deklarációja

Az eddigiekből már tudjuk, hogy a tulajdonságlista az elem jellemzésére szolgál. Természetesen ennek felépítése is a nyelvtan része. Itt adhatjuk meg, hogy az egyes elemek milyen jellemzőket tartalmazhatnak és azok milyen értékeket vehetnek fel. Nem mindegyik elemnek van tulajdonságlistája. A VKONYV XML folyam <VENDEG> elemének például egy tulajdonsága van, a sorszám. Az tulajdonságlisták általános formája a következő:

```
<!ATTLIST egy_elem_neve  
tulajdonság_név1 tulajdonság_jellegel jelz•
```

```
...  
tulajdonság_név_n tulajdonság_jellege_n jelz•  
>
```

A <VENDEG> elem példájánál maradva, így adhatjuk meg a “sorszam” tulajdonság nyelvtanát:

```
<!ATTLIST VENDEG sorszam CDATA #REQUIRED>
```

A fenti sor azt mondja meg az elemző program részére, hogy a VENDEG elemnek van egy sorszám nevű tulajdonsága, ami CDATA jellegű és kötelezően szerepelnie kell az elem minden előfordulásánál. Az egyes tulajdonságok jellege (típusa) a következő hatféle lehet:

- CDATA: Csak annyit akarunk ekkor állítani, hogy a tulajdonság egy tetszőleges karaktersorozat.

- ID: Az ID attribútum értékének egy névnek kell lennie.

- IDREF vagy IDREFS

- ENTITY vagy ENTITIES

- NMTOKEN vagy NMTOKENS

- Névlista: Ez egy egyszerű felsorolás típus. Pl.: (alma, körte, szilva)

A jelzőből a következő négy fajta létezik:

- **#REQUIRED:** Az elem minden dokumentumbeli előfordulásánál explicit módon meg kell adni az attribútum értékét.

- **#IMPLIED:** Az attribútum megadása nem kötelező és nincs alapértelmezett értéke. Ha nincs érték megadva, akkor az XML feldolgozónak nélküle kell tovább haladnia.

- **Egy érték:** Ekkor ez egy alapértelmezett értéke lesz az attribútumnak. Példa: “3.14”

- **#FIXED “érték”:** Ekkor nem kötelező az attribútum, de ha megadjuk, akkor ennek az értéknek kell lennie.

Egyed deklaráció (Entity declaration)

Az XML adatfolyam jól formált felépítése szakaszban már szó volt az egyedekről, most leírjuk azt is, hogyan és hol hozhatjuk létre őket. A DTD leírásban a következő háromfajta egyed lehetséges:

- Belső egyedek

- Külső egyedek

- Paraméter egyedek

A belső egyedek egy maróhelyettesítő mechanizmust valósítanak meg. Az

```
<!ENTITY BMGE “Budapesti M•szaki és Gazdaságtudományi Egyetem”>
```

sor létrehoz egy BMGE nevű egyedet, amit az XML szövegben “&BMGE;” alakban hívhatunk meg, melynek hatására megtörténik a szöveghelyettesítés. A Külső egyedek lehetővé teszik, hogy másik fájl tartalmára hivatkozzunk az XML dokumentumból. Így tartalmazhatnak szöveges és bináris adatot is. Ha a külső fájl szöveges adatot tartalmaz, akkor az, az eredeti dokumentumba illesztődik a referencia helyére és úgy kerül feldolgozásra, mintha a hivatkozó dokumentum része lenne. A bináris adat értelemszerűen nem kerül elemzésre. Paraméter egyedek csak a dokumentum típus deklarációban fordulhatnak elő. A paraméter egyedek deklarációja onnan ismerhető meg, hogy nevük előtt százalékkal “%” jel található. A paraméter egyed hivatkozások a dokumentum típus deklarációban azonnal feloldásra kerülnek, így a helyettesített szöveg is része a deklarációnak. Nézzünk egy példát, ahol két elem szerkezetét adjuk meg ezzel a módszerrel:

```
<!ELEMENT KONYV (ELOLAP, LAPOK+, ZAROLAP)>
```

```
<!ELEMENT FUZET (ELOLAP, LAPOK+, ZAROLAP)>
```

Kihasználva a hasonlóságot hozzunk létre egy paraméter egyedet:

```
<!ENTITY %TARTALOM “ELOLAP, LAPOK+, ZAROLAP”>
```

Az így kapott egyeddel a KONYV és FUZET elemek nyelvtani szabályai megfogalmazhatóak:

```
<!ELEMENT KONYV (%TARTALOM;)>
<!ELEMENT FUZET (%TARTALOM;)>
```

A VKONYV XML adatfolyam nyelvtani szabályai

A DTD-vel való ismerkedésünk befejezéseként alkossuk meg a VKONYV XML adatfolyam nyelvtani szabályait, amit egy vkonyv.dtd fájl-ban tárolhatunk, ezzel a tartalommal:

```
<!ELEMENT VKONYV (VENDEG)* >
<!ELEMENT VENDEG (NEV, EMAIL?, DATUM, SZOVEG)>
<!ATTLIST VENDEG sorszam CDATA #REQUIRED>
<!ELEMENT NEV (#PCDATA)>
<!ELEMENT EMAIL (#PCDATA)>
<!ELEMENT DATUM (#PCDATA)>
<!ELEMENT SZOVEG (#PCDATA)>
```

XDR, XSD

XDR, XML Data-Reduced:

Az XDR a nagyreményű Microsoft DTD utód - volt. A Microsoft gyorsan lemondott a DTD használatáról, és gyors ütemben belekezdett egy XML formátumú sémaleíró nyelv kidolgozásába, amelyet a Word Wide Web konzorciumnak is elküldött szabványosításra. Megszületett az XDR. Amellett, hogy XML formátumú még jóval flexibilisebb is, mint a DTD. A DTD-ben leírt struktúrának maradéktalanul meg kell felelni egy XML dokumentumnak. Az XDR is tud ilyen szigorú lenni, de emellett elő lehet azt is írni, hogy az ellenőrizendő dokumentum egyes részeiben lehetnek további elemek is, amelyet a séma nem ír le. Például egy személyről szóló XML adatlapban kötelezővé tesszük a név, születési dátum és az anyja neve elemeket, de ezen felül megengedjük, hogy egy buzgó gazdi például a kutyája nevét és haja színét is beleszerkessze a dokumentumba. A hangsúly nem azon van, hogy előírhatunk opcionális elemeket, hanem azon hogy megengedhetünk olyan elemeket is, amelyekről a séma készítésekor még nem is tudtuk, hogy lesznek. Ehhez kapcsolódó szolgáltatás, hogy XDR segítségével le lehet szabályozni a dokumentum egy részét is, nemcsak a teljes egészet. DTD-vel természetesen csak az egész dokumentumra lehet szabályokat definiálni. Emellett az XDR bővíthető, azaz az igények megváltozásakor nem kell a sémát kidobni, csak azokat az XML dokumentumokat fogadjuk el érvényesnek, amelyek szerkezete megfelel a formális leírásban szereplő feltételeknek. Például egy megrendelést leíró XML dokumentumra valószínűleg kikötnénk, hogy benne kell legyen a megrendelő neve, címe, adószáma stb. Ha a kapott megrendelés .xml-ben nem szerepel minden kívánatos adat, akkor visszadobjuk a megrendelést, mert nem érvényes. Az XML dokumentum szerkezetének, más néven sémájának leírására több módszer is a rendelkezésünkre áll, melyek fokozatosan, a használat során fejlődtek ki. Nézzük meg milyen egy másik névtér bevezetésével kiegészíteni a meglevőt. Utolsó, de nagyon fontos szolgáltatás az XDR-ben, hogy az elemek és attribútumoknak meg lehet adni a típusát (egész szám, dátum stb.). A DTD-ben minden szöveggént van deklarálva. A legtöbb, a közeli múltban fejlesztett Microsoft termék XDR-t használ sémaleírásra. Azonban már a legújabb az XSD, XML Schema Definition, amely most a leginkább aktuális sémaleíró nyelv. A Biztalk Server, illetve a .NET XML osztályok már tudják – tudni fogják ezt a sémaleírást is kezelni (természetesen az XDR mellett). A Visual Studio 7 egyik alapszolgáltatása az XSD sémák grafikus szerkesztése, konverziója XDR-ből XSD-be, XML dokumentumból XSD generálása stb.

Az XML adatfolyam megjelenítése

Az XML fájl-ok nem tartalmazzak semmilyen információt arra nézve, hogy hogyan kell őket megjeleníteni. Azt láttuk, hogy az IE5 egy fa formában jeleníti meg az XML dokumentációkat,

kihasználva azt, hogy azok egy-egy fát írnak le. Ennek a hiányosságnak a megoldására két szabványos megoldás is létezik:

- CSS (Cascaded Style Sheets)
- XSL (Extended Style Language)

A CSS

A CSS részletes leírását lásd DHTML részen belül, így itt csak a használatát mutatjuk be. A CSS leírások jellemzően *.css fájl-okban találhatóak meg. Legyen adott egy vkonyv.css kinézet leírásunk. Ekkor az első változatú VKONYV XML karakterfolyam megjelenítésére a CSS használatot a következő utasítással írhatjuk elő a megjelenítő alkalmazás részére:

```
<?xml version="1.0" encoding="WINDOWS-1250" ?>
<?xml-stylesheet type="text/css" href="vkonyv.css" ?>
<VKONYV>
<VENDEG sorszam="1">
...
</VKONYV>
```

A példa kedvéért nézzünk egy nagyon egyszerű CSS-t, amit a vkonyv.css fájl-ban található el. A fájl tartalma:

```
NEV { display: block; font-size: 20pt; font-weight: bold; }
EMAIL { display: block; font-size: 15pt; font-weight: bold; }
DATUM { display: block; font-size: 12pt; font-weight: bold; }
SZOVEG { display: block; font-size: 10pt; font-weight: bold; }
```

Érdekességgéppen leírjuk, hogy ezt a formát azért nevezi a szabvány cascade-nak, mert a megjelenítési beállítások különféle szintjeit egymás után vizsgálja meg a megjelenítő alkalmazás (Ezek a szintek: a böngésző alapértelmezései, egy külső stíluslap definíciói, a <style>...</style> tag által lokálisan megadott lehetőség, direkt stílusjegyek: ,).

Az XSL

Az XSL több, mint egy egyszerű stíluslap meghatározó nyelv, inkább egy olyan szövegfeldolgozónak tekinthető, mint a Perl vagy a UNIX-os AWK nyelvek. A feldolgozás eredmény karaktorsorozata természetesen egy HTML dokumentum is lehet, amit a böngészők meg tudnak jeleníteni. Az XSL két önálló nyelvi részből áll:

- a transzformációs nyelvből (XSL Transformation) és
- a formázó (XSL Formatting Objects, XSL-FO) nyelvből.

A transzformációs nyelv elemeivel olyan szabályokat definiálhatunk, amelyek segítségével egy XML dokumentumból egy másik dokumentumot, illetve általánosabban fogalmazva bármilyen byte-sorozatot hozhatunk létre. Ennek következtében a létrehozott dokumentum nem szükségszerűen lesz XML megfelelő, hiszen ezzel a módszerrel RTF, PDF, HTML, ... dokumentumok is gyárthatók.

Az XSL Formatting Objects (XSL-FO) az XSL ajánlás második része. Az XSL-FO egy olyan részletes modell, amellyel XML szintaxisú stíluslapokat hozhatunk létre, amik – a CSS-hez hasonlóan – meghatározzák, hogyan kell az XML dokumentumnak megjelennie. Az XSL Formatting Objects (XSL-FO) a HTML+CSS formátumú megjelenítésnél sokkal többet nyújt. Lehetőség van a különböző tipográfiai, a nyomdai kiadványoknál megszokott oldal és szövegformázási elemek (fejléc, lábléc, tartalomjegyzék, ...) szabályainak leírására. Az XSLFO dokumentumokat általában XSL transzformációval hozzuk létre. A szemléletesség érdekében elkészítettünk egy vkonyv1.xsl fájl leírást, amit az alább látható módon – a CSS-hez hasonlóan – építhetünk be egy XML dokumentumba:

```
<?xml version="1.0" encoding="WINDOWS-1250" ?>
```

```

<?xml-stylesheet type="text/xsl" href="vkonyv1.xsl"?>
<VKONYV>
<VENDEG sorszam="1">
...
</VKONYV>
A teljesség kedvéért nézzük meg a vkonyv1.xsl fájl tartalmát is:
<?xml version="1.0" encoding="WINDOWS-1250" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
<html><body> <H1>VENDEGKÖNYV</H1>
<xsl:apply-templates/>
</body></html>
</xsl:template>
<xsl:template match="VKONYV"> <br/> <xsl:apply-templates/> <hr/>
</xsl:template>
<xsl:template match="VKONYV/VENDEG"> <xsl:apply-templates/>
</xsl:template>
<xsl:template match="VKONYV/VENDEG/NEV">
<hr/>Neve: <b>
<xsl:value-of select="."/> </b>
</xsl:template>
<xsl:template match="VKONYV/VENDEG/EMAIL">
Email címe: <b>
<xsl:value-of select="."/> </b>
</xsl:template>
</xsl:stylesheet>

```

Az ábra mutatja a VKONYV XML adatfolyam kinézetét akkor, amikor a vkonyv1.xsl stíluslapot használtuk. Észrevehető, hogy ez a módszer a megjelenítendő adatok körét is könnyen kezelni tudja (itt most csak a Név és Email lett megjelenítve). Az XSL stíluslap nyelv egy XML alkalmazás. Az XSL lapok a mintaillesztés módszerét használják az egyes elemek megjelenítéséhez. Itt a tag nevek “xsl:”-tal kezdődnek, utalva arra, hogy itt olyan XML elemek vannak, amiket az XSL alkalmazás keretében akarunk használni. Ez a megoldás egy XML névtér (namespace) szabványon alapul, aminek az a célja, hogy az egyes XML alkalmazások tag-jei külön-külön névtérben legyenek (hasonlóan a C++ névtereihez). Itt az “xsl:”-ben az “xsl” szó a névtér neve. Röviden nézzük át, hogyan határozza meg a megjelenést a vkonyv1.xsl tartalma. A fájl első sora arra utal, hogy ez a fájl egy XML adatfolyamot tartalmaz. A második sor az “xsl” névtérrel vezeti be. A 3-7 sor azt mondja, hogy az egyes elemek megjelenítése előtt írjuk ki a az eredmény karaktorsorozatba a “<html><body><H1>VENDEGKÖNYV</H1>” füzért, majd alkalmazva a mintaleírásokat (apply-templates) kezdjük el feldolgozni az XML fa egyes elemeit. Az elem feldolgozások után a befejező lépés az, hogy a készítendő html dokumentumot is befejezzük, azaz kiírjuk a “</body></html>” karakterfüzért. A következő template szakaszok arról rendelkeznek, hogy mit kell csinálni a fa bejárása során, illetve akkor, amikor egy konkrét XML elemnél tart a feldolgozás. A <xsl:template match="VKONYV/VENDEG/NEV"> sorban a "VKONYV/VENDEG/NEV" kifejezést XPath-nak hívjuk. Az XSL vezérlő működését jobban átgondolva világossá válik, hogy az “applytemplates” kulcsszóval egy rekurzív definíciót tudunk megadni a feldolgozásra. A <xsl:value-of select="."/> mindig az aktuális faelem (node) értékét adja vissza.

Egy XML adatfolyam elemzése (Parse)

Amint azt már megtanultuk az XML dokumentumokhoz megadott nyelvtanok egyrészt egy új dokumentum létrehozásakor, másrészt egy létező dokumentum érvényességének ellenőrzéséhez szükségesek. Az XML dokumentumok elemzésére két szabványos megoldás terjedt el:

- DOM (Document Object Modell) alapú és

- SAX (Simple API for XML) alapú elemzők

A DOM felépíti az XML adatfolyamnak megfelelő, objektumokból álló fát, aminek az elemeit ezután közvetlenül el lehet érni (az objektumok metódusai és adattagjai használhatóak). A DOM fogalma ismerős lehet azoknak, akik a böngészőkben már írtak olyan JavaScript vagy Java programot, amik eléri a Window, Document, Link, ... objektumokat, hiszen az is egy DOM hierarchia. A SAX nem épít fel DOM szerkezetet, ugyanis itt az elemzés során mindig csak az aktuális elemet látjuk. A SAX emiatt nem teszi lehetővé az XML elemek közvetlen elérését, azokat csak sorosan olvassa fel, aminek következtében események generálódnak. Az elemző alkalmazás ezekre az eseményekre határozza meg az eseménykezelő metódusokat. A DOM és SAX API-k deklarációja a szabványos IDL nyelven van megfogalmazva, ezért azokat C++, Java, Pascal,... nyelveken implementálni kell. Az IBM, Oracle, Sun, Apache rendelkeznek Java-ban megvalósított implementációkkal. A szemléltető példákat az "Oracle XML Developer's Kit for Java" csomag használatával készítettük el, ami letölthető az Oracle TechNet helyről (itt C/C++ csomag is található). A két elemző működési elvében tapasztalható különbség miatt használják őket más, más célra. DOM elemzést általában kis méretű XML állományok feldolgozásánál használnak, mert a DOM funkciójából következőleg az egész dokumentumot beolvassa, és ez egy nagy állomány esetén jelentős memória terhelést jelent a rendszer szempontjából. Például webszervereknél amikor nagy XML-ből kell HTML kimenetet generálni esetleg párhuzamosan. Több felhasználó számára, ekkor a DOM nem hatékony ilyenkor SAX elemzést érdemes használni. XML alapú konfigurációs állományok feldolgozásánál viszont inkább a DOM az ami megfelelő támogatást nyújt.

Egy DOM elemző működése

Egy DOM elemző a VKONYV XML adatfolyamból az ábrán látható objektum hierarchiát épít fel a memóriában: A Java nyelv oldaláról vizsgálva ez azt jelenti, hogy XMLElement, XMLAttr, XMLNode típusú osztályok objektumai vesznek részt a DOM fa felépítésében. A szemléletesség érdekében tekintsünk egy egyszerű Java programot, ami elemzi a VKONYV XML adatfolyamot, illetve feldolgozási tevékenység gyanánt a Java konzolon egymás alatt megjeleníti a vendégek neveit. A programban megjegyzések magyarázzák el a feldolgozó algoritmus működését.

```
//
// Egy DOM elemző és feldolgozó
//
import java.net.*;
import org.w3c.dom.*;
import org.w3c.dom.Node;
import oracle.xml.parser.v2.*
//
public class TVendegLista
{
    //---
    // Main
    //
    public static void main(String[] args)
    {
        if ( args.length != 1 )
        {
            System.out.println("Használat: java TVendegLista xml_fájl");
        }
        try
        {
            // Az Elemzős egy referenciát ad vissza
            // a felépített DOM fára
            XMLDocument doc = Elemzes( args[0] );
            // Egy referenciát kapunk a DOM gyökérelemére
            Element gyokerElem = doc.getDocumentElement();
            // A felépített DOM fa feldolgozása
            DOMFaFeldolgozas( gyokerElem );
        }
    }
}
```

```

catch ( Exceptipon )
{
System.out.println("HIBA az elemzés során...");
}
// end main
//---
// Az elemzo rutin elkészítése
//
public XMLDocument Elemzes( String xml_fájl ) throws Exception
{
// Egy DOM parser objektum létrehozása
DOMParser parser = new DOMParser();
// Egy URL létrehozása
URL url = UrlFromFájl.createURL( xml_fájl );
// Elemzés, miközben felépül a DOM fa
parser.parse( url );
return parser.getDocument();
} // end Elemzes
//---
// A feldolgozás most azt jelenti, hogy a neveket
// listázni kell a Java konzolra
//
public void DOMFaFeldolgozas( Node EgyFapont )
{
if ( EgyFapont.getNodeType() == Node.ELEMENT_NODE )
{
if ( EgyFapont.getNodeName().equals("NEV") )
{
NevFeldolgozas((Element)EgyFapont);
}
}
else
{
NodeList gyerekek = EgyFapont.getChildNodes();
for (int i=0; i < gyerekek.getLength(); i++ )
{
DOMFaFeldolgozas(gyerekek.item(i)); // Rekurzió!
}
// end if
// end if
} // end DOMFaFeldolgozas
//---
// A NEV fapont feldolgozása
//
public void NevFeldolgozas( Element Fapont )
{
String nev = null;
Node gyerek = Fapont.getFirstChild();
Text text = (Text)gyerek;
nev = text.getData();
System.out.println( nev ); // Itt írjuk a konzolra
} // end NevFeldolgozas
} // end TVendegLista class

```

A “java TVendegLista vkonyv.xml” parancsra a program a konzolon ezt jeleníti meg:

```

X Y
Z V

```

Egy SAX elemző működése

A SAX elemzők sorosan végigolvassák a bemenő XML adatfolyamot, miközben különféle eseményeket generálnak. Az elemző, feldolgozó alkalmazások írása során a fő feladat a megfelelő eseménykezelő rutinok megírása és regisztrálása. A könnyebb érthetőség érdekében írunk olyan Java SAX elemzőt, ami kiírja a konzolra azt, hogy mennyi vendég van az XML fájlban tárolva.

```

//
// Egy SAX elemző és feldolgozó
//
import java.net.*;
import org.xml.sax.*;
import oracle.xml.parser.v2.*
//
//
// Az osztály leszármazottja az alapértelmezett

```



```

// SAX eseményeket kezelő osztálynak
//
public class TVendegListaSAX extends HandlerBase
{
    public int vendeg_cnt = 0; // Itt számoljuk a vendégeket
    //---
    // Ez egy eseménykezelő rutin, ami mindig az elemre lépéskor
    // automatikusan meghívódik
    //
    public void startElement( String name, AttributeList attrlist )
    {
        if ( name.equals("VENDEG") )
        {
            vendeg_cnt++;
        } // end startElement
    } //----
    // main
    //
    public static void main(String[] args)
    {
        if ( args.length != 1 )
        {
            System.out.println("Használat: java TVendegListaSAX xml_fájl");
        }
        // Az eseménykezelő osztály létrehozása
        TVendegListaSAX eseménykezelő = new TVendegListaSAX();
        // Egy SAX parser objektum létrehozása
        SAXParser parser = new SAXParser();
        // Az eseménykezelő regisztrálása a
        // most létrejött parser objektumnál
        parser.setDocumentHandler( eseménykezelő );
        //Kezdjük az elemzést!
        try
        {
            parser.parse( UrlFromFájl.createURL(args[0]));
        }
        catch (Exception)
        {
            System.out.println("HIBA az elemzés során!");
        }
        //--- Befejezésül kiírjuk az eredményt a konzolra
        System.out.println("A vendégek száma: " + Integer(vendeg_cnt).toString() );
    } // end main
} // end TVendegListaSAX class

```

A "java TVendegListaSAX vkonyv.xml" parancsra a program a konzolon ezt jeleníti meg:

A vendégek száma: 2

XML technológiák leírásának hivatalos helyei:

<http://www.w3c.org>

További érdekes XML technológiával is foglalkozó oldalak:

<http://www.xml.org>

<http://xml.apache.org>

Oracle TechNet:

<http://otn.oracle.com>

ASP – (Active Server Pages).

Egy kis alapozás

Amikor annak idején a HTML-t kitalálták, még senki sem gondolt arra, mi lesz a dolog vége. A HTML hipertext-leírónyelv eredetileg arra való, hogy segítségével egyszerű dokumentumokat hozzunk létre, amelyek egyes részei esetleg hivatkoznak más dokumentumok részeire (ez a hiperhivatkozás, hiperlink). Az eredeti HTML nyelv a hivatkozásokon kívül alig néhány elemet tartalmazott, amelyek különböző szintű címsorok, idézetek, esetleg listák létrehozását segítették. A sors fintora, hogy az Internet megjelenésével éppen a HTML lett az internetes kommunikáció

egyik alapja. (Nincs ezen mit csodálkozni: különféle adatok és közöttük felépített kapcsolatok leírására volt szükség, és a HTML éppen kapóra jött.) A ma használatos HTML persze már jócskán több, mint egyszerű dokumentumleíró nyelv – pontosan annyi köze van az “ős” HTML-hez, mint a mai dokumentumoknak az akkoriakhoz. Régen az adatok struktúrája képezte az alapot, ma inkább azok megjelenítése. Ahogy telt az idő, úgy szivárogtak bele a nyelvbe a tartalmat nem, azok megjelenítését annál inkább érintő elemek: képek, táblázatok, keretek (framek), színek, méretek, betűtípusok, külső objektumok, scriptrészletek és ki tudja még mi minden. A HTML 4-es változatát többek között pontosan azért alkották meg, hogy valamelyest (újra) szétválaszthassuk a tartalmat a megjelenítéstől, ezzel is csökkentve a HTML oldalak kódjában található nem kis káoszt. A tartalom és megjelenítés szétválasztása azóta szinte minden területen hódít, függetlenül attól, hogy a hálózaton található HTML oldalak nagy része a mai napig nem használja ki a HTML 4 lehetőségeit (köszönhető ez egyébként a szabványokkal többé-kevésbé hadilábon álló böngésző programoknak is).

Mozgásba lendül a kód...

Az idő múlásával egy másik területen is sokat fejlődött a HTML, illetve annak felhasználása. Kezdetben elég volt, ha egy dokumentumot létrehoztunk, annak tartalma nem, vagy csak ritkán változott. Később felmerült az igény arra, hogy a gyakran változó HTML oldalak tartalmát dinamikusan hozzák létre. Kezdetből két irányvonal létezett, attól függően, hogy a kiszolgálóra, vagy pedig a dokumentumot felhasználó ügyfélprogramra bízta a feladatot. Ez utóbbi megoldás nem biztos, hogy működik (senki sem garantálja, hogy az ügyfélprogram képes ezt a feladatot végrehajtani), ráadásul több szempontból előnytelen is: ha a cél az, hogy 100 dologból csak egy valami jelenjen meg az ügyfél képernyőjén, felesleges mind a százat elküldeni neki, majd ott kiválasztani a szükséges egyet – egyszerűbb, biztonságosabb és olcsóbb, ha már eleve csak a számára érdekes adatok kerülnek hozzá. Ehhez viszont a kiszolgálónak kell erőfeszítéseket tennie. A kiszolgálóoldali megoldások közös tulajdonsága, hogy a kiszolgáló mindig kész, feldolgozható HTML kódot küld az ügyfélnek – a kód tartalma viszont dinamikus, időről időre változik. Magyarán: a cél az, hogy HTML kódot generáljunk. A legkézenfekvőbb megoldás az volt, ha kész, teljes programokat írtak az egyes feladatok végrehajtására. A programok szabványos bemeneten (stdin) keresztül kapták a bemenő adatokat, majd a szabványos kimeneten (stdout) továbbították az általuk létrehozott kódot. A webkiszolgáló és a programok közötti kapcsolatot az ún. CGI (Common Gateway Interface) valósította meg, így a programok látszólag a kiszolgáló részeként működtek (és működnek ma is). Ezt a módszert CGI programozásnak nevezzük, és bár néhány területen még ma is használatos, hátrányai miatt lassan kiszorul. Mert: mit tehetünk, ha azt szeretnénk, hogy a CGI program mostantól más kódot generáljon? Egy: újraírjuk, újrafordítjuk és kicseréljük a programot. Kettő: Eleve olyan CGI alkalmazást írunk, ami a bemenő adatok segítségével paraméterezhető. Sőt, mi lenne, ha a bemenő paraméter egy valamilyen formában meghatározott parancssorozat, vagy akár egy valamilyen nyelven megírt script kód lenne? A CGI program azt értelmezi, végrehajtja, és visszaadja az eredményt – ez a megoldás a scriptnek köszönhetően teljesen dinamikus és bármire használható lenne – és az is. Jó példa erre a Perl nyelv, ahol a webprogramozás lelke a perl.exe. Bemenete egy Perl nyelven írt script, kimenete pedig a kész HTML kód.

ASP a láthatáron

A fenti megoldás egy kicsit mégis kényelmetlen: milyen jó lenne, ha a HTML oldalak általában statikus részét hagyományos módon, akár egy kényelmes WYSIWYG szerkesztővel

készíthetnénk, és csak a dinamikus részt kellene programozni! Az Active Server Pages (ASP) elve pontosan ez: amikor azt mondjuk, ASP, tulajdonképpen egy HTML kódba ágyazott, speciális programozási módszerről beszélünk. (Fontos, hogy az ASP nem egy programozási nyelv, hanem csak egy keretrendszer). Az ASP oldal végrehajtásakor a webkiszolgáló végigszalad az oldal tartalmán, és ha abban ASP scriptrészletet talál, végrehajtja. A HTML oldal és a script által esetleg visszaadott kódrészletek együttesen képezik az eredményt, amit azután az IIS elküld a böngészőnek. Lássunk egy példát:

```
<HTML><HEAD><TITLE></TITLE></HEAD>
<BODY>
<%
Response.Write("<center>Hello World!</center>")
%>
</BODY>
</HTML>
```

A HTML kód belsejében található `<%` és `%>` jelzi az ASP kód kezdetét és végét. A köztük található kódrészlet elvileg soha nem jut el az ügyfélhez, csakis a kód futtatása során keletkező kimenet (ami esetünkben a `Response.Write()` metódusnak átadott szövegrész). Az ASP scriptek beágyazásának módjáról kicsit később még lesz szó, most lássuk, mi lesz az eredmény:

```
<HTML><HEAD><TITLE></TITLE></HEAD>
<BODY>
<center>Hello World!</center>
</BODY>
</HTML>
```

Az ASP kód által generált kimenet tehát összemosódott a HTML kóddal. Ez jó, hiszen ráérünk az oldalt teljes egészében elkészíteni egy külső, kényelmes HTML szerkesztővel, majd utólag beágyazhatjuk az ASP kódot. Egy gondolat az ASP használatáról: mint látható, az ASP az IIS webkiszolgáló része. A Windows NT 4.0 Option Pack segítségével telepíthető Internet Information Server 4.0 (Windows NT 4.0 Workstation-ön Personal Web Server) már tartalmazza az ASP kezeléséhez szükséges komponenseket, amelyek a Windows 2000 IIS5 webkiszolgálójában természetesen alapértelmezett tartozékok. Ha azt szeretnénk, hogy egy fájl az IIS tényleg ASP oldalként kezeljen, adjunk a fájlunk `.asp` kiterjesztést (ha ezt nem tesszük, a kód végrehajtás nélkül eljut az ügyfélhez, mintha a HTML oldal tartalma lenne).

Az ASP kódok beágyazása

Az ASP scripte(ke)t az oldalba több módon is beágyazhatjuk. Lássuk mindenekelőtt a HTML szabványnak megfelelő módot:

```
<HTML><HEAD><TITLE></TITLE></HEAD>
<BODY>
<SCRIPT runat="server" language="vbscript">
Response.Write("<center>Hello World!</center>")
</SCRIPT>
</BODY>
</HTML>
```

A `SCRIPT` HTML elem segítségével tehát ugyanúgy ágyazhatunk be kiszolgálóoldalon futó kódot, mintha ügyféloldali scriptet írnánk – csak adjuk meg a `runat="server"` attribútumot. Hasonlóan az ügyféloldali megoldáshoz, természetesen kiszolgálóoldalon sem muszály az oldalon belül megírni a scriptet, megadhatunk fájlnevet is (`src` attribútum): `<SCRIPT runat="server" src="scfile.vbs">` Látható, hogy itt elhagytuk a scriptnyelv meghatározását. Ebben az esetben a kiszolgáló az alapértelmezett scriptnyelvet használja. Ezt két helyen határozhatjuk

meg: egyrészt, az adott .asp oldal tetejére írt, úgynevezett ASP direktíva segítségével : <%@ Language=VBScript %> Ha pedig ez hiányzik, a kiszolgáló a saját beállításait használja. A <% és %> használata rövidebb és kényelmesebb is, ezért továbbiakban – hacsak kifejezetten nincs szükség másra – ezt használjuk. Természetesen egy oldalon belül több scriptblokk is szerepelhet. Az ASP oldal tartalmát az IIS előlről hátrafelé haladva értékeli ki, beleértve magát a HTML kódot is. Az ASP kód által visszaadott kód az eredményben ott jelenik meg, ahol maga a script szerepel, például a :

```
<p>1<p><% Response.Write("a") %><p>2  
<% Response.Write("b") %><p>3
```

eredménye “1a2b3” és nem “ab123” vagy “123ab”. A fenti példában láthatjuk azt is, hogy akár soron belül is készíthetünk scriptblokkot (inline script), nem ritka az alábbihoz hasonló megoldás:

```
<INPUT type="text" value="<% =sTxt %>">
```

Ezután a szövegmezőben az sTxt változó tartalma jelenik meg. Újabb újdonsággal találkoztunk: a <% után írt = a Response.Write rövidítése, tehát a <%= "Hello!" %> egyenértékű a <%Response.Write("Hello!")%> sorral. Még egy fontos tudnivaló a több részletben beágyazott scriptekről: nem tilos az sem, hogy a script “közepén” egyszer csak tiszta HTML kód jelenjen meg. Ilyenkor az úgy viselkedik, mintha a kód része lenne, azaz ha az adott szakaszra rákerül a vezérlés, az is megjelenik, különben rejtve marad.

```
<% For i=1 To 10 %>  
<center>Hello World!</center>  
<% Next %>
```

A fentiek hatására például a Hello World! felirat tízszer íródik ki, az alábbi kódrészlet pedig a csillagok pillanatnyi állásától függően hol ezt, hol azt “mondja” (de sosem egyszerre a kettőt!):

```
<% Randomize ' <- Fontos, különben nem lenne ' véletlen! %>  
<% If Int(Rnd()*10) > 5 Then %>  
<center>Kököszi</center>  
<% Else %>  
<center>Bobosza</center>  
<% End If %>
```

Így nagyszerűen szegmentálhatjuk az oldalt. Ha például egy ASP oldalon több minden jelenhet meg, de nem egyidőben, akkor legjobb, ha HTML szerkesztővel létrehozuk az oldalt, benne az összes opcionális résszel, majd ezeket a részeket utólag “körbeépítjük” scripttel, ami majd eldönti, hogy az adott szakasz látsszon-e vagy sem. Bár a <% és a %> nem szerepelnek a HTML szabványban, mi bátran használjuk, hiszen ezek a jelek soha nem hagyják el a kiszolgálót. Ha az ASP script által generált kimenő kód HTML-kompatibilis, nyugodtak lehetünk abban, hogy mi minden tőlünk telhetőt megtettünk a szabványos kommunikáció érdekében. Ügyfél-kiszolgáló kommunikáció: a HTTP protokoll A HTML oldalak számítógépek közötti továbbításához ki kellett dolgozni egy adatátviteli szabványt. Ez lett a HTTP, azaz Hypertext Transfer Protocol. A HTTP nem más, mint egy jól definiált ügyfél-kiszolgáló kommunikáció.

A HTTP kommunikációt az ügyfél kezdeményezi: hálózati kapcsolatot létesít a kiszolgálóval és közli vele igényeit: ez a HTTP kérés (HTTP Request). A kérésre a kiszolgáló választ küld (HTTP Response), majd az eredeti definíció szerint megszakítja a kapcsolatot és szükség esetén minden kezdődik előlről. A kapcsolat megszakítására eredetileg azért volt szükség, hogy a fenntartott, használaton kívüli kapcsolatok ne terheljék feleslegesen a kiszolgálót. Manapság azonban más a helyzet: egy HTML oldalon képek, objektumok tömege lehet, így elvileg külön kapcsolatot kell felépíteni először a HTML kód, majd később minden egyes beágyazott kép és objektum letöltéséhez, és ma bizony éppen az újabb hálózati kapcsolatok létrehozása az, ami túlterhelheti a kiszolgálót (ráadásul ez nem is igazán hatékony). Ezért a HTTP 1.1 verziójában bevezették a

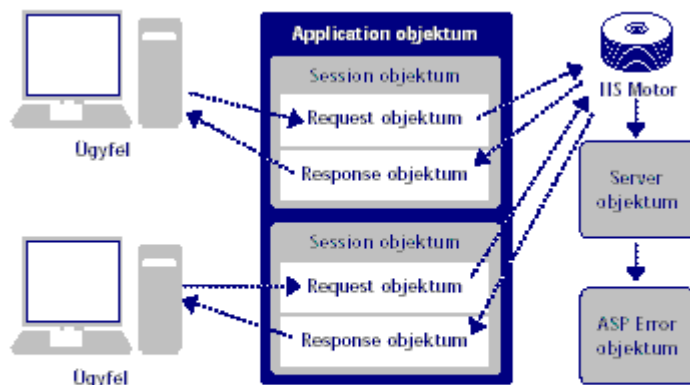
Keep-Alive opciót, amiben ha az ügyfél és a kiszolgáló megegyezik, a kapcsolat nem bomlik le azonnal (magyarul egy kapcsolat során több kérés és válasz is elhangozhat). Ha bárki hibát észlel, természetesen azonnal bontják a kapcsolatot. Ma már szinte minden böngésző így próbál kapcsolódni a kiszolgálóhoz, azok pedig általában elfogadják ezt a kérést. Alapértelmezésben így tesz az IIS is, erről a “HTTP Keep-Alives Enabled” opció kikapcsolásával beszélhetjük le (ezt az adott IIS web tulajdonságlapján, a “Web Site” oldalon találjuk). A HTTP kérés- és válaszüzenet egyaránt három fő részből áll:

- Parancs, illetve státuszinformáció
- Fejlécek, metaadatok
- HTTP tartalom

Az első részben (ami mindig az első sor) kéréskor a kért parancs, illetve annak paraméterei, valamint a verziószám utazik, válasz esetén pedig a státusz- vagy hibaüzenet kódja és leírása. Az ezután következő fejlécek a kapcsolat és a későbbi következő HTTP tartalom jellemzőit tartalmazzák, ezek vizsgálatára később részletesebben kitérünk, hiszen ami ügyfél-kiszolgáló kommunikáció és nem a HTML kód része, az itt található. Végül a HTTP tartalom, ami válasz esetén maga a HTML oldal, vagy mondjuk egy kép tartalma, kérés esetén pedig – ha van – a kiszolgálónak szánt bemenő adatok tömege. A fejléceket a HTTP tartalomtól egy üres sor választja el. Az ASP a HTML tartalom dinamikus létrehozása mellett természetesen lehetővé teszi a fejrészben kapott adatok feldolgozását és a válasz fejrészének manipulálását is.

Az ASP objektummodell

Az .asp oldalak programozását, a HTTP kommunikációt, a webkiszolgáló egyes szolgáltatásainak elérését külön objektummodell segíti. Az ASP objektummodelljének minden elemét elérhetjük az .asp oldalak kódjaiból. A későbbiek során mindegyik ASP objektumot bemutatjuk részletesen is, most vessünk egy röpke pillantást a teljes objektummodellre és annak elemeire.



Az ASP objektummodellje

Ha a tranzakciós műveletekhez használtObjectContext objektumot nem számítjuk, az objektummodell hat (Windows NT 4.0-n öt) objektumból áll. A Windows 2000-ben megjelent új objektum az ASPError, ami egy bekövetkezett hiba leírását tartalmazza, az .asp-be ágyazott, saját hibakezelést segíti. A Server objektum magát az IIS-t képviseli, néhány kiszolgálósintű beállítással és szolgáltatással. Az Application objektum egy webalkalmazást jelképez. A webalkalmazás különálló egység, általában egy könyvtárban és annak alkönyvtáraiban található

.asp kódok összessége, közös objektumokkal és beállításokkal. A Session objektum egy ügyfél és a kiszolgáló között “fennálló” kapcsolatot, munkamenetet jelképez. A “fennálló” kapcsolatot azért írtuk így, mert valójában nem egy kapcsolatról van szó. Az IIS (a háttérben cookie-k segítségével) azonosítja a felhasználót és a böngészőt, így az a böngésző bezárásáig saját munkamenetébe térhet vissza. A Request objektum egy HTTP kérést jelképez, segítségével kódból hozzáférhetünk a kérés minden eleméhez, legyen az HTTP fejléc értéke, a böngészőben tárolt cookie, vagy kérdőív tartama. A Response objektum pedig értelemszerűen a kérdésre küldendő választ jelképezi. Természetesen a Response objektum segítségével sem csak a válasz tartalmát, hanem a HTTP protokoll fejrészét is kezelhetjük. Egy IIS kiszolgálón belül Server és ASPError objektumból egy-egy létezik (utóbbi csak akkor érhető el, ha hiba történt). Application objektum minden webalkalmazás egyedi, globális objektuma, Session objektum minden munkamenethez (ügyfélhez) egy jön létre, egyidejűleg tehát több is létezhet. Request és Response objektum pedig mindig az adott kérésre és válaszra vonatkozik, újabb kérés esetén új példány jön létre belőlük is.

Egy HTTP válasz – a Response objektum

Kezdjük a végén: a Response objektummal, ami egy HTTP kérésre adott választ hivatott jelképezni. A Response objektum legegyszerűbb (és leggyakoribb) alkalmazását már láthattuk korábban: a Response.Write() metódus segítségével állítottuk elő az oldal tartalmát. A Write() használata egyszerű: minden, amit paraméterként átadunk neki, bekerül a válaszként visszaküldött adatsomagba. A Write() metódusról egyetlen dolgot kell tudni: a kiírt szöveg nem tartalmazhatja a %> jelsorozatot, helyette ezt kell írni: %\> Az ezt tartalmazó szöveget IIS automatikusan visszaalakítja majd az eredeti formára.

A válaszpuffer

Az .asp oldal előállítását természetesen több lépésben történik, az oldalban található scripttől függően előfordulhat az is, hogy az oldal tartalmának egy része csak bizonyos várakozási idő után áll rendelkezésre. Ilyenkor dönthetünk, hogy a már kész tartalmat elküldjük-e az ügyfélnek, majd várunk a folytatásra, vagy kivárjuk, amíg a teljes oldal elkészül, és csak a munka legvégén küldjük a komplett választ. Ez utóbbi esetben a “kimenet” egy pufferbe kerül. A pufferezés az IIS5-ben alapértelmezésben működik, míg az IIS4-ben alapértelmezésben ki van kapcsolva. A Response objektum segítségével mi magunk is kezelhetjük a puffert: mindenekelőtt, a Response.Buffer property-nek False értéket adva letilthatjuk, True segítségével pedig engedélyezhetjük a puffer használatát. A Clear() metódus kiüríti a puffert (Legyünk óvatosak! A “külső” HTML kód is a pufferbe kerül, törléskor az is elveszik!), a Flush() pedig elküldi azt, ami addig a pufferbe került, majd csak azután törli a tartalmát. Az oldal feldolgozását bármikor megszakíthatjuk a Response.End() meghívásával. Ha az oldal végrehajtása befejeződik, természetesen a puffer teljes tartalma az ügyfélhez kerül. Ha nyom nélkül szeretnénk befejezni a ténykedésünket, az End() meghívása előtt használjuk a Response.Clear() metódust.

HTTP fejlécek küldése

A HTTP válasz a tartalom mellett számos HTTP fejléccel is tartalmaz. Mi magunk is küldhetünk ilyen fejléccel a Response.AddHeader() metódus segítségével:

```
<%  
Response.AddHeader("MyHeader", "MyData")  
%>
```

A metódus két argumentuma a fejléc neve és értéke – természetesen a fenténél értelmesebb célra is felhasználhatjuk. Számos dolog van, ami közvetlenül programozható a Response objektumon keresztül és végső soron egy-egy HTTP fejléc elküldéséhez vezet (a Response.ContentType property beállítása pl. egy “Content-Type“ HTTP fejlécet küld, stb.), de előfordulhat, hogy olyasmit kell használnunk, ami nincs így “kivezetve“. A pufferek befolyásolják a HTTP fejlécek használatát, kikapcsolt puffer esetén HTTP fejlécet természetesen csak az oldal tartalma előtt küldhetünk, tehát a Response.AddHeader() metódus ekkor csak az .asp oldal elején (az ASP direktívák után) állhat. Fontos tudni, hogy egy fejléc már nem vonható vissza: amit egyszer létrehoztunk, az a válaszban már benne lesz, még akkor is, ha töröljük a válaszpuffert.

Tartalom és karakterkészlet

A HTTP válasz sokféle tartalmat hordozhat magában. Egyáltalán nem egyértelmű például, hogy egy HTML dokumentum milyen karakterkészlettel íródott. Sőt, még az sem biztos, hogy a válasz egy HTML oldal. Response.Charset = az oldalban használt karakterkészlet, kódtábla. Ha normális magyar betűket is használni szeretnénk, állítsuk “ISO-8859-2“-re. Természetesen ugyanezt HTML-ből, a <META> elem segítségével is megtehetjük, az alábbi két példa tehát egyenrangú (habár a fejlécben beállított karakterkészlet általában felülbírálja a HTML-ben meghatározottat – ez a böngészőn múlik):

```
<%  
Response.Charset = "ISO-8859-2"  
%>  
<HTML><HEAD>  
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-2">  
</HEAD>
```

...

Response.ContentType = a válasz MIME típusa (a tartalom típusa). HTML (és .asp) oldal esetén az értéke “text/html”, ha nem állítjuk be, ez az alapértelmezés is. Ennek a jellemzőnek az értékét akkor érdemes módosítani, ha a visszaküldött tartalom nem HTML, hanem mondjuk egy képfájl, mint alább:

```
<%  
Set oStream = Server.CreateObject("ADODB.Stream")  
oStream.Type = 1 ' adTypeBinary  
oStream.Open  
oStream.LoadFromFile(Server.MapPath("ms.jpg"))  
Response.ContentType = "image/jpeg"  
Response.BinaryWrite( oStream.Read )  
%>
```

A fent használt Response.BinaryWrite() metódus hasonló a .Write()-hoz, a különbség csak annyi, hogy ez binárisan, minden megkötés és konverzió nélkül írja a kimenetre az adatokat. A Server objektum MapPath() metódusáról később lesz szó, a lényege az, hogy egy relatív fájlnevből előállítja a fájl fizikai elérési útját a webkiszolgálón. Egy nálunk még elhanyagolt jellemző maradt utoljára: a Response.Pics jellemző értéke jelezheti, hogy egy adott oldal milyen mértékben és mennyiségben tartalmaz erőszakra, szexre, meztelenségre és csúnya nyelvre utaló “jeleket“. Ha ez az úgynevezett PICS-Label (ami egyébként nem más, mint egy speciális HTTP fejléc) utazik az oldallal, a szűrők képesek lennének kiválogatni a gyerekszobába nem való tartalmat. Ez az IIS beállításoknál egyébként webhelyenként, könyvtáranként, vagy akár fájlanként is beállítható. ASP-ben, a fenti beállítást kódból a következő módon érhetjük el:

```
<%
```

```
Response.Pics( "PICS-1.0 ""http://www.rsac.org/ratingsv01.html"" l by ""[mICK]"" on  
""2001. 01.08T21:26+0100"" exp ""2002.01.08T12:00 +0100"" r (v 3 s 0 n 0 l 0))" )%>
```

HTTP státusz és átirányítás

Response.Status = A HTTP válasz státuszüzenetét (ami, ha minden rendben ment, 200 OK) módosíthatjuk itt. A státuszüzeneteket a HTTP szabvány definiálja, és mivel a legtöbb funkció más módon is elérhető a Response objektumon keresztül, ezt a megoldást viszonylag ritkán használjuk. Hogy mégse maradjunk példa nélkül, lássunk egy átirányítást:

```
<%
```

```
Response.Status = "302 Object Moved" Response.AddHeader "Location",  
"http://www.microsoft.com"
```

```
%>
```

A fenti kódrészlet egyenrangú az alábbival:

```
<%
```

```
Response.Redirect("http://www.microsoft.com")
```

```
%>
```

A Response.Redirect() metódus tehát az ügyfél kérésének azonnali átirányítására való. A 302-es kódú HTTP üzenetnek (tehát az átirányításnak) egyetlen hátránya van: néhány proxy bizonyos körülmények között nem hajtja végre az automatikus átirányítást, hanem ronda “Object moved” hibaüzenetet küld vissza a böngészőbe. Ez akkor következhet be, ha az átirányítás mellett az adott oldal HTML tartalommal is bír (magyarul, ha a válaszban nem csak az átirányító fejlécek szerepelnek, hanem más is). Három megoldás kínálkozik:

- Redirect() előtt ürítsük ki a puffert (Clear() metódus), vagy eleve ne írjunk bele semmit (a Redirect() az olda elején szerepeljen)
- Ha kiszolgálón belül kell “átirányítani”, akkor használjuk inkább a Server objektum Transfer metódusát (IIS5-től), ami kiszolgálón belüli átirányítást végez anélkül, hogy az ügyfél erről tudomást szerezne
- Használjuk a HTML-ben gyakori átirányítási módszert, ilyenkor maga a böngésző kezd automatikus letöltésbe, ami ráadásul időzíthető:

```
<META http-equiv="refresh" content="0;URL=http://www.microsoft.com/">
```

A fenti példában az időzítés 0, azaz a böngésző azonnal belekezd az új cím letöltésébe. A <META> elemet a HTML oldal fejrészébe helyezzük el.

Ha sokáig tart az oldal előállítása

Ha egy oldal előállítása sokáig tart, vagy a webkiszolgáló túlterhelt, előfordulhat, hogy a kérés kiszolgálása közben (vagy előtt) az ügyfél megunja a várakozást és továbbáll. Az ilyenkor elvégzett munka kárba vész, ezért szükség esetén ellenőrizhetjük, nem hiába dolgozunk-e. Ha a Response.IsClientConnected property értéke hamis, akkor az ügyfél megunt a várakozást és bontotta a kapcsolatot, ha viszont igaz, érdemes még dolgozni. Természetesen felesleges minden sor előtt ellenőrizni, általában csak hosszú végrehajtási idejű oldalaknál van erre szükség, akkor is csak időközönként. Az IIS5 olyan komolyan veszi ezt, hogy minden kérés feldolgozása esetén ellenőrzi, hogy a kérés mennyi ideje érkezett. Ha a kiszolgáló túlterhelt, és a kérés több mint három másodperce várakozik, ellenőrzi, hogy megvan-e még az ügyfél, és csak akkor kezd bele a végrehajtásba, ha van kinek elküldeni a választ. Az IIS4 kicsit felemás módon viselkedik az .IsClientConnected jellemző kiértékelésekor. Ha az oldalunk ilyen kiszolgálón fut, tudnunk kell, hogy az .IsClientConnected csak akkor használható biztonságosan, ha az oldalból valamit már

elküldtünk az ügyfélnek (ha például a pufferelést bekapcsoltuk, csak a Flush() meghívása után számíthatunk helyes eredményre).

Gyorsítótárak – a cache

Mivel ugyanarra a tartalomra sokan, sokszor kíváncsiak, és nem szükséges a dolgokat újra és újra létrehozni és letölteni az eredeti származási helyükről, ezért használnak cacheeket. Gyorsítótárat alkalmaz a böngészőnk, saját gyorsítótárból dolgozik az Internet-szolgáltató, úgynevezett reverse-cache segíti a webkiszolgálókat a kérések gyors kiszolgálásában. Általánosságban elmondhatjuk, hogy aki tud, tartalékolja a dolgokat, hátha később még szükség lehet rá. Ezzel általában nincs is baj, de lehetnek dolgok, amelyeket felesleges elmenteni, mert a nevük, esetleg méretük hiába változatlan, tartalmuk gyakran eltérő. Kell-e jobb példa erre, mint a dinamikusan létrehozott weboldalak? Szerencsére a gyorsítótárak többsége távirányítható – a tartalom önmaga hordozhat olyan jeleket, amiket felismerve a gyorsítótár nem próbálkozik a tárolásával. Ilyen “jelek” (természetesen HTTP fejlécek) létrehozásában segít nekünk a Response objektum alábbi néhány szolgáltatása: Response.CacheControl = Az oldal tárolásának szabályai. Ha értéke “private”, akkor proxy kiszolgálók nem, csak és kizárólag privát, böngészőbeli gyorsítótárak tárolhatják az adatokat. Ez az alapértelmezés is. Ha a jellemző értéke “public”, akkor az oldalt bármelyik proxy kiszolgáló tárolhatja. Ha azt szeretnénk, hogy egyáltalán senki ne tárolja az oldalunkat, a property-nek adjuk ezt az értéket: “no-cache”. Ha tárolunk is valamit, időnként érdemes frissíteni. Minden tárolt tartalomnak van egy “lejárati ideje”, amit mi magunk állíthatunk be a Response.Expires és Response.ExpiresAbsolute jellemzők segítségével. Az előbbi az adott pillanattól számított lejárató időt jelenti (percben), míg az utóbbinak konkrét időpontot adhatunk meg, pl.:

```
<% Response.ExpiresAbsolute= #May 31,2001 13:30:15# %>
```

Az oldal tárolását elkerülhetjük úgy is, ha a lejárató időt -1-re állítjuk:

```
<% Response.Expires = -1 %>
```

Régebbi, a HTTP 1.1 szabvánnyal nem kompatibilis kiszolgálók nem értelmezik a CacheControl értékét, ezért néha speciális HTTP fejlécre van szükség:

```
<% Response.AddHeader "Pragma", "no-cache" %>
```

A legbiztonságosabb természetesen az, ha mindhárom módszert (CacheControl, Expires, Pragma) kombináljuk. Ha nem akarunk ASP-t használni, ezt a szokásos módon, <META> elemek segítségével tisztán HTML-ből is megtehetjük:

```
<META HTTP-EQUIV="CacheControl" CONTENT="no-cache">
```

```
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
```

```
<META HTTP-EQUIV="Expires" CONTENT="-1">
```

HTTP kérés – a Request objektum

Adatküldés a HTTP protokoll segítségével A dinamizmus, az interaktivitás egyik mozgatórugója természetesen az, ha menet közben adatokat kapunk az ügyfél oldaláról. Ezeket az adatokat a Request objektum segítségével érhetjük el. Adatok átadása az URL-ben (QueryString) A klasszikus adatbeviteli módszer, amikor az adatokat a kérés URL-jéhez csatoljuk, ilymódon:

```
http://localhost/qs.asp?input1=val1&input2=val2
```

Ekkor ilyen HTTP kérés indul a kiszolgáló felé:

```
GET /request.asp?input1=value1&input2=value2&submit=Submit HTTP/1.1
```

Ennek a módszernek több hátránya is van: egyrészt, a bemenő adatok növelik az URL hosszát, a kiszolgálónak elküldhető URL-ek mérete pedig biztonsági okokból általában korlátozva van. (Próbáljunk meg az IIS-nek elküldeni egy többszáz bájt hosszú címet! – A válasz szabványos

HTTP hibaüzenet: "404 – Request – URI too long"). Másrészt nemcsak kényelmetlen, de nem is igazán biztonságos, hogy az átadott adatok (amelyeket esetleg nem is mi írtunk be, hanem mondjuk egy kérdőív rejtett részei voltak) megjelennek a böngésző címsorában. Az átadott adatmezők név=adat formájúak (ld. fent: input1=val1), az egyes adatmezőket & jel választja el egymástól, az egészet pedig kérdőjel a fájlnévtől. Egy adott mező értékét a Request.QueryString("mezőnév") függvény adja vissza. Ha az adatok között ilyen mező nem szerepel, a visszaadott érték ("").

```
<%  
If Len( Request.QueryString("nev") ) Then  
Response.Write( "Szia " &  
f Request.QueryString("nev") & "!" )  
End If  
%>
```

Haz előző kódot elmentjük pl. qs.asp néven akkor, ha nevünket megadjuk az URL-ben (pl. qs.asp?nev=mick), akkor az oldal illendően köszönt minket. Egy mező "meglétét" a legegyszerűbben úgy ellenőrizhetjük, ha lekérdezzük a hosszát (ezt teszi a Len() függvény a példában). Ha ez nem 0, lehet dolgozni. Egy mezőnek azonban nem csak egy értéke lehet, a HTTP kérésben egy mezőnév egynél többször is szerepelhet: <http://localhost/qs.asp?nev=Piroska&nev=Farkas> Akkor ezt a következő módon lehet ezt feldolgozni:

```
<%  
Response.Write("Nevek száma: " & Request.QueryString("nev").Count & "<br>")  
For i=1 To Request.QueryString("nev").Count  
Response.Write( i & ": " &  
Request.QueryString("nev")(i) & "<br>")  
Next  
%>
```

A Request.QueryString("mezőnév").Count érték visszaadja az adott nevű mezők számát. Ha a sok közül egy konkrét értékre vagyunk kíváncsiak, akkor átadhatjuk az indexet is (a számozás 1-től kezdődik). Maradjunk a fenti példánál, a Farkas-ra így hivatkozhatunk:

```
Request.QueryString("nev")(2)
```

Ha egy mezőnek több értéke van, és mi mégis közvetlenül kérdezzük le (pl. Request.QueryString("nev")), akkor az értékek vesszővel elválasztott listáját kapjuk válaszként: "Piroska, Farkas". Ha pedig egyszerűen csak Request.QueryStringre hivatkozunk, akkor visszakapjuk a teljes kérdést: "nev=Piroska&nev=Farkas". Ez volt tehát az URL-be ágyazott lekérdezés feldolgozása. Egy fontos és sokszor zavaró tényezőre még szeretném felhívni a figyelmet: az URL-ek formátuma kötött, és mivel a lekérdezés (és főleg az átadott adatok) ilyenkor az URL részét képezik, ezeknek az adatoknak is meg kell felelniük bizonyos szabályoknak: például, minden írásjel, ékezetes karakter csakis kódolt formában (pl. egyenlőségjel: %3D) szerepelhet az URL-ben. Ez a kódolás pedig sokszor körülményes és kényelmetlen.

Adatfeltöltés a POST HTTP paranccsal

Szerencsére a HTTP protokoll tartalmaz egy, a fenténél fejlettebb megoldást is. A POST parancs használata esetén a feltöltendő adatok a HTTP üzenet törzsébe kerülnek. Az adatok kódolását persze így sem ússzuk meg, de az esetek többségében ezt a munkát nem mi, hanem a böngésző végzi. Kérdőív (Form) kitöltése esetén például, ha a FORM elem method attribútumát postra állítottuk, a következő kérés indul a kiszolgáló felé:

```
POST /request.asp HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 41
Connection: Keep-Alive
input1=value1&input2=value2&submit=Submit
```

Láthatjuk, hogy a kiszolgálónak szánt adatok nem a POST parancs paraméterében, hanem a HTTP üzenet törzsrészében utaznak. Érdekes megfigyelni a törzs kódolására vonatkozó Content-Type HTTP fejléc értékét is. Aki ismeri a HTML nyelvet, az tudhatja, hogy a HTML kérdőív elemének (FORM) method attribútuma határozza meg az adatok küldésének módját. Ha a method attribútum értéke "get", a kérdőív tartalmát az URLbe ágyazva, ha viszont az attribútum értéke "post", akkor a HTTP kérés törzsében küldi el a böngésző a kiszolgálónak. A kérdőív egyes mezőinek értékét a Request.Form kollekció tartalmazza. Az előző példához hasonlóan minden mező értékét lekérdezhetjük, ha a mezőnek több értéke van, akkor itt is használhatjuk a .Count jellemzőt és az indexeket is, pl.:

```
Request.Form("nev")
Request.Form("nev").Count
Request.Form("nev")(5)
```

Érdekesség: Kérdőív feldolgozása esetén a Submit nyomógomb értéke (felirata) is eljut a kiszolgálóhoz, de ha több ilyen van, akkor csak annak az egynek, amelyikre kattintottunk. Így megtehetjük azt, hogy ha egy kérdőívbe két Submit nyomógombot teszünk:

```
<INPUT type="submit" name="submit" value="Egyél">
<INPUT type="submit" name="submit" value="Igyál">
```

... a feldolgozáskor könnyen kitalálhatjuk, mit szeretne a kedves ügyfél:

```
<%
If Request.Form("submit") = "Egyél" Then
' Eszem
Else
' Iszom
End If
%>
```

A For Each utasítás segítségével (a többi kollekcióhoz hasonlóan) a .Form kollekció elemeit is kilistázhathatjuk:

```
<%
For Each mezo In Request.Form
' a mezo-be a mezónev kerül
Response.Write( "<b>" & mezo & " = </b>" & Request.Form(mezo) & "<br>" )
Next
%>
```

A HTTP tartalom kiolvasása

Nem kötelező a Request objektum kollekcióira támaszkodnunk, ha ki szeretnénk olvasni a HTTP kérés törzsében elküldött adatokat:

```
<%
bytes = Request.TotalBytes
data = Request.BinaryRead(bytes)
%>
```

A Request.TotalBytes jellemző visszaadja a törzsben található adatok méretét, a Request.BinaryRead() metódus pedig adott mennyiségű adatot olvas be, természetesen minden átalakítás és konverzió nélkül. Fontos! A Request.BinaryRead() metódus használata után már

nem használhatjuk a Request.Form kollekciót, és fordítva: ha a Request.Form-hoz már “hozzányúltunk”, a Request.BinaryRead() már hibát okoz.

Cookie

A cookie kis adatcsomag, amit a kiszolgáló kérésére a böngésző az ügyfél számítógépén tárol, és szükség esetén visszaküldi azt. Alapjában véve két különböző típusú cookie létezik: az első típus csak addig “él”, amíg a böngészővel egy kiszolgálónál tartózkodunk. A böngésző bezárásával az ilyen cookie tartalma elveszik. Ezeket a cookie-kat elsősorban átmeneti célra használjuk (például az ASP Session fenntartására). A másik fajta, felhasználói szemmel gyakrabban megfigyelt cookie annyiban különbözik az előzőtől, hogy a böngésző bezárásakor nem veszik el, hanem a számítógép lemezére kerül. Az ilyen cookie-knak érvényességi idejük van. Amíg ez az érvényességi idő le nem jár, addig a böngésző megőrzi az értéküket, és tartalmukat minden egyes látogatáskor visszaküldi a kiszolgálónak. A két cookie-fajtát csak a lejáratí idő különbözteti meg egymástól. Lássuk tehát, hogyan küldhetünk egyszerű, átmeneti célra használható cookie-t a böngészőnek:

```
<%  
Response.Cookies("nev") = "ertek"  
%>
```

Ha a felhasználó legközelebb felénk jár, ezt az adatot így olvashatjuk ki:

```
<%  
s = Request.Cookies("nev")  
%>
```

Egy cookie nem csak egy szöveget tartalmazhat, hanem többet is, egyfajta táblázatot, almezőket:

```
<%  
Response.Cookies("nev")("mezo1") = "ertek1"  
Response.Cookies("nev")("mezo2") = "ertek2"  
%>
```

A .HasKey metódus segítségével eldönthetjük, hogy egy cookie egyszerű szöveg, vagy almezők gyűjteménye, és ettől függően kezelhetjük is:

```
<%  
If Request.Cookies("nev").HasKey Then  
For Each mezo In Request.Cookies("nev")  
Response.Write( Request.Cookies("nev")(mezo))  
Next  
Else  
Response.Write( Request.Cookies("nev") )  
End If  
%>
```

A .HasKey elérhető a Response objektumon keresztül is. Erre azért van szükség, mert ha egy szöveges típusú cookieban almezőket hozunk létre, elveszik a szöveges érték – és fordítva, ha almezőkkel rendelkező cookie-nak szöveges értéket adnánk, elvesznének az almezők és azok értékei. A cookie érvényességét a Response objektum segítségével korlátozhatjuk időben és “térben”:

```
<%  
Response.Cookies("nev").Expires = "01-Jan-2003"  
Response.Cookies("nev").Domain = "erise.hu"  
Response.Cookies("nev").Path = "/dir1/dir2"  
%>
```

Az .Expires jellemző értéke határozza meg, hogy az cookie meddig marad életben. Ha nem adjuk meg, a böngésző lezárásakor elveszik. A .Domain jellemző segítségével beállíthatjuk, hogy a böngésző milyen domainek elérése esetén küldje vissza a cookie-t. A .Path pedig kiszolgálón belüli részletezést jelent: ha két azonos nevű cookie létezik, ugyanarra a domain-re, akkor annak az értékét fogjuk visszakapni, ahol a .Path értéke közelebb van a valósághoz. Ha tehát az oldal a /dir1/dir2 könyvtárak mélyén található, a két cookie .Path értéke pedig a /dir1 és a /dir1/dir2, akkor az utóbbit fogjuk viszontlátni. Ha azt szeretnénk, hogy egy cookie az adott pillanattól számított

egy évig legyen érvényes, használjuk a DateAdd() és a Now függvényt:

```
<%
```

```
Response.Cookies("egyevigjo").Expires = DateAdd( "yyyy", 1, Now)
```

```
%>
```

Az IIS naplójának írása

Ha nagyon akarunk írhatunk, de körültekintőnek kell lennünk. A Response.AppendToLog() metódusnak átadott szövegrész bekerül az IIS naplófájljába (tehát nem az Eseménynaplóba!). A szöveg nem tartalmazhat vesszőt, mert a naplófájl egyes mezőit vesszők választják el, és ez megzavarhatná a naplók későbbi feldolgozását. A bejegyzés csak akkor kerül be a naplóba, ha az IIS naplózás beállításai között bekattintottuk az “URI Query” mezőt. Bánjunk óvatosan ezzel a lehetőséggel. Ha az IIS naplója W3C vagy NCSA formátumban készül, az általunk átadott szöveg a naplóban a kért URL helyén (W3C formátum esetén), vagy ahhoz hozzáfűzve (NCSA) jelenik meg. Ennek nyilvánvalóan sok értelme nincsen, ezért javasolt a következő alternatíva, hogy az ilyen bejegyzéseket írjuk inkább szövegfájlba, vagy – ha mindenképpen ennél a megoldásnál akarunk maradni – használjuk a Microsoft IIS naplóformátumot.

Felhasználóazonosítás névvel és jelszóval

A webkiszolgáló tartalmának elérését sokszor korlátozni szeretnénk. Előfordulhat, hogy bizonyos adatokhoz való hozzáférés előtt szükség van a felhasználók azonosítására. A HTTP természetesen magában hordozza ennek lehetőségét is. Először is, a böngésző egy hagyományos kérést küld a kiszolgálónak. Amikor az alapértelmezett anonymous felhasználó (aki az IIS esetén egyébként megfelel az IUSR_számitógépnév felhasználónak) nem jogosult egy kért erőforrás elérésére, a kiszolgáló egy “401 Unauthorized” üzenettel válaszol:

```
HTTP/1.1 401 Unauthorized
```

```
Server: Microsoft-IIS/5.0
```

```
Date: Mon, 05 Feb 2001 21:05:25 GMT
```

```
WWW-Authenticate: Negotiate
```

```
WWW-Authenticate: NTLM
```

```
WWW-Authenticate: Basic realm="localhost"
```

```
Content-Length: 0
```

```
Content-Type: text/html
```

```
Cache-control: private
```

A lényeg mindenekelőtt, természetesen a 401-es kódú HTTP válaszüzenet, ami azt jelzi az ügyfélnek, hogy a hozzáférést megtagadtuk. A WWW-Authenticate HTTP fejlécek a lehetséges felhasználóazonosítási módszereket jelzik. Ezekből természetesen több is van, bár az Internet Explorer kivételével szinte mindegyik böngésző csak a Basic azonosítást ismeri. A Basic azonosítással viszont két nagy baj van:

1. A Basic felhasználóazonosítás során a jelszó kódolatlanul utazik a hálózaton mindaddig, amíg valamilyen kiegészítő megoldással (pl. https://) ezt át nem hidaljuk

2. Az IIS5 alapértelmezésben nem engedélyezi a Basic típusú azonosítást; ez természetesen azt jelenti, hogy az Internet Explorer-en kívül más böngészővel csak az anonymous által egyébként is elérhető oldalakhoz férhetünk hozzá.

Alapértelmezés, hogy felhasználóazonosításra akkor van szükség, ha az anonymous felhasználó hozzáférési jogai egy adott feladathoz már nem elegendők. Felhasználóazonosítást tehát legegyszerűbben úgy kényszeríthetünk ki, ha az NTFS fájlrendszerre telepített IIS könyvtára alatt (ez az \inetpub\wwwroot) a kívánt fájl(ko)n vagy könyvtár(ak)on korlátozzuk az IUSR_számitógépnév felhasználó hozzáférési jogait. Az IIS ilyenkor automatikus felhasználóazonosításba kezd, és csak akkor engedi “be” a felhasználót, ha őt a megadott jelszó segítségével a felhasználói adatbázisban sikeresen azonosította. Ha ezt nem akarjuk, az azonosítást kérhetjük kódból is: mint már tudjuk, a felhasználóazonosítást tulajdonképpen egy 401-es kódú HTTP válaszüzenet váltja ki. Vajon mi történik, ha a Response.Status segítségével mi magunk küldjük vissza ezt az üzenetet, valahogy így:

```
<% Response.Status = "401 Unauthorized" %>
```

A státuszüzenet mellé az IIS automatikusan mellékeli a megfelelő WWW-Authenticate mezőket és elvégzi helyettünk a felhasználóazonosítást. A felhasználó neve, jelszava (ha elérhető) és a felhasználóazonosítás típusa bármikor megtalálható a ServerVariables kollekcióban:

```
Request.ServerVariables("AUTH_TYPE")
```

```
Request.ServerVariables("AUTH_USER")
```

```
Request.ServerVariables("AUTH_PASSWORD")
```

A jelszó csak akkor látható, ha a típus “basic”; más esetekben az AUTH_PASSWORD mező értéke üres. Ha nem volt felhasználóazonosítás (pl. mert anonymous módon értük el az adott scriptet), az AUTH_USER értéke is üres lesz. Lássunk ezután egy példát, ami felhasználóazonosítást kér, majd ha az sikeres volt (azaz ha az IIS a Windows 2000/NT felhasználói adatbázisában a felhasználót megtalálta), kiírja a fenti adatokat:

```
<%
```

```
If Request.ServerVariables("AUTH_USER")="" Then Response.Status = "401 Unauthorized"
```

```
Response.End
```

```
End If
```

```
%>
```

```
<HTML>
```

```
<HEAD><TITLE>User LogOn Page</TITLE></HEAD>
```

```
<BODY>
```

```
AuthType: <% = Request.ServerVariables("AUTH_TYPE") %><BR>
```

```
Username: <% = Request.ServerVariables("AUTH_USER") %><BR>
```

```
Password: <% = Request.ServerVariables("AUTH_PASSWORD")%>
```

```
<BR>
```

```
</BODY>
```

```
</HTML>
```

A fenti kód első része ellenőrzi, hogy a felhasználó azonosította-e már magát. Ha nem (a felhasználónév üres), visszaküldi a státuszüzenetet, majd befejezi az oldal végrehajtását. Miután a felhasználó bejelentkezett, a vezérlés már eljut a valódi tartalomhoz: kiírjuk a felhasználóazonosítás típusát, a nevét és jelszavát. Érdeemes megfigyelni, hogy a különféle böngészők és azonosítási módszerek hogyan befolyásolják az adatokat: Basic azonosítás esetén például látható a jelszó, míg a Negotiate módon azonosított felhasználó nevében benne van a tartomány neve is (a \ jel előtti rész).

A Request kollekciók

A Request objektum lehetővé teszi, hogy a különböző kollekciókban található adatokat a forrás megadása (pl. Request.Form(“nev”)) nélkül, közvetlenül a Request(“nev”) hívással érjük el.

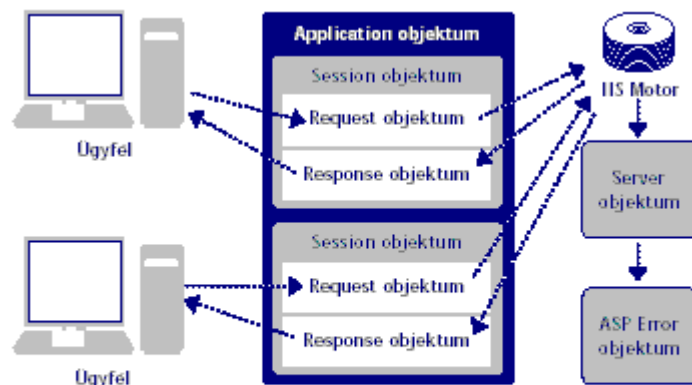
Ilyenkor az IIS az összes kollekción végigfut, és visszaadja az adott nevű elem első előfordulását. A sorrend a következő:

- QueryString
- Form
- Cookies
- ClientCertificate
- ServerVariables

Néha hasznos lehet, de általában nem jó, ha ezt a közvetlen hivatkozást használjuk: egyrészt, feleslegesen terheljük vele a kiszolgálót, másrészt pedig, nem lehetünk biztosak abban, hogy az adat onnan jött, ahonnan mi vártuk. Képzeljük el, hogy egy kérdőív “név” mezőjét szeretnénk beolvasni, ehelyett azt kapjuk, amit a felhasználó kézzel begépel az URL végére!

Az ASP alkalmazás és munkamenet

Lássuk csak újra az előző számban bemutatott ábrát az IIS objektumhierarchiájáról:



Az ASP objektummodellje

Jól látható, hogy az eddig tárgyalt objektumok, a Request és a Response mindig egy aktuális HTTP kérést és az arra adott választ jelképezik. A következő kérés esetén mindkét objektumból új keletkezik. A következőkben az alkalmazás (Application) és a munkamenet (Session) objektumokról lesz szó. Ezek az objektumok már nem tűnnek el ilyen egykönnyen az IIS memóriájából, hiszen feladatuk pontosan az, hogy több kérést is átfogó műveleteket, központi adattárolást tegyenek lehetővé.

Az ASP munkamenet

Az ASP munkamenet (Session) célja teljesen hasonló, csak hogy ez az Application-nel ellentétben nem felhasználók közötti, hanem egy adott felhasználó műveletei fölötti globális objektum. Ha úgy tetszik, számos Request és Response fölött uralkodó valami, ami megmarad egészen addig, amíg a felhasználó el nem hagyja a webhelyet, vagy be nem csukja a böngészőjét. Természetesen Session objektumból már nem csak egy van: ahány felhasználó, annyi Session. Az IIS egy kis cookiet küld minden felhasználónak, akit azután felismer mindaddig, amíg a cookie megmarad (márpedig az csak addig marad meg, amíg a böngészőt be nem zárjuk). Nézzük csak meg, mit mond az IIS egy teljesen átlagos kérésre:

GET /default.asp HTTP/1.1

...

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Mon, 12 Feb 2001 22:21:53 GMT
Content-Length: 2354
Content-Type: text/html
Set-Cookie:ASPSESSIONIDGQGQGVVDQ=IBGLAICAJDOELPGDLNDPODPM;path=/Cache-control: private

...

Kapunk egy érdekes nevű cookie-t: az (egyébként változó) ASPSESSIONIDxxxxxxx nevű cookie tartalmának segítségével az IIS egyértelműen azonosítja a visszatérő böngészőt, és ugyanabba a környezetbe helyezi (azaz, mindenki az első látogatáskor részére létrehozott, különbejáratú Session objektumba pottyan). Természetesen kell, hogy legyen egy bizonyos időkorlát is: ha valaki 20 percen belül nem jelentkezik, a részére létrehozott Session objektum elveszik, és legközelebb újra tiszta lappal indul. Ez az időkorlát is beállítható. És akár oldalanként is kikapcsolhatjuk, az oldal tetejére írt ASP direktíva segítségével:

```
<%@ENABLESESSIONSTATE="False"%>
```

A Session objektum

A Session.SessionID jellemző visszaadja a Session azonosítóját. Ez az azonosító sokszor jó szolgálatot tehet, hiszen segítségével azonosítani lehet a visszatérő felhasználót. (Mielőtt valaki félreértene: visszatérő alatt most azt értjük, aki a Session időkorlát lejártá előtt “visszatér”, vagy azalatt barangol oldalainkon). A Session.LCID és a Session.CodePage jellemzők a szokásos locale és karaktertábla-azonosítók; az objektum negyedik, s egyben utolsó jellemzője pedig a Session.Timeout, amit állítva egy adott munkamenet erejéig felülbírállhatjuk az alapértelmezett 20 perces időkorlátot. Sokkal érdekesebb ennél az, amire a Session objektumot eredetileg kitalálták: írhatunk bele és olvashatunk belőle, gyakorlatilag bármit; a beleírt érték pedig természetesen megmarad mindaddig, amíg az objektum életben van. Lássuk, hogyan tárolhatunk el egy értéket, hogy azután később felhasználhassuk:

```
Session("counter") = 1
```

```
Session("counter") = Session("counter") + 1
```

```
x = Session("counter")
```

A Session.Contents egy kollekció, aminek segítségével visszanyerhetjük mindazt, amit sikerült az objektumba belapátolnunk (anélkül, hogy tudnánk a nevét), valahogy így:

```
For Each oItem In Session.Contents
```

```
Response.Write(Session(" & oItem & ") = " & Session(oItem) & "<br>" )
```

```
Next
```

Ebből a kollekcióból persze törölni is lehet. Az alábbi példa első két sora egy-egy elemet (pl. a “counter” nevűt, vagy éppen a másodikat), míg a harmadik a teljes tartalmat törli:

```
Session.Contents.Remove("counter")
```

```
Session.Contents.Remove(2)
```

```
Session.Contents.RemoveAll()
```

A tartalom tehát ilyenkor elveszik, de az objektum megmarad. Azért fontos ezt megemlíteni, mert a Session objektumot önmagát is lehet törölni:

```
Session.Abandon()
```

A Session.Abandon() hívás után (természetesen miután az adott oldalt már végleg kiküldtük) a Session objektum törlődik a memóriából. A felhasználó legközelebbi jelentkezésekor új, üres objektum jön majd létre. Természetesen nem csak számot és szöveget tárolhatunk a Session-ben, hanem akár komplett objektumokat is. Valahogy így:

```
<%
```



```

Set oStream = Server.CreateObject("ADODB.Stream")
oStream.Type = 1 ' adTypeBinary
oStream.Open
oStream.LoadFromFile( Server.MapPath("ms.jpg") )
Set Session("mypic") = oStream
%>

```

A kiolvassó kód, a mi kiolvassa képet és elküldi a böngészőnek:

```

<%
If IsObject( Session("mypic") ) Then
Set oStream = Session("mypic")
Response.ContentType = "image/jpeg"
Response.BinaryWrite( oStream.Read )
Else
%>
<HTML><HEAD></HEAD>
<BODY>A kép nem található.</BODY>
</HTML>
<%
End If
%>

```

A kód elején található ellenőrzés azért kell, mert ha a Session("mypic") nem tartalmazza még az objektumot, akkor az értékadás azonnal hibát jelezne. Az IsObject() függvény értéke akkor igaz, ha a neki paraméterként átadott változó tényleg egy objektum – ha nem az, a kép helyett egy egyszerű HTML hibaüzenetet küldünk vissza. Figyeljük meg, hogy az objektumok kezelésénél – a más típusú változókkal ellentétben – használnunk kell a Set utasítást:

```

Set oStream = Server.CreateObject("ADODB.Stream")
Set Session("mypic") = oStream
Set oMyObj = Session("mypic")

```

A global.asa fájl

A global.asa fájl tulajdonképpen az Application objektum leírásánál kellene, hogy előkerüljön. A fájl azonban tartalmazhat a Session objektumra vonatkozó részeket is, ezért a tárgyalásától nem tekinthetünk el. A global.asa fájl egy speciális állomány, amit az ASP alkalmazás gyökérfkönyvtárában lehet elhelyezni (az alapértelmezett ASP alkalmazás gyökérfkönyvtára például természetesen az \inetpub\wwwroot). A fájl arra való, hogy ebben helyezhessük el a globális objektumokat létrehozó és –eseményeket kezelő kódrészleteket, úgyismint:

- Az Application objektum eseményeit (létrehozását, megsemmisítését) kezelő rutinok (Application_OnStart és Application_OnEnd)
- A Session objektumok létrehozását és megsemmisítését kezelő eljárások (Session_OnStart és Session_OnEnd)
- Globális objektumok létrehozása az <OBJECT> elem segítségével
- Típuskönyvtárak (type libraries) betöltése

A Session_OnStart szubrutin értelemszerűen az adott Session létrejöttekor, a Session_OnEnd pedig az objektum megsemmisülése előtt fut le. Ezeket a rutinokat a global.asa fájlba <SCRIPT></SCRIPT> elemek közé kell megírunk, például:

```

<SCRIPT LANGUAGE=VBScript RUNAT=Server>
Sub Session_OnStart
Session("starttime") = Now
Set Session("oFSO") = Server.CreateObject("Scripting.FileSystemObject")
End Sub

```

```

Sub Session_OnEnd
Set Session("oFSO") = Nothing
End Sub
</SCRIPT>

```

A fenti példában a Session létrejöttkor beleírjuk a pontos időt, és létrehozunk egy FileSystemObject objektumot, amit a továbbiak során majd kényelmesen használhatunk, a Session_OnEnd szubrutinban pedig felszabadítjuk az FileSystemObject objektum által lefoglalt memóriaterületet. Objektumokat globálisan is létrehozhatunk, az <OBJECT> elem segítségével, például:

```

<OBJECT RUNAT=Server SCOPE=Session ID="oGlobFSO"
PROGID="Scripting.FileSystemObject">
<SCRIPT LANGUAGE=VBScript RUNAT=Server>
...
</SCRIPT>

```

Az így létrehozott objektumokra azután alkalmazásszerte a .asp oldalakban közvetlen nevükkel hivatkozhatunk:

```

<%
If oGlobFSO.FileExists(strFileName) Then
...
End If
%>

```

Az <OBJECT> elem SCOPE paramétere határozza meg, hogy az objektum hány példányban jöjjön létre. A paraméter értéke esetünkben természetesen "session", ami azt jelenti, hogy minden egyes Session létrejöttkor újabb FileSystemObject keletkezik. A StaticObjects kollekció Egyetlen kollekció maradt a végére: a Session.StaticObjects kollekció a global.asa-ban "session" scope-pal létrehozott objektumokat tartalmazza. A kollekció tartalmát a szokásos módon érhetjük el:

```

For Each oItem In Session.StaticObjects
Response.Write( "Session(" & oItem & ") = " & Session(oItem) & "<br>" )
Next

```

Az ASP alkalmazás

Egy ASP alkalmazás tulajdonképpen nem más, mint egy adott könyvtárban és az összes alkönyvtárban található .asp kódok halmaza. Csakhogy a valóságban ennél kicsit bonyolultabb a helyzet, hiszen ha csak erről lenne szó, nem lett volna szükség az alkalmazások létrehozására. Mint tudjuk, a HTTP eredetileg állapotmentes világ: jön egy kérés, mi kiszolgáljuk, azután jön a következő, és a kiszolgálónak végülis fogalma sincs arról, hogy melyik kérést éppen ki küldte. Lehet, hogy ugyanaz a felhasználó, lehet, hogy a világ két végéről két olvasó jelentkezett. Mégis, milyen jó lenne, ha lenne egy globális "valami", amiben adatokat, sőt, akár kész objektumokat tárolhatunk, és bármikor szükség van rá, csak "fel" kell nyúlnunk érte, és hopp, máris a rendelkezésünkre áll! Nos, pontosan erre való az Application objektum. Segítségével a felhasználók, kérések között adatokat oszthatunk meg egyszerűen, anélkül, hogy például lemeze kellene írunk azokat. Az Application objektumba írt információ mindaddig megmarad, amíg az adott alkalmazást (gyakorlatilag az IIS-t) újra nem indítjuk. Amint az ábrán is látható, az ASP alkalmazás egy globális objektum a felhasználók kérései és az azokra adott válaszok fölött. Fontos, hogy Application objektum minden ASP alkalmazásban csak egy van.

Egy Applicationhoz választható futtatási jogosultságok:

- None: Csak statikus működés engedélyezett. Scriptek és programok nem futtathatók. Ezt használjuk akkor, ha nem használunk aktív komponenseket

- Scripts only: Scriptfájlok végrehajtása engedélyezett, de különálló programok továbbra sem futtathatók. Ez az alapértelmezett és ajánlott beállítás
- Scripts and Executables: A legveszélyesebb, CGI-kompatibilitás miatt megtartott opció. Ilyenkor a scriptek mellett a könyvtárakban található programok is lefuthatnak.

Az alkalmazás védelme

A meghatározás tulajdonképpen hibás, ugyanis nem a webalkalmazásunkat, hanem magát az IIS-t védjük a saját a többi webalkalmazástól). Hogy miért kell ez a védekezés? A webalkalmazások saját útjaikat járják. Az esetek többségében (például .dll fájlokban megírt) külső komponenseket használnak, hibátlan program pedig, mint tudjuk, nincsen. A .dll-ek közismert tulajdonsága, hogy a szülőprocessz címtartományába töltődnek be, és ha a .dll készül elhalálozni, a szülőprocesszt is magával rántja. Így lehet egyetlen .dll segítségével a nullával egyenlővé tenni a webkiszolgálót. Pontosan az ilyen esetek elkerülésére találták ki az "alkalmazások védelme" ("Application Protection") opciót. Egy (.dll-eket töltögető) webalkalmazás (kvázi maga a .dll) három módon töltődhet be:

- (Low): az IIS processzbe. Ez gyors, de veszélyes működést eredményez, hiszen ha a .dll kipukkan, megy vele az IIS is.
- (Medium – Pooled): egy, az IIS processztől elválasztva futó processzbe. A processz neve dllhost.exe, megleshetjük a Task Managerben. Minden Medium szintű webalkalmazást ugyanaz a dllhost.exe futtat, tehát ha a több Medium szintű alkalmazásból egy elhasal, magával rántja a többit is – az IIS viszont talpon marad. Ez az alapértelmezés, mert viszonylag biztonságos, és nem igényel sok erőforrást.
- (High – Isolated) A High szintre helyezett webalkalmazások mindegyike saját dllhost.exe-t kap, amibe annyira és annyszor rúghat bele, ahányszor csak akar, önmagán kívül senkinek sem árthat vele. Akit érdekel, kipróbálhatja: nyisson meg egy Task Manager-t, állítsa a futó processzeket név szerint sorba, és meglátja, hogy n darab dllhost.exe fut. Majd állítson egy webalkalmazást High szintre, és nyisson meg egy oldalt belőle: a futó dllhost.exe-k száma $n+1$ -re nő (az Unload hatására pedig értelemszerűen eggyel csökken). Ez a legbiztonságosabb megoldás, de a gyakori kontextusváltás miatt sok erőforrást igényel, ezért a Microsoft nem ajánlja, hogy egy kiszolgálóra 2-3-nál több ilyen szintű alkalmazás kerüljön.

Az Application objektum

Az Application objektum tehát nem más, mint egy közös tárolóhely, amit az adott webalkalmazás minden scriptje elér. Az Application objektumban adatokat, objektumokat tárolhatunk el, hogy majd később kiolvassuk onnan (csakúgy, mint a Session objektumnál):

```
Application("counter") = 12
```

```
Set Application("myObject") = oMyObject
```

```
Set oMyObject2 = Application("myObject")
```

Amint az a fenti példában is látható, az Application objektumban is tárolhatunk más objektumokat. Nem szabad elfelejtkezni azonban két nagyon fontos dologról:

- Az objektumokat Set értékadás segítségével kell az Application objektumba betölteni, és a kiolvasásuk is a Set parancs segítségével történik.
- Nem minden objektum alkalmas arra, hogy az Application objektumba töltsük! Az IIS ugyanis többszálú alkalmazás, egyidőben több felhasználót szolgál ki. Az Application objektumba csak olyan objektumokat tölthetünk be, amelyek többszálú (FreeThreaded) végrehajtási módban képesek működni és úgynevezett Both threading-modellt alkalmaznak.

Az alábbi példa az Application objektumot használja egy primitív számláló létrehozásához:

```
<%  
Application.Lock  
Application("count") = Application("count") + 1  
Application.Unlock  
Response.Write("Üdv! Te vagy az IIS újraindítása óta a(z) " & Application("counter") &  
látogató.")  
%>
```

A példa a valóságban azért használhatatlan, mert a webalkalmazás újraindulásának pillanatában – reboot vagy unload esetén – az Application objektum tartalma, ezzel pedig a számláló értéke is elveszik. Az első és a harmadik sorban látható Application.Lock() és Application.Unlock() metódus használatára azért van szükség, mert az Application globális objektum, és előfordulhat, hogy egyszerre több felhasználó szeretné írni ugyanazokat az adatokat. Az ütközések elkerülése végett minden írásművelet előtt az objektumot zárolni kell, majd a lehető legrövidebb időn belül az Unlock() metódus segítségével fel kell oldani, ugyanis míg az Application objektum zárolva van, senki más nem férhet hozzá, mint az, aki eredetileg zárolta azt. Ha mi nem oldanánk fel a zárolást, az IIS az oldal végrehajtása után, de legkésőbb a scriptfuttatás időtúllépésekor felszabadítja azt. Természetesen az Application objektum tartalmát is elérhetjük kollekciókon keresztül, erre szolgál az Application.Contents kollekció:

```
For Each oItem In Application.Contents  
Response.Write(Application(" & oItem & ") = " &Application(oItem) & "<br>" )  
Next
```

Ebből a kollekcióból is pontosan ugyanúgy lehet törölni elemeket, mint a Session-ből:

```
Application.Contents.Remove("counter")  
Application.Contents.Remove(2)  
Application.Contents.RemoveAll()
```

Az első sor a "counter" értékét, a második sor az Application objektumon belüli második változót, míg a legutolsó sor az Application objektum teljes tartalmát törölte. Az Application.StaticObjects kollekció a global.asa fájlban az <OBJECT> elem segítségével, "Application" scope-ban létrehozott változókat tartalmazza. A kollekció pontosan úgy használható, mint az Application.Contents:

```
For Each oItem In Application.StaticObjects  
Response.Write(Application(" & oItem & ") = " &Application(oItem) & "<br>" )  
Next
```

Az egyetlen különbség az, hogy ebből a kollekcióból nem törölhetünk elemeket (azok elvesznek maguktól a webalkalmazás újraindításakor).

A global.asa fájl mégegyszer

Mint azt már a Session objektum ismertetésekor röviden leírtam, a global.asa fájl egy speciális állomány, amit az ASP alkalmazás gyökérkönyvtárában kell elhelyezni (de a használata nem kötelező). A fájl arra való, hogy ebben helyezzük el a globális objektumokat létrehozó és eseményeket kezelő kódrészleteket, úgymint:

- az Application objektum eseményeit (létrehozását, megsemmisítését) kezelő rutinok (Application_OnStart és Application_OnEnd)
- a Session objektumok létrehozását és megsemmisítését kezelő eljárások (Session_OnStart és Session_OnEnd)
- globális objektumok létrehozása az <OBJECT> elem segítségével
- típuskönyvtárak (type libraries) betöltése

A global.asa fájlban található rutinokat <SCRIPT></SCRIPT> elemek közé kell elhelyezni. A statikus objektumok létrehozására szolgáló <OBJECT> elemet a <SCRIPT> blokkon kívülre kell elhelyezni, míg a típuskönyvtár-definíciók a blokkon kívülre és belülre egyaránt kerülhetnek, de érdemes azt is már a fájl elején, a <SCRIPT> blokkon kívül letudni. Ha a global.asa fájl tartalma megváltozik, az IIS minden már megkezdett kapcsolatot kiszolgál, és csak azok lezárása után tölti be az új változatot. Ez természetesen az összes Session és az Application objektum lezárását és újbóli megnyitását is jelenti. A global.asa betöltése és feldolgozása során az IIS nem fogad új kapcsolatokat, a felhasználók ezidő alatt csinos kis hibaüzenettel találkoznak ("a kérés nem szolgálható ki, az alkalmazás újraindul"). Események a global.asa-ban A global.asa-ban négy különféle eseményt kezelő rutint definiálhatunk, ezek az Application_OnStart, Application_OnEnd, Session_OnStart, és a Session_OnEnd. Lássunk egy mintát:

```
<SCRIPT LANGUAGE='VBScript' RUNAT='Server'>
Sub Application_OnStart
Application("appstarttime") = Now
End Sub
Sub Application_OnEnd
' ...
End Sub
Sub Session_OnStart
Session("starttime") = Now
Set Session("oFS") = Server.CreateObject("Scripting.FileSystemObject")
End Sub
Sub Session_OnEnd
Set Session("oFS") = Nothing
End Sub
</SCRIPT>
```

- Application_OnStart: a webalkalmazás indulásakor, egyszeri alkalommal fut le.
- Application_OnEnd: az alkalmazás leállításakor fut le, az alkalmazás életében ugyancsak egyszer
- Session_OnStart: Session objektum létrehozásakor, gyakorlatilag minden új felhasználó belépésekor hajtódik végre
- Session_OnEnd: ez pedig a Session lezárásakor lép működésbe. (például ha időtúllépés vagy Session.Abandon() hívása miatt az IIS megszünteti a session-t)

Írunk az Application objektumba, és mégsem zároljuk előtte, ugyanis ez az esemény egyszer és csakis egyszer következik be, mégpedig az alkalmazás élettartama legelején, amikor másnak még esélye sincs az Application objektumhoz hozzáférni. Ez az egyetlen hely, ahol nem kötelező használni a Lock/Unlock metódust.

Objektumok és típuskönyvtárak

Az objektumok létrehozásának módját a Session leírásánál már megismertük:

```
<OBJECT RUNAT=Server SCOPE=Application ID="oGFSO"
PROGID="Scripting.FileSystemObject">
```

A különbség csak annyi, hogy a scope (futási, létrehozási környezet) most nem Session, hanem Application lesz. Ezután az oGFSO objektumot a webalkalmazás minden scriptjében közvetlen hivatkozással elérhetjük, és az megjelenik az Application.StaticObjects kollekcióban is. Nagyon

kell ügyelni arra, hogy az Application objektumban csak Both, free threaded objektumokat tároljunk (ha <OBJECT> segítségével szeretnénk rossz objektumot létrehozni, az IIS hibát jelez, a dinamikus létrehozásnál viszont nem, ott csak később, a fura működés során derülhet fény a problémára). A COM objektumok általában típuskönyvtárakat is tartalmaznak. Ezekből a típuskönyvtárakból lehet kiolvasni az objektum metódusainak, jellemzőinek a listáját, de sokszor mindenféle konstansokat is. Ha például egy fájlt meg szeretnénk nyitni olvasásra a FileSystemObject segítségével, a következőt kell írunk:

```
Set oFSO = Server.CreateObject("Scripting.FileSystemObject")
```

```
Set oFile = oFSO.OpenTextFile("file.txt", 1)
```

Ehhez tudnunk kell, hogy az OpenTextFile metódus második paraméterének 1 értéke az olvasást jelenti (ForReading). Ha azonban a global.asa tetején megadjuk a következő definíciót:

```
<!--METADATA TYPE="TypeLib" FILE="scrrun.dll"-->
```

(az scrrun.dll tartalmazza többek között a FileSystemObject objektumot), akkor így is írhatjuk::

```
Set oFSO = Server.CreateObject("Scripting.FileSystemObject")
```

```
Set oFile = oFSO.OpenTextFile("file.txt", ForReading)
```

A számérték helyett tehát használhatók a típuskönyvtárban definiált konstansok, amelyek sokszor (nagyon sokszor) megkönnyítik az ember munkáját.

A Server objektum

A Server objektum is egy és oszthatatlan, és főleg kényelmi szolgáltatásai miatt hasznos. Gondoltunk-e már például arra, hogy ezt írjuk ki a felhasználó böngészőjébe: “Vízszintes vonal: <HR>”. Ha ezt elküldjük a böngészőbe, megjelenik a felirat, majd maga a vízszintes vonal, hiszen a böngésző értelmezi a kódban található HTML tagot. Ha a HTML elemet magát szeretnénk megjeleníteni, a < jelet < a >-t pedig > entity-vel kell helyettesítenünk, tehát valahogy így: “Vízszintes vonal: <HR>”. Ezen kívül még sok más elemet is kódolni kell, nem beszélve a speciális karakterekről. Szerencsére itt van a Server.HTMLEncode() metódus, ami elvégzi ezt a kódolást:

```
Response.Write( Server.HTMLEncode("Nesze <HR>") )
```

A HTMLEncode() azért is nagyon fontos, mert okos használatával elkerülhetjük az úgynevezett Cross Site Scripting hibát. Ezt a fontos biztonsági hibát az okozza, hogy egyes weboldalak (akár hibaüzenet formájában is) válogatás nélkül, és kódolatlanul visszaküldenek bizonyos, részükre előzőleg átadott szövegrészeket.

A másik hasznos kódoló metódus a Server.URLEncode(). Mint azt talán mindenki tudja, az URL-ekben nem szerepelhet akármilyen karakter. Ami nem megszokott, azt kódolni kell, általában %xx formában, ahol xx a karakter hexadecimális kódja. (Még a szóközt is helyettesítik, + jellel). Ha URL-eket “építünk” fel az ASP oldalainkban, mindig használjuk ezt a metódust is.

A Server.MapPath() nagyon gyakran használt metódus. Arra való, hogy a virtuális könyvtár- és fájlneveket (pl. http://localhost/asp4/default.asp) valós fájlnevekké alakítsa (pl. C:\inetpub\wwwroot\asp4\default.asp). Ha a MapPath()-nek átadott virtuális név “/” vagy “\” karakterrel kezdődik, akkor a virtuális gyökérkönyvtártól, ellenkező esetben pedig a hívás helyétől relatív fájlnevekké dolgozik (nem lesz ugyanaz az eredménye tehát a “default.asp” és a “/default.asp” kódolásának – kivéve ha éppen a gyökérkönyvtárban “állunk”). Természetesen használhatók a “.” és “..” karakterek, azaz mozoghatunk a virtuális könyvtárak által alkotott térben (a “.” az aktuális, a “..” pedig a virtuális szülőkönyvtárra mutat). A Server.MapPath() metódust nem használhatjuk a global.asa Application_OnEnd eseményének kezelése közben, egyébként pedig legyünk óvatosak, mert a global.asa-ban meghívott MapPath() nem a global.asa, hanem a felhasználó által megnyitni kívánt fájl alapján dolgozik.

A `Server.ScriptTimeout` beállításával a scriptek időtúllépésének határát növelhetjük meg (ez az érték nem lehet alacsonyabb, mint a tulajdonságlapokon beállított alapértelmezés), illetve kérdezhetjük le.

A `Server.Transfer()` és a `Server.Execute()` két, az IIS5-ben új szolgáltatás. A `Server.Transfer()` metódus egyszerűen átadja a vezérlést az alkalmazásban található másik .asp kódnak. Minden átvett adat megmarad, többek között a `Request` objektum teljes tartalma is. A `Server.Transfer()` kiválthatja a `Response.Redirect()` használatát, mert ehhez nincs szükség a böngésző közreműködésére (és így felesleges hálózati forgalomra). Természetesen ez a megoldás csak akkor működik, ha webkiszolgálón belül szeretnénk átirányítani a felhasználót. A `Server.Execute()` szubrutinszerűen végrehajtja az adott oldalt, majd a végrehajtás befejezésre után visszatér az eredetihez (mintha csak beszúrtuk volna a kódot). Ennek előnye az `<!-- include -->` kitétel szemben az, hogy itt akár dinamikusan is generálhatjuk a futtatandó oldal nevét.

A `Server.GetLastError()` visszaadja az utolsó ASP hibának adatait tartalmazó `ASPError` objektumot.

Végül, de egyáltalán nem utolsósorban: a `Server.CreateObject()` metódus létrehoz egy adott objektumot. Ezt már nagyon sokat használtuk, anélkül, hogy tudtuk volna, pontosan mit jelent. Ha a létrehozott objektum tartalmaz `OnStartPage` metódust, azt is meghívja. A `Server.CreateObject()` mellett objektumok létrehozására használhatnánk egyszerűen a `CreateObject()` hívást is. Ez utóbbi nem az IIS, hanem a scriptmotor szolgáltatása, ezért használata az ASP oldalakban – néhány kivételtől eltekintve – nem ajánlott. Az objektum addig marad életben, amíg azt nem töröljük, illetve az ASP oldal végrehajtása véget nem ér. Ha az objektumot a `Session` vagy `Application` objektumba töltöttük, akkor az objektum élete csak az adott `Session` vagy `Application` objektum megszűnésekor ér véget. Egy létrehozott objektumot úgy törölhetünk, ha a változónak más értéket adunk (akár szöveget is, de elterjedt és szép megoldás a `Nothing`):

` Létrehozzuk:

```
Set oFSO = Server.CreateObject("Scripting.FileSystemObject")
```

` Megszüntetjük:

```
Set oFSO = Nothing
```

Naplózás az eseménynaplóba

Ezelőtt bemutattuk a `Response.AppendToLog()` metódust, amelynek segítségével írni lehet az IIS szöveges naplójába. Az ilyen bejegyzések feldolgozása viszonylag nehézkes, ráadásul – ha még emlékszünk – vannak olyan naplóformátumok is, amikor ez a módszer nem vezet eredményre. Szerencsére a Windows Scripting Host-nak köszönhetően a feladatot sokkal elegánsabban is megoldhatjuk. A Windows Scripting Host (és vele együtt természetesen a teljes WSH objektummodell) már az Option Pack 4-gyel bekerül(hetet)t a Windows NT 4.0-ba, és azóta természetesen a Windows 2000 scriptprogramozásának lelkét képezi. A WSH objektummodelljét szerencsére ASP oldalakból is elérhetjük. A `WScript.Shell` objektum `LogEvent()` metódusa lehetővé teszi, hogy belekontárkodjunk a Windows NT/2000 eseménynaplójába. (Windows 9x esetén a bejegyzések a Windows könyvtárban, a `wsh.log` fájlban jönnek létre). A `LogEvent()` metódus három paramétert vár, amiből az első kettőt kötelező megadni, a harmadiknak pedig csak Windows NT/2000-n van értelme:

- A bejegyzés típusa (0: Success, 1: Error, 2: Warning, 4: Information, 8: Audit Success, 16: Audit Error)
- A bejegyzés szövege

- Opcionálisan a Windows NT/2000 számítógép neve, ahol a bejegyzést létre kell hozni (természetesen csak a megfelelő jogosultságok megléte esetén)

```
<%
Set oShell = Server.CreateObject("WScript.Shell")
oShell.LogEvent 0, "Hello from ASP!"
oShell.LogEvent 1, "ERROR from ASP!"
oShell.LogEvent 2, "Warning from ASP!"
oShell.LogEvent 4, "Info from ASP!"
oShell.LogEvent 8, "Audit Success from ASP!"
oShell.LogEvent 16, "Audit Error from ASP!"
Set oShell = Nothing
%>
```

A fenti példában a Server.CreateObject() metódus segítségével létrehoztunk egy Shell objektumot, elkészítettünk néhány bejegyzést, majd jó kiscserkész módjára kitakarítottunk magunk után (az objektumváltozó értékét Nothing-ra állítva felszabadítjuk az objektum által lefoglalt erőforrásokat). A generált naplóbejegyzések (az audit log is) az eseménynapló Application Log-jába kerülnek, a Source mezőben a “WSH” felirat szerepel. Ha a bejegyzést megnyitjuk, a leírás mezőben láthatjuk az általunk megadott szöveget.

Speciális karakterek a HTML kódban

Térjünk vissza kicsit a HTML és ASP mezsgyéjéhez. A HTML dokumentumokban a különleges karaktereket speciális módon, úgynevezett entity-k segítségével írták le. A (kalapos) hosszú ű kódja például û a (hullámos) hosszú ő pedig õ. Sokan a mai napig használják ezt a kódolást, pedig egyrészt hosszú, kényelmetlen, másrészt pedig felesleges. A böngészők már régóta képesek feldolgozni a különféle kódtáblákban írt HTML dokumentumokat – azokba pedig egyszerűen csak bele kell írni a betűket. Itt jegyezném meg, hogy bizonyos jeleket még mindig érdemes entity-k segítségével leírni. Ilyen például a TM (™), a © (©), az ® (®), illetve a matematikai jelek, és más speciális karakterek (€ ¥ ½ §). Ha egy böngészőben megkeressük az Encoding vagy Character Set menüpontot, láthatjuk, hogy mely kódtáblákat képes felismerni és használni. A HTML oldal kódjában pedig a készítő megadhatja a használt kódtábla azonosítóját, az oldal nyelvét, de ha ez elmarad, akkor is van rá esély, hogy a böngésző helyesen ismeri fel azt.

Árvíztűrő tükörfúrógép

A fenti mondat tartalmazza az összes speciális magyar betűt:

```
<html>
<head></head>
<body>
árvíztűrő tükörfúrógép - ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP
</body>
</html>
```

Ha ezt az oldalt megjelenítjük, a böngésző a kódtábla megadása híján megpróbálja felismerni a használt változatot. Ha úgy dönt, hogy nyugat-európai kódolást választ, jönnek a kalapos ékezetek. (Az éppen használt kódtáblát az Encoding menüben láthatjuk kiválasztva. Ha ezt kézzel módosítjuk, a kódolás helyreáll.) A találgatások elkerülése érdekében a böngészőt kifejezetten utasíthatjuk egy adott kódtábla használatára:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
```



```
</head>
<body>
árvíztűrő tükörfúrógép - ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP
</body>
</html>
```

A lényeg az úgynevezett Content-Type fejlécben van, ahol nemcsak az oldal HTML mibenlétét határozzuk meg, hanem a használt kódtablát is. ASP-ben használhatjuk a már említett Response.Charset tulajdonságot. Ha azt használjuk, ugyanezt a hatást érjük el, miközben az ASP oldalon belül elkerülhetjük a META elem használatát :

```
<% Response.Charset = "iso-8859-2" %>
<html>
<head></head>
<body>
árvíztűrő tükörfúrógép - ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP
</body>
</html>
```

További META elemek

Ha már itt tartunk, felsorolnék néhány, eddig még nem említett hasznos metainformációt, amit a weboldalakba ágyazva különféle hatást érhetünk el.

```
<META name="author" content="X Y">
<META name="date" content="12/27/02">
<META name="description" content="ASP">
<META name="keywords" content="ASP">
<META name="robots" content="noindex, nofollow">
<META http-equiv="Content-Language" content="hu">
```

A fenti információk nagy része egyelőre igazán csak a webes keresők számára érdekes. Az author (szerző), date, keywords, description mezők tartalmát a jobb keresők feldolgozzák, keresés során a keywords mezőben megjelenő kulcsszavak például nagyobb súllyal bírnak, mint a dokumentum szövegéből kivonatolt minta. A description mező értékét a keresésre adott válaszban szokás megjeleníteni. Ha oldalunk tartalmaz ilyen mezőt, akkor a rövid leírás nem a dokumentum első néhány sora lesz, hanem a description-ként megadott szöveg. A robots metaelem a keresők felderítő-egységeinek (ezek a robotok) működését szabályozza, a noindex érték arra utasítja a robotot, hogy az oldalt ne indexelje, ne tárolja az adatbázisba, a nofollow pedig azt jelenti, hogy az oldalon található hivatkozásokat nem kell követni. A legfontosabb mégis a Content-Language, ami nem is metainformáció, hanem HTTP fejléc. Értéke a dokumentum nyelvére utaló rövidítés, magyar esetén természetesen a "hu".

Kódtablák

A különféle kódtablák használata azonban nem oldott meg minden gondot. Nem nagyon használhatunk egy dokumentumon belül több kódtablát (a HTML codepage fejléc az egész dokumentumra vonatkozik). Ennél is nagyobb gond az, hogy a legtöbb távol-keleti nyelv több ezer különböző karaktert tartalmaz, ezt pedig nehezen lehet leírni egy 8 bites azonosítóval. Ezeken a területeken természetesen 16 bites kódtablák alakultak ki, ahol egy jelet két bájt határoz meg. Persze ilyen kódtablából is több fajta létezik, ezért kézenfekvő volt, hogy a kódtablák közötti káoszt szabályozni kell. Több vezető számítástechnikai, informatikai cég, nyelvészek, könyvtárosok szervezetei, és még sokan mások ezért megalapították a Unicode Consortiumot. A szervezet célja az volt, hogy olyan szabályrendszert dolgozzanak ki, amivel végre egységesen le lehet írni a világon beszélt (és már vagy még nem használt) összes nyelv minden karakterét,

grafikai szimbólumokat, nyelvi elemeket. A Unicode szerint is minden karaktert két bájt azonosít, így 65535 különböző jel leírására van lehetőség. A táblázat még napjainkban is frissül, sőt, van olyan területe is, ahova mi magunk definiálhatunk karaktereket (a 0xE000-tól 0xF8FF-ig). Ha van kéznél egy Windows 2000, eudcedit.exe! A Unicode azonban nem kódolási szabvány. A karakterek kétbájtos értékét többféleképpen is felhasználhatjuk a dokumentumokban. Többféle kódolási mód terjedt el, természetesen mindegyik alapja maga a Unicode táblázat. Az egyik módszer szerint a dokumentum minden egyes karakterét két bájton ábrázolják, ezzel természetesen megduplázva a méretét. Az alternatív kódolási módok közül az UTF-7 és az UTF-8 terjedt el, ezek közül az UTF-8 lett a gyakoribb.

Az UTF-8 kódolás

Kódolás, és nem kódtábla, hiszen itt már szó sincs tábláról. Kódtáblaként maga a Unicode funkcionál. Lássuk, mit tud az UTF-8. Ebben a kódban egy karakter kódjának hossza 1 és 3 bájt között mozog. Az ASCII karaktereket, tehát az első 127 jelet hagyományosan, egy bájton kezeljük. Ennek köszönhető, hogy az UTF-8 dokumentumok, ha nem tartalmaznak túl sok extra karaktert, emberi szemmel még jól olvashatók. A 128 feletti értékek már nem férnek el egy bájton, ezért azok esetén már hosszabb leíróra van szükség. Miért nem fér bele az első bájtba 255 karakter? Hát azért, mert akkor nem tudnánk megkülönböztetni a 0x6161-es kódot a 0x61 0x61-től. Ezért aztán, ha egy UTF-8 bájt értéke:

- 0x00-0x7F: az egybájtos ASCII karakter kódja
- 0x80-0xBF: több-bájtos érték további bájtjai
- 0xC2-0xDF: kétbájtos érték első bájtja
- 0xE0-0xEF: hárombájtos érték első bájtja

Tehát minél “messzebb” van a Unicode táblában egy karakter, annál hosszabban kell leírni (de legfeljebb három bájton). Ha egy karakter kódja

- 0x0001-0x007F, akkor 1 bájt (0x01-0x7F)
- 0x0080-0x07FF, akkor 2 bájt (0xC280-0xDFBF)
- 0x0800 felett, akkor 3 bájt (0xE0A080-0xEFBBFF)

A távol-keleten tehát az UTF-8 kódolás kicsit pocsékoló, hiszen ott a legtöbb használt karakter az utolsó tartományba esik. A világ nagy részén viszont, ahol az írás alapját valahol mégiscsak a latin betűk képezik, az UTF-8 sok helyet megspórol. A kódtáblák használatához mindössze egy dolgunk van: a Control Panel / Regional Settings dialógus General ablakának alján pipálgassuk be a megcélzott területeket, és némi telepítgetés után birtokunkba vehetjük a teljes Unicode univerzumot.

Lokalizálás

Lokalizálás alatt nem csak az adott ország vagy nyelv betűkészletének használatát értjük, hanem sok minden mást is. Néhány példa, a teljesség igénye nélkül: dátumformátum, hónapok, napok nevei, 12/24 órás időszámítás, AM/PM helyi elnevezése, a dátum és idő elemeit elválasztó karakterek, az elemek sorrendje, a fizetőeszköz jele és írásának módja, a számok írásának módja, a számjegyek csoportosítása, a csoportosításra szolgáló jel, a tizedesjel, a hét első napja, a használt naptár, a sorbarendezés alapja, satöbbi. Minderre fel kell készülnünk, amikor lokalizált alkalmazást készítünk, és nincs ez másképp az ASP alkalmazások esetén sem. A Windows teljes mértékben támogatja ezeket a lokalizációs megoldásokat, ezért tulajdonképpen nincsen nehéz dolgunk. Maga a Windows is a Regional Settings lokalizációs beállításai alapján működik, így jelennek meg a dátumok, időpontok, számok, de még a numerikus billentyűzet “.” gombjának

jelentése is (magyar beállítás esetén ugyanis – helyesen – nem tizedespontot, hanem tizedesvesszőt ír). A GetLocale() függvény például meghívása után visszaadja a számítógépen használt alapértelmezett beállítás kódját, a SetLocale() segítségével pedig beállíthatunk egy más értéket. Ha az adott nyelv támogatása nincs telepítve, a SetLocale() meghívása hibát okoz. A FormatCurrency(), FormatDateTime(), FormatNumber(), FormatPercent() függvények pedig az éppen érvényes beállításnak megfelelően készítik el a fizetőeszköz, dátum és idő, szám és százaléktételeket (szöveges formában, persze).

Hibakezelés az ASP oldalban

Az ASP alkalmazásokban fellépő hibák rendszerint kellemes hibaüzenet formájában jelennek meg a felhasználónál. Természetesen nem feltétlenül egészséges, ha egy-egy ilyen hibaüzenet, esetleg kódrészlet nyilvánosságra kerül, ezért megvan a módja annak, hogy – miközben mi a hiba kezelését végezzük – a felhasználót megkíméljük a kínos részletektől. Az IIS alapértelmezése szerint a \winnt\help\iisHelp\common\500-100.asp oldalt hívja meg. Miközben nyilván nagyon hasznos a részletes hibaüzenet, éles környezetben ez nem feltétlenül igény. Az ASP scriptek végrehajtása során fellépő hibák esetén a teendőt a virtuális könyvtár tulajdonságlapjának Custom Errors oldalán az 500;100 sorhoz tartozó beállítása határozza meg. A hibák kezeléséhez a új IIS Server-től az alábbi támogatást kapjuk: egyrészt a hiba esetén történő átirányítás lehetőségét, másrészt pedig a hiba felismeréséhez, feldolgozásához szükséges információkat, az ASPError objektum formájában. Mielőtt még saját hibakezelő oldalt írunk előtte megfelelő könyvtárban be kell állítani a hibakezelő oldalt a sajátunkra. Mindenekelőtt, ha a pufferba már írtak, amikor a hiba bekövetkezett, itt az ideje, hogy kiürítsük, és a hibakezelő oldal tiszta lappal indulhasson:

```
<%
```

```
If Response.Buffer Then
```

```
Response.Clear
```

Azután: a válasz státuszát állítsuk be hibaüzenetre, nehogy valaki azt higgye, hogy ez a valódi oldal:

```
Response.Status = "500 Internal Server Error"
```

Majd állítsuk be az oldal tartalmát HTML-re, a lejáratit időt 0-ra (így a gyorsítótárak nem fogják az oldalt letárolni):

```
Response.ContentType = "text/html"
```

```
Response.Expires = 0
```

```
End If
```

A következő lépés a hibát leíró ASPError objektum előcsalogatása, amit a következőképpen tehetünk meg:

```
Set objASPError = Server.GetLastError
```

Miután az objektum megvan, belekezdhetünk a jellemzők lekérdezésébe. Az egyes jellemzők jelentése a következő:

- .Category: a hiba kategóriája, szöveges jellemző, pl. "Microsoft VBScript compilation"
- .Number: a hiba kódja, hexadecimálisan megjelenítve sokatmondó lehet (rá lehet keresni a tudásbázisban)
- .Description: a hiba leírása, maga a hibaüzenet
- .File: a fájl neve, ahol a hiba keletkezett
- .Line, .Column: a hiba felismerésének sora és oszlopa (ez az adat nem mindig áll rendelkezésre, ilyenkor az egyes jellemzők értéke -1). Vigyázzunk, a hiba felismerésének helye nem feltétlenül egyezik meg a hiba helyével!
- .Source: a hibát okozó sor kódja. Nem mindig áll rendelkezésre

- .ASPCode, .ASPDescription: a hiba ASP hibakódja és leírása – viszonylag ritkán kapnak értéket

Állítsunk össze ezekből az adatokból egy hibaüzenetet, egyenlőre egy szöveges változóban. Azért ne írjuk ki a képernyőre, mert egyáltalán nem biztos, hogy a felhasználóra tartozik a hiba leírása, jellege és főleg a helye. Miután a hibaüzenetet összeállítottuk, eldönthetjük, hogy kiírjuk-e a böngészőbe, vagy egy illendő bocsánatkérő szöveg keretében elintézzük a dolgot a színfalak mögött. Én azt a megoldást választottam, hogy a felhasználó láthatja a hibaüzenetet, de csak akkor, ha a böngésző a kiszolgálón fut, azaz a kliens és a szerver IP címe megegyezik. Az ellenőrzés pillanatában a hibaüzenet szövegét már az sErrorText változó tartalmazza:

```
<%
If Request.ServerVariables("LOCAL_ADDR") = _
Request.ServerVariables("REMOTE_ADDR") Or _
Request.ServerVariables("REMOTE_ADDR") = _
"127.0.0.1" Then
%>
<HR>
<PRE>
<% Response.Write Server.HtmlEncode(sErrorText) %>
</PRE>
<% Else %>
<P>A hibát naplóztuk. Kérjük, próbálkozzon újra
néhány perc múlva.</P>
<% End If %>
```

Vegyük észre a hibaüzenet kiírásánál használt Server.HtmlEncode() függvényt. Mint mindig, ismeretlen szöveg kiírásánál használnunk kell, különben az esetleg (scriptkódban nagy eséllyel) előforduló speciális karakterek megzavarhatják a HTML kódot – gondoljunk csak néhány ártalmatlan kacsacsőrre, ami scriptkorában még matematikai műveletet jelképezett, most pedig félkész HTML elemeknek értelmeznénk őket. Következő lépésként a korábban bemutatott módszerrel a hibát eltároljuk az Eseménynaplóba:

```
Set oShell = Server.CreateObject("WScript.Shell")
oShell.LogEvent 1, "ASP HIBA!" & vbCRLF & sErrorText
Set oShell = Nothing
```

Végül, biztos, ami biztos, a hibát naplózzuk le egy közönséges szövegfájlba is:

```
Set oFSO = Server.CreateObject
A ("Scripting.FileSystemObject")
Set oFile = oFSO.OpenTextFile
A ("C:\asperror.txt", 8, True)
oFile.Write sErrorText & vbCRLF & vbCRLF
oFile.Close
Set oFSO = Nothing
```

Ezzel készen is vagyunk. Felmerülhet (és remélem, van már, akiben a kód olvasása során fel is merült), hogy a kód így nem teljes: mi történik például akkor, ha a naplófájlt valamilyen okból nem lehet megnyitni (például mert éppen más valaki ír bele)? Ilyenkor természetesen hiba keletkezik. A hibakezelő rutinban keletkező hiba megoldása klasszikus feladat. Ilyenkor a hibakezelő oldal már nyilván nem képes a szálatat tovább a kezében tartani, szükség van egy felsőbb beavatkozására. Ezt maga az IIS végzi el, mi ilyenkor legjobb tudása szerint a felhasználó elé tárja a hibát.

A hiba nem mindig végzetes: az esetek többségében fel tudunk készülni arra az esetre, ha valami “baj” történne – a lényeg csak a hiba biztonságos felismerése. A scriptnyelvek éppen ezért általában beépített hibakezelési elemeket tartalmaznak, amelyeket használva a kisebb hibákat

még a nagyobb problémák bekövetkezte előtt kivédhetjük. A futás közbeni hibakezelő eszközök közös tulajdonsága, hogy a hiba kezelése után a program futtatása folytatódik. A futás közbeni hibakezelés már eléggé scriptnyelv-specifikus, a két beépített nyelv, a VBScript és a JScript elemei a következő.

Kezdjük a VBScript-tel: itt a futás közbeni hibakezelést mindenekelőtt be kell kapcsolni, ha ezt nem tesszük meg, mindenképpen hiba keletkezik. A hibakezelés bekapcsolására a következő parancs használható:

On Error Resume Next

Azaz kb. "Hiba esetén folytasd a végrehajtást". Ezután rajtunk áll, hogy kijavítjuk-e a hibát, vagy egyáltalán foglalkozunk vele. Mindaddig amíg ez a parancs érvényes (márpedig abban a procedúrában ahol kiadtuk, valamint az abból hívott procedúrákban, érvényes), a hiba miatt a VBScript nem fog leállni. Éppen ezért a fenti parancsot globálisan kiadni nem túl ésszerű, a hibakezelés bekapcsolását korlátozzuk kényes helyekre –oda, ahol fel tudunk készülni a hibák javítására. A hiba kezelésére (és felismerésére) az Err objektum használható, amelynek fontosabb jellemzői a következők:

- .Number – a hiba kódja, az Err objektum talán legfontosabb jellemzője. Értéke 0, ha minden rendben van, ettől eltérő, ha hiba történt.
- .Description – a hiba szöveges leírása
- .Source – a hibát "okozó" objektum azonosítója

Az előző példát tehát ki kell bővíteni:

On Error Resume Next

`... hibát okozó rész

If Err.Number <> 0 Then

` Hiba a hibakezelőben...

Err.Clear()

End If

Miután egy hibát kezeltünk, az Err objektumot vissza kell állítani a kezdőértékekre, különben a hibát a következő ellenőrző rutin is elkapná. Az Err.Clear() metódust pontosan erre találták ki. Mi is meghívhatjuk (célszerűen a hibakezelő rutin végén), de a VBScript automatikusan meghívja a következő utasítások végrehajtása során:

On Error Resume Next

Exit Sub

Exit Function

... azaz, a procedúrákból, függvényből történő visszatéréskor, valamint a hibakezelés bekapcsolásakor. Ha elegünk volt a kánaánból, a futás közbeni hibák kezelését visszakapcsolhatjuk az:

On Error Goto 0

paranccsal. Ilyenkor hiba esetén ismét a jól ismert hibakezelési megoldások veszik át az irányítást. Az objektumnak van még egy érdekes metódusa, ez pedig az Err.Raise(). Ez a függvény arra való, hogy hibát váltsunk ki a rendszerben. Az Err.Raise() által generált hibák teljes értékűek.

try...catch...finally

A JScript hibakezelése másképp működik, inkább hasonlít a C nyelvben található try...catch utasításpárra. A szintaxis a következő:

try {

// hibát okozó művelet

}

catch(hibaobjektum) {

// hibakezelés

```

}
finally {
// mindenképp lefutó utasítások
}

```

JScriptben tehát minden hibát okozó művelet(-csoport) esetén külön fel kell készülnünk az esetleg előforduló hibák kezelésére, ami azt is jelenti, hogy minden egyes esetben külön meg kell írunk a hibakezelő rutint is. A gyanús sorokat a try{} blokkba kell írni. Ha az utasítások végrehajtása során bármilyen hiba keletkezne, a végrehajtás a catch{} blokkban folytatódik. A catch paramétereként megadott változó tartalmazza a hiba leírását, egy Error objektumot. Példaképpen lássuk, hogyan kaphatjuk el a FileSystemObject által visszaadott hibát:

```

var oFSO = new ActiveXObject ("Scripting.FileSystemObject");
try {
oFSO.CopyFile("qweqweq.txt", "eqwewqe.txt");
}
catch(e) {
Response.Write("Error: " + e);
Response.Write("<br>Error Number: " + (e.number & 0xFFFF) );
Response.Write("<br>Description: " + e.description);
}

```

A hiba oka nyilvánvaló: az aktuális könyvtárban valószínűleg nincsen "qweqweq.txt" nevű fájl, amit másolhatnánk, ezért hiba történik. A catch-ben megkapott változó egy Error objektumot tartalmaz, erről már az első sorban kiírt [object Error] is tájékoztat minket. A JScript Error objektumának két jellemzője van: az Error.number a hiba kódját, az Error.description a hiba leírását tartalmazza. A hibakód előcsalogatásához az Error.number által visszaadott számot "dekódolnunk" kell, erre való a példában látott (Error.number & 0xFFFF) kifejezés. A try...catch...finally hármas utolsó tagjának az az értelme, hogy olyan utasításokat is megadhassunk, amelyek lefutnak függetlenül attól, hogy a try blokkban történt-e hiba, vagy sem. Hiszen gondoljunk bele: hiba esetén a try blokkból azonnal kiugrunk, az esetleg hátralévő utasítások mennek a sülyesztőbe. Ez a helye az erőforrások, például adatbáziskapcsolatok felszabadításának, lezárásának. JScriptben is idézhetünk elő hibát: a throw() függvény hívásával hasonló eredményt érünk el, mint a VBScript-beli .Raise metódussal. A throw() paramétere a hiba maga, ami lehet egy hibakód, egy szöveg, vagy egy előzőleg előállított, és megfelelően "kitöltött" Error objektum is:

```

var oErr = new Error();
oErr.number = 1234;
oErr.description = "User Error";
try {
throw(oErr);
}
catch(e) {

```

...

Ha a catch blokkban, a hiba kezelése során zsákutcába jutunk, és feladjuk a harcot, a munkát tovább passzolhatjuk az eggyel magasabb szinten található hibakezelőnek a throw() függvénnyel. Paraméterként adjuk neki a catch()-ben átvett változót, valahogy így:

```

catch(e) {
Response.Write("Error: " + e);
Response.Write("<br>Error Number: " +
A (e.number & 0xFFFF) );
Response.Write("<br>Description: " + e.description);
throw(e);

```

}

A második throw() hívás segítségével a hibát tovább passzoltuk az IIS-nek, küzdjön vele ő. A küzdelem eredménye a beállításoknak megfelelő IIS hibaüzenet lesz, mintha csak nem is lett volna a try...catch blokk.

VBScript röviden

Adatípusok

A VBScript egyetlen adattípust definiál és ez a Variant. A Variant típus egy speciális adattípus, amely különféle információk tárolására szolgál. A Variant típusnak attól függően, hogy milyen típusú adatokat tárol a következő altípusai lehetnek.

Altípus	Leírás
Empty	Variant típus nincs inicializálva. Értéke 0 ha számot jelent, és 0 hosszúságú szöveg ha szöveg változó
Null	Variant nem tartalmaz adatot
Boolean	Értéke vagy True vagy False.
Byte	Értéke 0-255-ig terjedő egész szám
Integer	értéke -32,768-32,767 közötti egész szám.
Currency	értéke -922,337,203,685,477.5808- 922,337,203,685,477.5807 között
Long	értéke -2,147,483,648-2,147,483,647 egész
Single	Egyszeres pontosságú lebegőpontos szám -3.402823E38-től -1.401298E-45-ig negatív értékek esetén; és 1.401298E-45-től 3.402823E38-ig pozitív értékek esetén
Double	Dupla pontosságú lebegőpontos szám -1.79769313486232E308-től -4.94065645841247E-324-ig negatív számok esetén és 4.94065645841247E-324-től 1.79769313486232E308-ig pozitív számok esetén
Date (Time)	Egy szám, ami 100 Január 1-től , 9999. December 31-ig terjedő időpontot jelöl.
String	Szöveg ami maximum 2 milliárd karakter hosszú lehet
Object	Objektum tartalma van
Error	Egy hiba számot tartalmaz

Konstans definiálás

VBScriptben a Const kulcsszóval lehet konstansokat definiálni. Például:

```
Const MyString = "This is my string."
```

```
Const MyAge = 49
```

```
Const CutoffDate = #6-1-97#
```

Dátumok definiálásánál a # karaktert kell használni, mit az utolsó példa mutatja.

VBScript operátorok:

aritmetikai

	Leírás	Szimbólum
Hatványozás		^
Negálás		-
Szorzás		*
Osztás		/

Egész osztás	\
Modulus	Mod
Összeadás	+
Kivonás	-
Szöveg összefűzés	&

Összehasonlító

	Leírás	Szimbólum
Egyenlő	=	
Nem egyenlő	<>	
Kisebb	<	
Nagyobb	>	
Kisebb egyenlő	<=	
Nagyobb egyenlő	>=	
Objektum egyezőség	Is	

Logikai

	Leírás	Szimbólum
Logikai tagadás	Not	
Logikai és	And	
Logikai vagy	Or	
Logikai kizáró vagy	Xor	
Logikai ekvivalencia	Eqv	
Logikai implikáció	Imp	

VBScript változók:

Változó deklarálás:

A változók deklarálása VBScriptben a következő módon történik:

```
Dim Names(9)      ' Declare an array with 10 elements.
Dim Names()       ' Declare a dynamic array.
Dim MyVar, MyNum   ' Declare two variables.
Private MyNumber   ' Private Variant variable.
Private MyArray(9) ' Private array variable.
' Multiple Private declarations of Variant variables.
Private MyNumber, MyVar, YourNumber
Public MyNumber    ' Public Variant variable.
Public MyArray(9)  ' Public array variable.
' Multiple Public declarations of Variant variables.
Public MyNumber, MyVar, YourNumber
```

A második sor bemutatta, hogyan lehet egyszerre több változót deklarálni. Egyszerűen vesszővel elválasztva fel kell sorolni a változók neveit.

- **Public** kulcsszóval deklarált változó az összes eljárásban és az összes scriptben látható lesz
- **Dim** kulcsszóval deklarált változó script szinten látható az összes eljárásból, eljárás szinten csak a deklaráló eljárásból látható
- **Private** kulcsszóval deklarált változó csak abból scriptből látható amelyikben definiálva voltak

Elágazások

Pálda:

```
If myDate < Now Then myDate = Now
If value = 0 Then
    AlertLabel.ForeColor = vbRed
    AlertLabel.Font.Bold = True
    AlertLabel.Font.Italic = True
End If
If value = 0 Then
    AlertLabel.ForeColor = vbRed
    AlertLabel.Font.Bold = True
    AlertLabel.Font.Italic = True
Else
    AlertLabel.ForeColor = vbBlack
    AlertLabel.Font.Bold = False
    AlertLabel.Font.Italic = False
End If
If value = 0 Then
    MsgBox value
ElseIf value = 1 Then
    MsgBox value
ElseIf value = 2 then
    MsgBox value
Else
    MsgBox "Value out of range!"
End If
Select Case Document.Form1.CardType.Options(SelectedIndex).Text
    Case "MasterCard"
        DisplayMCLogo
        ValidateMCAccount
    Case "Visa"
        DisplayVisaLogo
        ValidateVisaAccount
    Case "American Express"
        DisplayAMEXCOLogo
        ValidateAMEXCOAccount
    Case Else
        DisplayUnknownImage
        PromptAgain
End Select
```

Ciklusok:

Előtesztelő ciklus amíg igaz:

```
Dim counter, myNum
counter = 0
myNum = 20
Do While myNum > 10
    myNum = myNum - 1
    counter = counter + 1
Loop
MsgBox "The loop made " & counter & " repetitions."
```

Hátultesztelő ciklus amíg igaz:

```
Dim counter, myNum
counter = 0
myNum = 9
Do
    myNum = myNum - 1
    counter = counter + 1
Loop While myNum > 10
MsgBox "The loop made " & counter & " repetitions."
```

Előlesztelő ciklus amíg igaz nem lesz:

```
Dim counter, myNum
counter = 0
myNum = 20
Do Until myNum = 10
    myNum = myNum - 1
    counter = counter + 1
Loop
MsgBox "The loop made " & counter & " repetitions."
```

Hátultesztelő ciklus amíg igaz nem lesz:

```
Dim counter, myNum
counter = 0
myNum = 1
Do
    myNum = myNum + 1
    counter = counter + 1
Loop Until myNum = 10
MsgBox "The loop made " & counter & " repetitions."
```

Kiléps ciklusból EXIT DO segítségével:

```
Do Until myNum = 10
    myNum = myNum - 1
    counter = counter + 1
    If myNum < 10 Then Exit Do
Loop
```

FOR ciklus:

Egyenként léptető:

```
Dim x
For x = 1 To 50
    MyProc
Next
```

Step-ben megadott érték szerint léptet, ha ez az érték negatív akkor visszafelé számlál a ciklus:

```
Dim j, total
For j = 2 To 10 Step 2
    total = total + j
Next
Dim d 'Create a variable
Set d = CreateObject("Scripting.Dictionary")
d.Add "0", "Athens" 'Add some keys and items
d.Add "1", "Belgrade"
d.Add "2", "Cairo"
For Each I in d
```

```
Document.frmForm.Elements(I).Value = D.Item(I)
```

```
Next
```

CGI

A CGI (Common Gateway Interface, általános átjáró felület) nevében is benne van, hogy - mindenféle tévhit ellenében - nem egy programnyelvről van szó, hanem egy felületről, amely összeköti programunkat a HTTP szerverrel, ezen keresztül a felhasználóval. Belátható, hogy CGI programok írásához akármelyik programnyelv felhasználható, egyetlen megkötés, hogy az adott operációs rendszer támogassa azt, és emellett célszerű viszonylag gyorsnak lennie. Az így elkészített alkalmazásaink a WWW-n -World Wide Web- keresztül válnak elérhetővé, így gyakorlatilag bárholnán használhatók, ahol hozzáférhető a Világháló. A technika lényege a dinamikus adatok megjelenítése, tárolása. Fontos róla tudni, hogy az ilyen programok a szerver oldalon hajtódnak végre, tehát nem vetődhet fel olyan probléma, hogy a kliens program -a felhasználó böngészője- nem támogatja ezeket. A honlapokba való integrálásra többféle mód is van, a leggyakrabban csak egy egyszerű hivatkozást teszünk rá, például egy link elhelyezésével - ugyanúgy, mintha az egy HTML dokumentum lenne-, de szintén mindennapos az űrlapok feldolgozásánál való alkalmazás. Ezekben az esetekben a programunk feladata az adatok feldolgozása, és/vagy tárolása, valamint továbbítása mellett egy megfelelő HTML lap előállítására lesz. Tudni kell azonban, hogy a CGI felület alkalmas bármilyen formátumú kimenet előállítására, így akár grafikus kimenetet is készíthetünk, tehát ilyen programokat képként is beilleszthetünk HTML lapokba. Ebből kikövetkeztethető, hogy a kimenet generálásakor nem csak magát a tartalmat, hanem egy válasz-fejlécet is el kell készíteni. Ez általában egy "Content-type: [mime típus]" sorból áll, mellyel meghatározhatjuk a kimenet tartalmának típusát (MIME, Multipurpose Internet Mail Extension). Egy másik hasonló fejléc a Location, mely egy másik fájlra, vagy URL-re irányítja át a szörfprogramot. Fontos még tudni, hogy a fejlécet el kell választani a többi adattól, ezt két újsor karakterrel tehetjük meg (\n). A típus megadása egy fő, és egy altípussal történik. Néhány gyakran használtak közül:

text/html	HTML tartalom (.htm, .html)
text/plain	egyszerű szöveg (.txt)
image/jpeg	JPEG grafika (.jpg, .jpeg, .jpe)
image/gif	GIF kép (.gif)
video/mpeg	MPEG videó (.mpeg, .mpg, .mpe)
video/quicktime	QuickTime videó (.mov, .qt)
application/zip	ZIP archív (.zip)
application/octet-stream	az ilyen tartalmat a böngésző nem jeleníti meg, hanem felkínálja lementésre

Az esetek nagy részében a programot tartalmazó fájl első sorában meg kell adnunk a szervernek a fordító elérési útját egy "#!/elérési/út/fordító" sorral, mely Unix rendszereknél általában /usr/bin/perl, vagy /usr/local/bin/perl - az elérési útról a szerver webmesterétől kaphatunk felvilágosítást.

Ha már Unix rendszereknél tartunk, akkor a jogosultságokat is meg kell említenem. A programokat futtathatóvá kell tenni, és mivel nem előre lefordított programról van szó, az olvasási jog is szükséges, tehát a "chmod 755 programnév.cgi" utasítást kell kiadnunk -így nekünk az olvasás, és a végrehajtás mellett írási jogunk is lesz.

A legfőbb tudnivalók most már a birtokunkban vannak, készítsünk el egy igen egyszerű CGI programot! A példa programokhoz most a következő fejezetben ismertetett Perl nyelvet használjuk fel, de lehetne akár PHP-ban vagy akár C nyelven is megírni ezeket.

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print "Helló, Világ...\n";
exit;
```

Most pedig futtassuk le egy webserveren keresztül! Ennek a kimenete egy Helló, Világ... felirat lesz. Persze ha megnézzük az oldal forrását akkor láthatjuk, hogy ez nem egy szabályos HTML oldal, ezért bővítsük tovább programunkat, készítsünk dinamikus kimenetet!

```
#!/usr/bin/perl
$hanyadika = (localtime(time))[3];
print "Content-type: text/html\n\n";
print <<VEGE;
<HTML>
<HEAD>
<TITLE>CGI példa</TITLE>
</HEAD>
<BODY BGCOLOR="#CCCCCC" TEXT="#666666">
<H1>CGI példa</H1>
<HR NOSHADE>
<H3>"Helló, Világ!"</H3>
<P><FONT COLOR="#000099">Ma <B>$hanyadika</B>-e/a van.</FONT>
</BODY>
</HTML>
VEGE
exit;
```

A kimenet így már szabályos HTML kód, és dinamikus is, hiszen minden nap más számot ír ki. A következő kis program a CGI környezeti változóit iratja ki, amik hasznos információkkal rendelkeznek a a programunk futási környezetéről.

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print <<VEGE;
<HTML>
<HEAD>
<TITLE>CGI környezeti változók</TITLE>
</HEAD>
<BODY>
<H1>CGI környezeti változók</H1>
<PRE>
VEGE
foreach (keys %ENV) {
print "<B>$_:</B> $ENV{$_}\n";
}
print "</PRE>\n</BODY>\n</HTML>";
exit;
```

Vegyük sorra a fontosabb kulcs-érték párokat.

SERVER_SOFTWARE A szerver-program neve, és verziója.

GATEWAY_INTERFACE A gateway (a webservert és a program közötti felület) neve és verziója, általában "CGI/1.1".

DOCUMENT_ROOT Az a könyvtár, amit a kliens a gyökérkönyvtárnak lát.

REMOTE_ADDR A kliens, vagy az általa használt proxy szerver IP címe.

REMOTE_HOST	A kliens host neve, általában nem áll rendelkezésre (a szerver nem határozza meg, vagy egyáltalán nincs).
SERVER_PROTOCOL	A használt protokoll, és annak verziója.
REQUEST_METHOD	A HTTP kérés típusa (ld. később).
QUERY_STRING	GET típusú hívás esetén ebből nyerhetők ki az adatok.
CONTENT_LENGTH	POST típusú hívás esetén az elküldött bájtok száma (ld. később).
HTTP_USER_AGENT	A kliens böngészőprogramjának a neve és verziója, általában az operációs rendszerrel kiegészítve.
HTTP_ACCEPT	A kliens által elfogadott mime típusok listája, általában szerepel benne a */*, mivel így a szerver minden típust elküld, és a böngésző dönti el, hogy mit kezd vele.
HTTP_ACCEPT_LANGUAGE	Azokat a nyelveket találjuk itt, amelyeket a böngésző tulajdonosa beállított, hogy elfogad, el tud olvasni.
SCRIPT_NAME	Programunk virtuális helye (azért virtuális, mert itt az a könyvtár számít a gyökérkönyvtárnak, amelyet a kliens is annak lát).
SCRIPT_FILENAME	Programunk helye a szerveren (a valódi gyökérkönyvtárból számolva)
SERVER_NAME	A szerver neve, vagy IP címe.
REQUEST_URI	Ezt az URI-t kérte a kliens.
SERVER_PORT	A port száma, ahol a kérés érkezett.
HTTP_HOST	A szerver host neve.
PATH_INFO	A CGI program virtuális könyvtárként is használható, ebben a változóban a programnév utáni, további alkönyvtárak találhatók.
PATH_TRANSLATED	Az előző teljes változata.
CONTENT_TYPE	A kéréshez csatolt információ mime típusa (ld. később)

A **SCRIPT_NAME**, és a **SCRIPT_FILENAME** alapján megkülönböztetünk kétféle elérési utat, egy valódit, és egy virtuálisat. Tudni kell, hogy HTTP-n keresztül nem látható a szerver háttértárának a teljes tartalma. Amit a kliens gyökérkönyvtárnak lát, az nem a szerver gyökérkönyvtára, csak egy, a szerver program konfigurálásánál megadott mappa. Az alábbi program a környezeti változókat felhasználva meghatározza valódi, és virtuális helyét a szerveren (utóbbi elé a szerver host-ját is kiírja, így kapjuk meg a teljes url-t):

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print <<VEGE;
ÉN <B>http://$ENV{"HTTP_HOST"}$ENV{"SCRIPT_NAME"}</B> vagyok, de valójában
a(z) <B>$ENV{"SCRIPT_FILENAME"}</B> elérési úton vagyok megtalálható.
VEGE
exit;
```

A végeredményről leolvasható, hogy a látszólag /cgi-bin/pelda.cgi fájl valójában hol is található a szerveren:

Mint már említettem, a bemeneti adatok nagy részét a CGI programok a környezeti változókon keresztül kapják meg a böngészőtől. Két lehetséges módon küldhetünk adatokat, ezek a GET, és a POST eljárások. A GET esetén a sztring az url-ben kerül elküldésre, melyet egy kérdőjel választ el a program nevétől, például: <http://www.szerver.hu/cgi-bin/program.cgi?adatok>. A **QUERY_STRING** változó tartalmazza az így elküldött sztringet. A POST eljárás ebből a

szempontból kivételnek számít, mivel ebben az esetben a küldött karaktereket nem egy környezeti változóból, hanem a standard bemenetről (STDIN) olvashatjuk be. Azonban tudni kell, hogy a sztring után nincs fájlvége karakter, ezért az így elküldött adatok beolvasásánál szükségünk van az elküldött karakterek számára. Erre szolgál a CONTENT_LENGTH környezeti változó, mely ezt a számot tartalmazza. Ezek után belátható, hogy hogyan olvashatjuk be a POST módszerrel küldött adatokat egy változóba:

```
read(STDIN, $adatok, $ENV{"CONTENT_LENGTH"});
```

Az, hogy GET, vagy POST módszert használtunk-e, meg tudható a REQUEST_METHOD változóból, így a program akár maga is el tudja dönteni, hogy honnan kell beolvasnia az adatokat. Most pedig nézzünk meg egy programot, mely egy HTML űrlapot állít elő, majd a visszaküldés után megjeleníti a kapott adatokat.

```
#!/usr/bin/perl
read STDIN, $buffer, $ENV{"CONTENT_LENGTH"};
print <<END;
Content-type: text/html
<HTML>
<HEAD>
<TITLE>CGI példa</TITLE>
</HEAD>
<BODY BGCOLOR="#EEEEEE" TEXT="#006699">
<H3>?rlap</H3>
<FORM ACTION="$ENV{"SCRIPT_NAME"}" METHOD="post">
Név: <INPUT TYPE="text" NAME="nev">
<BR>Életkor: <SELECT NAME="kor">
<OPTION>Kérem, válasszon...
<OPTION VALUE="-14">14 év alatt
<OPTION VALUE="15-24">15-24 év
<OPTION VALUE="25-39">25-39 év
<OPTION VALUE="40-">40 évnél több</SELECT>
<BR><INPUT TYPE="checkbox" NAME="mobil" VALUE="igen"> Van mobiltelefonom.
<INPUT TYPE="checkbox" NAME="auto" VALUE="igen"> Van autóm.
<BR><INPUT TYPE="submit" VALUE="Mehet!"></FORM>
<H3>Url-ben kapott adatok:</H3>
<PRE>
$ENV{"QUERY_STRING"}
</PRE>
<H3>Standard bemenetre kapott adatok:</H3>
<PRE>
$buffer
</PRE>
</BODY>
</HTML>
END
exit;
```

Ezek után lássuk magát a PERL nyelvet.

Perl

Perl 5 nyelv rövid összefoglalása

A Perl nyelv egy interpretált - illetve betöltéskor fordított - nyelv. Eredetileg rendszeradminisztrációs feladatok megkönnyítésére írta Larry Wall, mert nem volt kedve a meglévő eszközök korlátaival bajlódni. A nyelv meglévő eszközökre lett alapozva: C, sed, awk

és sh programokra. Perl-ben csak a számítógép hardware korlátai érvényesülnek: egy teljes file-t képes beolvasni egy string változóba (ha van elég memória), tetszőleges mélységű rekurzió futtatható benne (ha van türelmünk és memóriánk). Asszociatív tömbök elérését hash táblákkal gyorsítja (ami meglepően hatékony programok írását teszi lehetővé). Nagyon gyors és rugalmas mintaillesztő algoritmusa van szövegek keresésére és cseréjére (eredetileg szövegfile-ok feldolgozására találták ki, mint azt a neve is mutatja: Practical Extraction and Report Language). Képes bináris adatokkal is dolgozni, és ezekből bonyolult adatstruktúrákat felépíteni. Az adminisztrációs feladatok megkönnyítésére az asszociatív tömbökhöz adatbázis file-okat rendelhetünk, melyek szerkezetét egy gyakorlott programozó maga is megadhatja. Az 5-ös verziótól kezdve már használhatjuk a moduláris programozást támogató nyelvi konstrukciókat, sőt már Objektum Orientált eszközöket is. A Perl-ben setuid programok sokkal biztonságosabban írhatók, mint C nyelvben az adatfolyam követését biztosító funkciók miatt (ld.: -T kapcsoló). Elkerülhetünk egy csomó egyszerű hibát, amit a C programban csak debuggolással fedeznénk fel. Egyszerűen sokkal jobb a fejlesztési idő/futtatási idő arány ha egy ritkán használt, vagy futási időben nem kritikus (pl. CGI program) program írásánál. Ez a leírás UNIX rendszert használó gépeken használható fel igazán. A konkrét nyelvi részek a DOS, OS/2 és Windows alatt futó Perl interpreterekre is igaz, de a környezetfüggő részek és a példák csak UNIX alapú rendszereken mennek.

Indulás

Kezdetnek mindjárt megnézhetjük a minimális Perl programot, ami a "Hello World!!!" szöveg kiírásához kell:

```
#!/usr/local/bin/perl
print "Hello World!!!\n";
```

A Perl program egy egyszerű szövegfile, és követi a shell scriptek gyakorlatát a "#" karaktert használva a megjegyzések jelölésére. Itt ez az első sorban egy speciális jelentéssel is bír, hiszen az első sorba írt "!" kezdetű megjegyzés a script interpreterét határozza meg. Ha ez nem lenne ott, akkor a perl <programnév> paranccsal lehetne lefuttatni. Látható, hogy a kiírás egy egyszerű print utasítással megadható, és az utasítást mindig egy ";" zárja be. A szövegek egy C programhoz hasonlóan itt sem tartalmazznak implicit sorvége jelet - ezt a megszokott módon nekünk kell kiírni. (Sor beolvasásakor viszont nem vágja le a sorvége jelet, így azt is a programozónak kell kezelnie.)

Adatstruktúrák

Három alapvető adatstruktúra található a nyelvben: skalár, skalárok tömbje és az asszociatív tömb. Ezeket - mint azt későbbiekben látni fogjuk - mutatókkal is lehet kombinálni, így lényegében mindenféle adatstruktúrát megalkothatunk. A normál tömbök indexe - általában - 0-val kezdődik. Az asszociatív tömböket szövegekkel kell indexelni. A változó nevének első karaktere határozza meg annak típusát:

\$ skalárokat,

@ számmal indexelt és a

% asszociatív tömböket jelöl.

Ez az első karakter a változónak a kifejezésben betöltött szerepét kell, hogy takarja.

Példák:

```
$days      # egyszerű skalár
$days[3]   # days nevű tömb negyedik eleme
```

```

@days      # az egész tömb ($days[0], .. , $days[$#days])
$days{'Feb'} # days nevű asszociatív tömb egy eleme
%days      # az egész asszociatív tömb (kulcs, érték párok)
$#days     # days tömb utolsó indexe

```

Az alprogramokat még a & jellel is megjelölhetjük. Erre akkor lehet szükség, ha az alprogramot deklarációja előtt szeretnénk használni. Függvényre a deklarációja után már a & szimbólum nélkül is hivatkozhatunk (ez - a nyelv előző verziójától eltérően - minden függvényre alkalmazható). A változók nevei egyébként a típusuknak megfelelő külön szimbólumtáblába kerülnek, tehát használhatunk azonos nevű tömböt, asszociatív tömböt és skalárt. (Ekkor \$size[0] az @ize tömb része, nem pedig az \$size változót helyettesíti.) A változók nevei betűvel kell hogy kezdődjenek, és tetszőleges alfanumerikus karakterrel, illetve aláhúzásjellel folytatható. A változó neve nem kell, hogy megadott legyen, akár egy kifejezés is állhat helyette, melynek aktuális értéke lesz a valódi név! Vannak speciális változónevek alapvető paraméterek jelölésére. Ezek az Előre definiált változók részben megtalálhatóak.

Környezet

A kifejezés típusa a kifejezés környezetétől is függhet! Például az `int(<STDIN>)` eredménye egy egész szám lesz, amit a bemenet egy sorából állít elő az `int` függvény, de a `sort(<STDIN>)` eredménye egy lista lesz, amit a bemenet összes sorának beolvasása után a `sort` függvény rendezéssel állít elő!

Skalárok

Skalár változók sokféle értéket tárolhatnak: szám, szöveg, bináris adat vagy mutató. A tartalom a használat során mindig szükséges formára alakul, azaz kiírhatunk egy számot szöveggént egy változóba, majd ezt a változót megszorozhatjuk kettővel. Az aritmetikai műveleteknél a számok bináris formára konvertálódnak, így ezek pontosságát a processzor korlátozza. Vannak persze kiegészítő modulok tetszőleges pontosságú számok kezelésére is. Feltételek vizsgálatánál csak az üres szöveg, illetve a 0 érték jelent hamisat. A skalár változóknak alapvetően két állapota lehet: definiált, vagy definiálatlan. Ezeket a `defined`, illetve `undefined` függvényekkel kérdezhetjük le. Egy változó addig definiálatlan, amíg valaki nem ad neki értéket, vagy nem definiálja explicit módon. Definiálatlan változónak - a környezetétől függően - 0 vagy üres szöveg értéke van.

Példák:

```

$i = 2;                # egész
$f = 2.3;              # racionális
$szam = 1_000_000_000; # ez egy nagy egész
$hexa = 0xffeedd;      # hexadecimális szám
$szoveg = "Ez egy szoveg"; # egy szöveg
print "Valami\n";      # egy szöveg megjelenítése
print <<VEGE;          # hosszabb szöveg megjelenítése
Valami                # "Itt a dokumentum" stílus
VEGE                  # adatbevitellel (VEGE a zárószimbólum)
$szoveg = <<VEGE;      # $szoveg = "ez is szoveg\n"
ez is szovega         # ez is megy
VEGE
fv(<<EGY, <<KETTO);    # sőt ez is!
elso parameter

```


EGY
masodik parameter
KETTO

Tömbök

A Perl-ben használt tömböket inkább indexelt listáknak kéne nevezni a kiterjeszthetőségük miatt. Ha bármelyik típusban egy eddig nem létező elemnek adunk értéket az automatikusan definiálódik. A tömböket persze nem csak elemenként lehet feltölteni:

```
@tomb = ('elso', 2, $harom); # háromelemű tömb
%szinek = (                               # asszociatív tömb
    'piros' => 0x00f,
    'kék' => 0x0f0,
    'zöld' => 0xf00,
);
```

És az utóbbival ekvivalens alak:

```
%szinek = (
    'piros', 0x00f,
    'kék', 0x0f0,
    'zöld', 0xf00,
); #egyszerű felsorolás, párosával
```

Értékadásokban (rész)tömböket is értékül adhatunk. Ezekben az esetekben a balérték listája az első elemtől kezdve addig töltődik fel, amíg van új elem a jobb oldalon. Az értékadás akkor és csak akkor érvényes, ha a baloldalon legális balértékek szerepelnek.

```
($a,$b,$c)=(1,2,3)    #a változók sorban értéket kapnak
($a,$b,@maradek)=@valami#ez is helyes
```

Tömb balértékként használata csak utolsó elemnek érdemes, mert egy korábbi tömb az összes jobboldalt magába olvassa. Ez jó a lokális paraméterek átvételénél. A \$[változó mutatja, hogy a tömbök melyik indexen kezdődnek (ez a C-ben 0). Ennek értéke módosítható ,ezért a következő két értékadás ekvivalens:

```
@tomb=()              #üressé teszi a tömböt
$#tomb=$[-1;          #a tömb hosszát -1-re állítja
```

Skalár környezetben a tömb felhasználása a tömb aktuális hosszát adja vissza. Értékadásnál a résztömbök automatikusan elemenként bemásolódnak, így a keletkező tömb homogén lesz.

```
@tomb=(@lista1,@lista2,&alprg)
```

Egy asszociatív tömb skalár környezetben visszaadja, hogy van-e benne kulcs-érték pár. Ez csupán arra jó, hogy a hash-elő algoritmus hatékonyságát vizsgáljuk.

Szintaxis ... utasítások

A Perl program utasítások és deklarációk egymásutánja, amelybe megjegyzéseket a # jel használatával tehetünk. Ennek hatására a Perl interpreter a sor végéig lévő szövegrészt megjegyzésnek tekinti. Annak ellenére, hogy a nyelv UNIX-szűrő jellegű nyelvi elemeket is tartalmaz, minden utasítás csak egyszer kerül végrehajtásra, kivéve, ha ciklusban szerepel. A nyelvben az utasítások és deklarációk szabadon - akár keverve is, mint egy C++ programban - követhetik egymást. Megkötés csak az alprogramok hívására van: a még nem deklarált alprogramot csak a & szimbólum használatával lehet meghívni.

Egyszerű utasítás

Egy utasítás egy pontosvesszővel lezárt jelsorozat lehet. Ezekből több is szerepelhet egy sorban. Egy utasítás után egy módosító kifejezés állhat, amely ennek az egyetlen utasításnak a hatását befolyásolja:

- utasítás if *EXPR*
- utasítás unless *EXPR*
- utasítás while *EXPR*
- utasítás until *EXPR*

Egy üzenet kiírása feltételtől függően:

```
print "Hello kedves olvasó!\n" if $bobszedu_program;
```

Összetett utasítás

Itt kell megemlíteni a BLOKK fogalmát, amely { és } jelekkel közbezárt utasítássorozat. (Itt fontosak a { és } jelek, még egyetlen utasításnál is!) Ez alapján a lehetséges formák:

```
if (EXPR) BLOKK
    # BLOKK végrehajtódik ha az EXPR igaz
if (EXPR) BLOKK1 else BLOKK2
    # ha EXPR igaz akkor BLOKK1, egyébként BLOKK2
    # lesz végrehajtva
if (EXPR) BLOKK elsif (EXPR) BLOKK ... else BLOKK
```

Az if utasítás szintaxisa itt egyértelmű lesz, mivel a BLOKK nem állhat egyetlen utasításból.

```
CIMKE while (EXPR) BLOKK
CIMKE while (EXPR) BLOKK continue BLOKK
```

A while ciklus törzsében végrehajtott next utasítás hatására a vezérlés a continue BLOKK-ra kerül, majd újra elindul a ciklusmag.

```
CIMKE for (EXPR1; EXPR2; EXPR3) BLOKK
```

Ez a szokásos C-beli ciklus formája. A következő két forma ekvivalens:

```
for($i = 1; $i < 10; $i++) {
    ...
}
$ i = 1;
while($i < 10) {
    ...
} continue {
    $i++;
}
```

```
CIMKE foreach változó (TOMB) BLOKK
```

Ez a shell-ekben meglévő ciklus egy változata. Itt a változó sorban felveszi a TOMB elemeit értékül, és így indul el a ciklusmag.

```
CIMKE BLOKK continue BLOKK
```

Ez a végtelen ciklus volt...

A ciklusokban használható a next, a last és a redo utasítás, melyek a ciklusbeli utasítások végrehajtását vezérik. A fent említett next hatására a futás a ciklus elejére kerül, és a feltétel újratestelésével folytatódik a működés. A redo ehhez hasonló, csak itt a feltétel tesztelése nélkül kerül a végrehajtás a ciklus első utasítására. A last pedig a ciklusból kiugrik, és az utána következő első utasításon folyik tovább a program végrehajtása. A három utasítás használatában érdekesség, hogy mindegyiket címkézni is lehet és ekkor az ugrások az adott címkével ellátott ciklusra vonatkoznak. Mivel a blokk egy egyszer lefutó ciklusnak tekinthető, ezért ezek az

utasítások blokkban is használhatóak. A switch utasításra nincs külön forma, de van rá egypár lehetséges megoldás, például:

```
SWITCH: {
    /^abc/ && do { $abc = 1; last SWITCH; };
    /^def/ && do { $def = 1; last SWITCH; };
    /^xyz/ && do { $xyz = 1; last SWITCH; };
    $nothing = 1;
}
```

A /^abc/ alakú feltételek mintaillesztésre szolgálnak. Ha egy minta illeszkedik a \$_ változó tartalmához, akkor a hozzá tartozó feltétel második tagja is kiértékelésre kerül, azaz a do blokk is végrehajtódik.

Operátorok és precedenciájuk

A C nyelvben érvényes szabályok érvényesek, és még van egypár - shell script-ekből ismerős - új operátor.

Operátor	kiértékelés iránya	leírás
lista (,)	balról jobbra	kifejezések listája
->	balról jobbra	hivatkozás
++,--	nem asszociatív	növelés, csökkentés
**	jobbról balra	hatványozás
!,~, \ és unáris +, -	jobbról balra	nem, bináris nem, címoperátor, +, -
=~, !~	balról jobbra	szöveg illeszkedés, nem illeszkedés
*, /, %, x	balról jobbra	szorzás, osztás, modulus, ismétlés
+, -, .	balról jobbra	összeadás, kivonás, konkatenáció
<<, >>	balról jobbra	balra, illetve jobbra shift
unáris operátorok	nem asszociatív	pl. file tesztelés -f
<, >, <=, >=, lt, gt, le, ge	nem asszociatív	szám illetve szöveg összehasonlítása
==, !=, <=>, eq, ne, cmp	nem asszociatív	szám illetve szöveg összehasonlítása
&	balról jobbra	bináris AND
~, ^	balról jobbra	bináris OR és XOR
&&	balról jobbra	logikai AND
	balról jobbra	logikai OR
..	nem asszociatív	tartomány
?:	jobbról balra	feltételes értékadás
=, +=, -=, *= ...	jobbról balra	értékadás operátorai
, =>	balról jobbra	vessző és kulcs operátor
lista operátorok	nem asszociatív	lista manipulációk
Not	balról jobbra	logikai NEM
And	balról jobbra	logikai ÉS
or, xor	balról jobbra	logikai VAGY-ok

Itt csak az ismeretlennek tűnő operátorokat fogom kiemelni:

címoperátor

olyan mint a C nyelv & operátora; visszaadja az operandus objektum címét (alprogramokra is megy!)

ismétlés

egy szöveget meg lehet ismételni néhányszor, pl.: "ha"x3 == "hahaha"

konkatenáció

szövegek összefűzése, pl.: "ha"."ha" == "haha"

file tesztelés

ezek a shell programokban megszokott file tesztelő operátorok, pl.: -f "hello.c" akkor igaz, ha a hello.c file létezik

szöveg összehasonlítás

Erre a lt,gt,le,ge,eq,ne operátorok szolgálnak. Ha szövegeket az == operátorokkal hasonlítgatjuk, akkor azok memóriabeli címei kerülnek összehasonlításra, nem tartalmi! cmp értéke -1, 0, vagy 1 lehet a szövegektől függően.

szám összehasonlítás

Szokásos operátorokon kívül a <=> szerepel itt. Ennek -1 az értéke, ha az első szám nagyobb, 1, ha a második, és 0, ha egyenlő a két szám. Ez az operátor - az előbb említett cmp operátorhoz hasonlóan - főleg rendezéseknél használható jól.

I/O

Az alapvető és legfontosabb I/O operátor a < és a >. Ha egy file leíróját ilyen jelek közé rakjuk, akkor egy sort olvashatunk belőle. A beolvasott sor automatikusan a \$_ változóhoz rendelődik, ha nem adunk meg mást. Egy egyszerű cat (UNIX cat parancs) így is leírható:

```
while(<STDIN>)
{
    print $_; # print; is lehetne, hiszen az $_ az
              # alapertelemezett argumentum
}
```

Megjegyezendő még, hogy a C nyelv három operátora nem szerepel a nyelvben, ezek a következők:

- & - cím-operátor, helyette a \ használható,
- - dereferencia-operátor, erre valók a \$,@,%,& operátorok
- (TYPE) - típus-kényszerítés

A szövegekhez még járul egypár "idézőjel" operátor

Általában	hivatalos jelölés	értelme	megjegyzés
"	q{}	szöveg literál	ez a "szó szerinti" szöveg literál
""	qq{}	szöveg literál	változókat helyettesíti a szövegben

``	qx{}	parancs	az adott szöveget, mint egy shell parancssort végrehajtja
	qw{}	a szövegből egy szólistát csinál	pl.: paraméterátadás
//	m{}	mintaillesztés	változókat értékükkel helyettesíti
	s{ }{ }	csere	változókat értékükkel helyettesíti
	tr{ }{ }	betűcsere	az szövegeket mint cseretáblát használja

Mintaillesztés

Használat

A mintaillesztést több dologra is lehet használni a =~ vagy a !~ operátorokkal:

Szövegrészek felismerésére: egy szövegben a grep programhoz hasonlóan kereshetünk szövegeket.

Pl.: \$sor =~ /keresett szo/;

Szövegekben a sed-hez hasonlóan lecserélhetünk egyes részeket.

Pl.: \$sor =~ s/eredeti/uj szo/;

Szövegekből kikereshetünk minket érdeklő részeket, és azokat szelektíven ki is nyerhetjük (mint az awk-ban). Pl.: (\$erdekes_szo) = /\s+(\w+)\\$/;

Módosítók

A // után általában írhatunk valamilyen karaktert, ami a keresést egy kicsit módosítja:

i	kis- és nagybetűket nem különbözteti meg
m	többsoros szövegben keres
s	a szöveget egyetlen sorként kezeli
x	kibővített keresési mód

Az i módosító használatával így a /perl/i kifejezés a Perl szövegre is illeszkedni fog.

Az x módosító tulajdonképpen arra jó, hogy egy kicsit lazább szintaxszissal írassuk le a kereső kifejezéseket. Ezzel már lehetőség van többsoros, sőt megjegyzésekkel tűzdelt kifejezés írására is!

Regular Expressions, reguláris kifejezések

Egy ilyen kifejezésben egy szöveghez illeszthető mintát lehet leírni. Ebbe a mintába persze belevehetünk extra dolgokat is, hogy a nekünk szükséges részt nyerjük ki a szövegből. (A UNIX grep parancsában megszokott dolgok a () csoportosító operátortól eltekintve.)

Az alapok

\	a következő karakter speciális
^	a sor elejéhez illeszkedik
.	egy tetszőleges karakterhez illeszkedik (kivéve az újsort)
\$	sor végéhez illeszkedik

	alternatívák jelölése
()	csoportosítás
[]	karakter-osztály kijelölése

A . csak akkor fog az újsor karakterhez illeszkedni, ha erre az s módosítóval külön megkérjük, pl.:

```
$szoveg = <<VEGE;
Tobbsoros szoveg, amelyben a PERL
perl szavakat kell majd megtalálni.
VEGE
print "illeszkedik/s\n" if $szoveg =~ /PERL.perl/s; # igaz
print "illeszkedik/\n" if $szoveg =~ /PERL.perl/; # hamis
```

A sor eleje és vége inkább rekord elejét és végét jelenti, hiszen a nyelvben meg lehet határozni, hogy mi válassza el a sorokat (\$*). Alapesetben ez persze az újsor karakter, de ezt meg lehet változtatni...

Karaktorsorozatok

Egy egyszerű kifejezés ismétlődését a következőkkel lehet jelölni:

*	0 vagy több
+	1 vagy több
?	0 vagy 1
{n}	pontosan n-szer
{n,}	n-szer vagy többször
{n,m}	n <= ismétlődések száma <= m

Alapértelmezés szerint ekkor a leghosszabb ismétlődés fog illeszkedni ezekhez a részekhez. Ha minimális számú illeszkedést szeretnénk, akkor mindegyik után odatehetjük a ? jelet:

```
"hhhhhh" =~ /h{2,4}/;      # itt "hhhh" fog illeszkedni
"hhhhhh" =~ /h{2,4}?/;     # itt csak "hh"
```

Speciális karakterek

\t	tabulátorjel
\n	újsor
\r	return
\f	form feed
\v	vertical tab
\a	csengő
\e	escape
\033	oktális számrendszerben megadott karakter
\x1b	karakter hexadecimálisan
\c[kontrol karakter
\l	kisbetű
\u	nagybetű
\L	kisbetű \E-ig
\U	nagybetű \E-ig
\E	...

\Q	metakarakterek normálisak \E-ig
\w	"szó" karakter (alfanumerikus és _)
\W	nem-szó karakter
\s	whitespace
\S	nem whitespace
\d	számjegy
\D	nem számjegy
\b	szóhatárhoz illeszkedik
\B	nem szóhatár
\A	string elejéhez illeszkedik
\Z	string vége
\G	oda illeszkedik, ahol az előző illesztés végetért

Az x módosító használatával még lehet használni más dolgokat is, de ezek szerintem már csak igen-igen ritkán kerülnek elő.

Csoportosítás

Ha egy szövegből részeket ki akarunk nyerni, akkor a () karaktereket kell használnunk. Ha ezekkel bezárunk egy karaktersorozatot a reguláris kifejezésben, akkor az ahhoz illeszkedő karakterekre a kiejezésen kívül is hivatkozhatunk. Ha egy csere környezetben használjuk, akkor az így kapott karaktersorozatokra a \$1, \$2, \$3 ... változókkal lehet hivatkozni:

```
s/^( [^ ]* ) * ( [^ ]* ) /$2 $1/;          # első két szó felcserélése
if(`date` =~ /(.):(.):(.)/) {             # idő kiírása
    print "ora: $1, perc: $2, masodperc: $3\n";
}
```

Ha lista környezetben használjuk az illesztést, akkor a kiválasztott értékeket közvetlenül is megkaphatjuk:

```
($ora, $perc, $masodperc) = (`date` =~ /(.):(.):(.)/);
```

Operátorok átlapolása

A Perl operátorainak nagyrésze átlapolható. Ehhez a %OVERLOAD tömböt kell használni, mely az átlapolt operátorokhoz tartozó függvényeket tárolja.

```
package Valami;
    %OVERLOAD = (
        '+' => \&myadd,
        '-' => \&mysub,
        # stb.
    );
...
package main;
    $a = new Valami 57;
    $b = $a + 5;
```

Elképzelhető az az eset is, hogy nincsen definiált operátor az adott típusú változóhoz. Ennek kezelésére szolgál az %OVERLOAD tömbnek "fallback" nevű eleme. A "fallback" értékétől függően három eset lehetséges:

definiálatlan

Ekkor a Perl először elindítja a MAGIC AUTOGENERATION nevű eljárást, mely egy függvényt próbál generálni az adott operátorhoz. Ha ez nem megy, akkor a felhasználó által definiált "nomethod"-ot hívja, ami ezekre az esetekre lett létrehozva. Ha ez sem működik, akkor exception lép fel.

TRUE

Minden az előző esetnek megfelelően zajlik, csak exception nem lép fel, hanem minden úgy folytatódik, mintha nem is lett volna átlapolás.

FALSE

A definiálatlan esettől abban tér el, hogy az autogenerálást nem próbálja meg.

Beépített függvények

Ez a rész iszonyú nagy, és csomó olyan információt tartalmaz, amit már amúgy is ismer az ember (legalábbis a C programokban használta már őket). Itt csak néhány speciális dolgot fogok ismertetni, ami Perl jellegzetesség.

Ha valaki ismeri a szokásos C függvényeket, akkor azokat bátran használhatja, biztosan megvan. Ha nem úgy viselkedik, ahogy várta, akkor érdemes igénybe venni a POSIX modult:

```
use POSIX;
```

Ezek után már bitosan meglesz az összes POSIX függvény.

Ha valakinek még ez sem elég, akkor igénybeveheti a különböző modulokat. Már a nyelvvel együtt lehet találni adatbáziskezelő kiterjesztésektől kezdve a terminálkezelő függvényekig sok mindent. Ezeken kívül az Internetben legalább száz új modul kínál kiterjesztéseket különféle nyelvek és könyvtárak felé (pl.: SQL adatbáziskezelés, X11 grafikus felület, Prolog...).

néhány függvény...

bless REF,PACKAGE

A REF referenciát egy PACKAGE modulban leírt objektumként fogja kezelni. Lényegében ez az útja egy új objektum létrehozásának.

caller

Megmonja, hogy ki hívta az aktuális kódrészletet...

```
($package, $filename, $line) = caller;
```

chop

Levágja a '\n' jelet a sor végéről. Ez sorok olvasása után hasznos.

```
while(<>) {  
    chop; # $_ végéről vág  
    ...  
}  
chop($cwd = `pwd`); # aktuális könyvtár
```

defined EXPR

Megmondja, hogy egy EXPR változó definiált-e.

delete EXPR

Egy elem törlése egy asszociatív tömbből, pl.: delete \$tomb{"eleme"};

die LIST

Ez lényegében a C könyvtári `exit()` függvény, csak a `LIST` tartalmát még kiírja mielőtt kilép a programból.

do BLOCK

Végrehajtja a `BLOCK`-ot majd az utolsó utasítás értékét adja vissza. Ha ciklusmódosítóval használjuk, akkor a ciklusmag még a tesztelés előtt biztos lefut.

do EXPR

Az `EXPR`-t, mint Perl forrásnevet használja, és végrehajtja a benne lévő utasításokat. Minden hívásnál újra kielemez a fájlnevet, ezért nem érdemes például ciklusban használni.

dump LABEL

Egy `core dump`! Ha a `core dump` eredményeképpen kapott file-t futtatjuk, akkor az a `LABEL` címkénél fog elindulni. Ez a módja egy Perl program "fordításának".

each ASSOC_ARRAY

Egy asszociatív tömb elemeit iterálja:

```
while(($key,$value) = each %ENV)
{
    print "$key=$value\n";
}
```

környezeti változók kiírása (ld. még a változókat)

eval BLOCK

A `BLOCK` futtatása az aktuális környezetben. A `BLOCK` egy string is lehet!

exists EXPR

Megmondja, hogy létezik-e az asszociatív tömb egy eleme, pl.: `if exists $tomb{"eleme"} ...`

glob EXPR

`EXPR` file-név teljesen kiterjesztve. Úgy működik mint a shell-ek file-név kiterjesztő algoritmus.

keys ASSOC_ARRAY

`ASSOC_ARRAY` kulcsait adja vissza egy tömbben.

local EXPR

`EXPR`-ben felsorolt változók a blokkra nézve lokálisak lesznek. Ez a dinamikus láthatóság.

my EXPR

`EXPR`-ben lévő változók csak az aktuális blokk-ban lesznek láthatóak. A `local`-tól eltérően ez fordítási időben értékelődik ki, tehát a változónevekben nem lehetnek mágikus karaktertek. Az így definiált változók inkább a C nyelv lokális változóihoz állnak közel, mert a külvilág számára teljesen láthatatlanok lesznek.

```
sub kiir
{
    my ($szoveg1, $szoveg2) = @_;
    chop $szoveg1;
    chop $szoveg2;
    print $szoveg1, ' ', $szoveg2, '\n\n';
}
```

open FILEHANDLE, EXPR

`EXPR`-ben leírt file megnyitása a `FILEHANDLE` leíróba.

```
open(FILE,"valami.txt");    # írás-olvasás
open(BE,"</etc/passwd");    # olvasás
open(KI,">/tmp/output");    # írás
```

```
open(FINGER,"finger @augusta |"); # olvasás pipe-ból
open(RENDEZ,"| sort >/tmp/ki");# írás pipe-ba
```

Az így megnyitott file-okat a close-val lehet lezárni, és I/O műveletekben lehet használni:

```
print FILE "Egy sor\n";      # "Egy sor" kiírása a FILE-ba
$Sor = <BE>                  # sor olvasása
```

pack MINTA,LIST

A LIST tartalmát a MINTA szerint binárisan összehatolja. Főleg külső eljárások hívása előtt kell megtenni.

print FILEHANDLE LIST

Kiírás általános utasítása. A FILEHANDLE elhagyható, ekkor a standard kimenetre ír ki. A FILEHANDLE és a LIST között nem lehet vessző!

return

Alprogramból érték visszaadása. Ha ez nem szerepel, akkor az alprogram utolsó utasításának értéke lesz visszaadva.

shift LIST

A LIST tömböt elmozdítja egy elemmel lefele. Az első elemmel tér vissza, és az törlődik is abból.

sort LIST, vagy sort SUBNAME LIST

LIST rendezése lexikografikusan, vagy a SUBNAME-ben definiált alprogramnak megfelelően. A SUBNAME alprogramnak egy érvényes összehasonlításnak kell lenni.

study SCALAR

Az adott változó tartalmát tanulmányozza egy kicsit, hogy később több mintaillesztést hatékonyabban végezhesünk el rajta. Ez tényleg csak akkor jó, ha több mintaillesztést használunk ugyanarra a változóra!

tie VARIABLE,PACKAGENAME,LIST

Az adott változót hozzárendeli a PACKAGENAME-ben leírt struktúrához. Ezzel lehet egy egyszerű asszociatív tömbhöz hozzárendelni egy adatbázist, ahol a tömb indexei lényegében keresési kulcsok lesznek. A PACKAGENAME modulban a változó típusától függően különböző függvényeket kell megírni:

- 1) asszociatív tömb
 - a) TIEHASH objectname, LIST
 - b) DESTROY this
 - c) FETCH this, key
- 2) STORE this, key, value
 - (a) DELETE this, key
 - b) EXISTS this, key
 - c) FIRSTKEY this, key
 - d) NEXTKEY this, key
- 3) tömb
 - a) TIEARRAY objectname, LIST
 - b) DESTROY this
 - c) FETCH this, key
- 4) STORE this, key, value
 - (1) skálár
 - b) TIESCALAR objectname, LIST
 - c) DESTROY this

- d) FETCH this
- e) STORE this, value

undef EXPR

EXPR változó megszüntetése

untie VARIABLE

Egy tie kapcsolat megszüntetése.

unapck

pack függvény ellenkezője

use MODULE, LIST

MODULE modul LIST-ben felsorolt nevei láthatóvá válnak az aktuális package-ban. Ha a LIST elmarad, akkor a MODULE által exportált változók válnak láthatóvá. A no kulcsszó a use ellentettje, azaz a láthatóságot megszünteti.

wantarray

Igaz lesz az értéke, ha a végrehajtás alatt álló alprogram visszatérési értékét egy listához akarjuk rendelni. Ezzel a függvényhívással az alprogram lényegében meg tudja nézni, hogy milyen környezetben hívták meg őt, és ettől függően akár más-más típusú visszatérési értéke is lehet!

warn LIST

Mint a die, csak nem lép ki a programból.

Előre definiált változók

Ha az awk-os neveket szeretnénk használni, akkor a

use English;

sort kell még beírni a programba.

A file-ok "objektum-szerű" használatához a

use FileHandle;

sort kell beírni. Ezek után a következők ekvivalensek:

print KIMENET "Hello World!!!\n";

KIMENET->print("Hello World!!!\n");

A változókból itt is csak a legfontosabbakat fogom megemlíteni:

\$_, \$ARG

Az alapértelmezett változó. Ha valahol nincs változó, akkor ott ez lesz használva.

\$1, \$2, \$3...

Reguláris kifejezésekből kapott betűcsoportok.

\$&, \$MATCH

Legutolsó mintaillesztésnél az illesztett rész.

\$[

Ez a változó mutatja, hogy a tömbök hányadik elemén kezdődik adat. Ez állítható a különböző nyelvekben megszokotthoz igazodva.

\$`, \$PREMATCH

Legutolsó mintaillesztésnél a \$& előtti rész.

\$', \$POSTMATCH

Legutolsó mintaillesztésnél a \$& utáni rész.

\$., \$NR

Utolsó olvasási műveletnél az olvasott sor sorszáma. (Ez csak olvasható változóként kezelendő!)

\$/, \$RS, \$INPUT_RECORD_SEPARATOR

Input rekordok elválasztása. Ez alapesetben az újsor karakter, de bármire lecserélhető. Ha az üres stringet adjuk meg, akkor üres sorok lesznek a határok. Ez nem ugyanaz mint `$/ = "\n\n"`; mert a `$/ = "\n\n"`; több üres sort is egyetlen határnak tekint.

\$|, \$OUTPUT_AUTOFLUSH

Output bufferelését szünteti meg, ha az értéke nem egy.

\$\, \$ORS, \$OUTPUT_RECORD_SEPARATOR

Az a karakter, amit a print ki fog írni minden sor végén. Ez alapesetben üres.

\$?, \$CHILD_ERROR

Utolsó gyermek process visszatérési értéke. Ez a wait() függvény visszatérési értéke, tehát a valódi exit() értéket (`$? >> 8`)-ként kaphatjuk meg.

\$\$, \$PID, \$PROCESS_ID

A process azonosító száma.

\$<, \$UID, \$REAL_USER_ID

Valódi user azonosítója.

\$>, \$EUID, \$EFFECTIVE_USER_ID

A futás közbeni jogok tulajdonosa. Ez az érték csak a setuid programoknál vált át a file tulajdonosának jogaira. Ez persze írható változó, tehát a `$> = 0`; a root-tá válás egy módja, csak ez nem mindig fog bejönni :-).

\$(, \$GID, \$REAL_GROUP_ID

Valódi csoport azonosítója. Ha a rendszer több csoportot is támogat egyszerre, akkor ez egy listája azoknak a csoportoknak, amiben a process benne van.

\$), \$EGID, \$EFFECTIVE_GROUP_ID

Effektív csoport azonosítója. Ez setgid program futtatásakor különbözhet az előzőtől.

\$0, \$PROGRAM_NAME

A programot indító parancs neve (egy Perl script-nál a script neve, nem a "perl" szó).

@ARGV

A parancssori argumentumok listája.

@INC

Azon könyvtárak listája, ahol a Perl elkezd keresgélni egy modul után.

%INC

A használt modulok tömbje (filenév, elérési-út) elemekkel.

%ENV

Környezeti változók tömbje, pl.:

```
print "Otthonom: ", $ENV{"HOME"}, "\n";
```

%SIG

Kivételkezelők tömbje.

```
sub handler { # az első paraméter a signal neve
    local($sig) = @_;
    print "Elkaptam $sig-t!\n";
    exit(0);
}
$SIG{'INT'} = 'handler';
$SIG{'QUIT'} = 'handler';
...
$SIG{'INT'} = 'DEFAULT'; # alapértelmezett
$SIG{'QUIT'} = 'IGNORE'; # figyelmen kívül hagy
```

Néhány Perl-beli esemény is kezelhető így. A `$SIG{__WARN__}` a `warn` által kiváltott, a `$SIG{__DIE__}` pedig a `die` által kiváltott esemény lekezelésére szolgál. Mindkét esetben átadásra kerülnek a `warn`, illetve a `die` paraméterei.

Alprogramok írása

Alprogramok deklarálása:

```
sub NEV;      # a NEV ismertté tétele
sub NEV BLOCK # deklarálás és definíció
```

Egy alprogramot nem kell előre deklarálni ahhoz, hogy használhassuk. Az alprogramoknál a paraméterlistát sem kell deklarálni, mert ez változhat. A hívott alprogram a paramétereket a `@_` listán keresztül kapja meg. Az alprogram utolsó utasításának az értéke lesz a visszatérési érték, ha csak nincs egy `return` utasításban más megadva.

```
sub max {
    my $max = pop(@_);
    foreach $elem (@_) {
        $max = $elem if $max < $elem;
    }
    $max;
}
```

Alprogram hívása:

```
&NEV;          # az aktuális @_ -t adja tovább
&NEV(LISTA);   # Ha &-t írunk, akkor () kötelező
NEV(LISTA);    # & elhagyható ha van () vagy már deklarált
NEV LISTA;     # () elhagyható, ha a NEV már deklarált
```

Alprogram hívása akkor megy ilyen egyszerűen ha az az őt tartalmazó modulban látható. Ha ettől eltérő modulban lett deklarálva, akkor modulhivatkozással, vagy valamilyen objektum-orientált technikával kell meghívni a kívánt eljárást (ld. később).

```
$m = &max(1,2,3); # a hatás ugyan az
$m = max 1 2 3;
```

Névtelen alprogram létrehozása:

```
$alpref = sub BLOCK;      # deklarálás
&$alpref(1,2,3);          # hívás
```

Ez a technika főleg egyszer használatos alprogramoknál lehet hasznos, például egy signal kezelő átdefiniálásakor.

Névtelen alprogramok definiált környezete

A névtelen alprogramokra jellemző, hogy mindig abban a környezetben futnak, amelyben definiálták őket, még akkor is, amikor az adott környezeten kívülről kerülnek meghívásra. Ez egy érdekes módja lehet a paraméterátadásnak, és callback jellegű kódrészeket lehet vele írni.

```
sub newprint {
    my $x = shift;
    return sub { my $y = shift; print "$x, $y!\n"; };
}
```

```
$h = newprint("Hello");
$g = newprint("Üdvözet");
# Valamivel később...
&$h("világ!");
&$g("mindenkinek");
```

És az eredmény:
Hello, világ!
Üdvözet, mindenkinek!

Beépített függvények átlapolása

Sok beépített függvény átlapolható, de ezt csak akkor érdemes kipróbálni, ha jó okunk van rá. Ilyen eset lehet a nem Unix-rendszereken a környezet emulálása. Az átlapolás a subs pragma segítségével történhet, például így:

```
use subs 'chdir';      #az átlapolandó beépített függvények
chdir $valahova;      #az új chdir használata
sub chdir { ... }     #a chdir megvalósítása
```

Modulok

package

A Perl nyelv lehetőséget ad különböző láthatósági körök használatára. Ezeket a láthatósági köröket moduloknak nevezhetjük, amelyet a package kulcsszó vezet be. A package hatása az adott blokk végéig, vagy a következő package-ig tart. Alap esetben egy egyszerű programban minden a main modulba kerül be. Ha leírjuk a package szót, akkor az ez után következő deklarációk már az új modulba fognak tartozni. A modul neveihez a :: hivatkozás operátorral férhetünk hozzá (ez régen egy ' jel volt, de az egyszerűbb olvashatóság érdekében, meg a C++ programozók kedvéért ez megváltozott).

```
$elso = 1;      # ez a $main::elso
package MODUL;  # új modul kezdete
$masodik = 1; # $MODUL::masodik
$elso = 1;      # $MODUL::elso
$main::elso = 2;# $main::elso
```

A főmodul neveihez még a \$::elso hivatkozással is hozzáférhetünk.

Szimbólumtábla

A modulok szimbólumtáblái futás közben elérhetőek, sőt módosíthatóak!!! A modulhoz a modul nevével megegyező szimbólumtábla tartozik, ami lényegében egy asszociatív tömb: %main::, avagy %MODULE::. Az itt lévő bejegyzésekhez a *nev alakban is hozzáférhetünk.

```
local(*main::alma) = *main::korte;
local($main::{'alma'}) = $main::{'korte'};
```

Ez a példa egy új álnév létrehozását mutatja. Ezek után minden korte-re alma-ként is hivatkozhatunk. Az egyetlen különbség az, hogy az első fordítási időben értékelődik ki.

Konstruktor, destruktor

Ha a modulban BEGIN, illetve END kulcsszóval jelzett blokkot definiálunk, akkor azok a package használata előtt, illetve után lefutnak.

```

package hibakezeles;
BEGIN {
    open(HIBAK,">./hibak");
}
END {
    close(HIBAK);
}
sub kezeles {
    local ($szoveg) = @_;
    print HIBAK $szoveg, "\n";
}

```

A programban elindított BEGIN blokkokhoz képest fordított sorrendben fognak lefutni az END blokkok. Egy modulban lévő nevekhez a use kulcsszóval férhetünk hozzá:

```

use MODUL;
use hibakezeles kezeles;

```

A use használata ekvivalens a következővel:

```

BEGIN { require MODUL; import MODUL; }

```

Modulokat implementáló file-okat az @INC által meghatározott könyvtárakban keresi a rendszer. A .pm, .pl és .ph kiterjesztéseket nem kell kiírni a filenevek után.

Bonyolultabb struktúrák

A gondok mindig a mutatókkal kezdődnek, de ez itt még álnevekkel is kombinálódik... Perl 4-ben kicsit nehézkes volt a bonyolult adatstruktúrák kezelése, de most már vannak referenciák, így már mindazt a borzalmat el lehet követni, amit egy C programban csak el tudunk képzelni. Sőt még többet is, mert egy C programban nem lehetett a változóneveket menet közben manipulálni.

Referenciák létrehozása

Legegyszerűbb a \ operátor használata. Ezzel egy újabb hivatkozást készíthetünk egy változóra (egy már biztosan van a szimbólumtáblában):

```

$scalarref = \$scalar;
$arrayref = \@ARGV;
$hashref = \%ENV;
$coderef = \&handler;

```

Névtelen dolgokra is lehet hivatkozni:

```

$arrayref = [1, 2, ['a', 'b', 'c']];
$hashref = {
    'Ádám' => 'Éva',
    'Clyde' => 'Bonnie',
};
$coderef = sub { print "Nem nyert!\n"; };

```

Referenciák használata

Nagyon egyszerű egy referenciát használni: a programban egy változó neve helyére bárhova beírhatunk egy megfelelő típusú referenciát tartalmazó változót. Az egyszerű esetek, amikor egy egyszerű struktúrát szeretnénk használni:

```

$ketto = $$scalarref;

```

```

print "A program neve:", $$arrayref[0], "\n";
print "HOME=", $$hashref{'HOME'};
&{$coderef}(1,2,3);

```

Persze lehet mutatni mutatóra is:

```

$refrefref = \\\"valami\";
print $$$$refrefref;

```

Bármilyen hely, ahol egy referenciát kapnánk meg, helyettesíthető egy blokkal, amely értéke a referencia lesz:

```

$ketto = ${$scalarref};
print "A program neve:", ${$arrayref}[0], "\n";
print "HOME=", ${$hashref}{'HOME'};
&{$coderef}(1,2,3);

```

No persze a blokk lehet bonyolultabb is:

```

&{ $inditas{$index} }(1,2,3);      # megfelelő eljárás indul
$$hashref{"KEY"} = "VALUE";      # 1. eset
${$hashref}{"KEY"} = "VALUE";    # 2. eset
${$hashref}{"KEY"} = "VALUE";    # 3. eset
${$hashref->{"KEY"}} = "VALUE";   # 4. eset

```

Itt az 1. és 2., illetve a 3. és 4. eset egyenértékű.

A fenti esetek egyszerűsítésére szolgál a -> operátor:

```

print "A program neve:", $arrayref->[0], "\n";
print "HOME=", $hashref->{'HOME'};

```

Ennek balértéke bármilyen kifejezés lehet, amely egy referenciát ad vissza. Ez az operátor el is hagyható {} vagy [] zárójelek között (de tényleg csak közöttük!):

```

$array[$x]->{"valami"}->[0] = "január";
$array[$x]{"valami"}[0] = "január";

```

Ezzel el is jutottunk a többdimenziós C-beli tömbökhöz:

```

$tomb[42][4][2] += 42;

```

Szimbólikus hivatkozások

Ha a fent említett hivatkozásokban egy blokk nem egy változó referenciájával, hanem egy stringgel tér vissza, a nyelv akkor sem esik kétségbe. Szorgalmasan elkezd böngészni a szimbólumtáblát, hátha talál egy ilyen bejegyzést:

```

$nev = "ize";
$$nev = 1;      # $ize
${$nev} = 2;    # $ize
${$nev x 2} = 3; # $izeize
$nev->[0] = 4;   # $ize[0]
&$nev();        # &ize()
$modulom = "MAS"
${"${modulom}::$nev"} = 5;      # $MAS::ize

```

Ha ez a szabadság nem tetszik nekünk, akkor megköthetjük kezünket a use strict; használatával.

OOP

Amit a Perl objektumokról tudni kell:

Egy objektum csak egy egyszerű referencia, amely történetesen tudja, hogy melyik osztályhoz tartozik. Egy osztály egy package, amely tudja az objektum hivatkozásokat kezelni, és van némi támogatása az öröklésre. Egy metódus az egy egyszerű alprogram, amelynek az első paramétere egy objektum referencia vagy egy package-név lesz.

Objektum

Az objektum bármilyen referencia lehet, ami meg van áldva azzal a tudással, hogy hol jött létre:

```
package ValamiUj;
sub new { bless {} } # 1. verzió
sub new {
    # kicsit bonyolultabban...
    my $self = {};
    bless $self;
    $self->initialize();
    return $self;
}
```

A referencia általában egy asszociatív tömb szokott lenni, amely aztán az objektum saját kis szimbólum táblájaként szolgál. Az objektumokat újra meg lehet áldani. Ekkor az objektum az új osztályba kerül, az eredeti osztályát "elfelejti", hisz egy objektum egyszerre csak egy osztályhoz tartozhat. Ezek után az új osztálynak kell gondoskodnia az eredeti adattagokkal kapcsolatos teendőkről. ld.Destruktorok

Osztály

Az osztály egy package. Nincs semmi különös jelölés arra, hogy ez egy osztály, sőt még az inicializáló eljárást sem kell new-nak hívni. Egy osztályban csak a metódusok öröklésére van támogatás az @ISA tömbbön keresztül. Ez egy modul neveket tartalmazó tömb. Ha egy metódust nem található meg az aktuális package-ban, akkor az ebben a tömbben felsorolt modulok lesznek bejárva a hiányzó alprogramért. Ez egy mélységi keresés lesz. Ha itt sem talál semmit, és van egy AUTOLOAD nevű függvény, akkor megpróbálja ezzel előszedetni a hiányzó eljárást. Ha ez a lehetőség sem járt sikerrel, akkor egy UNIVERSAL-nak nevezett modulban fog keresgélni a rendszer. Az @ISA tömb szépsége az, hogy menet közben is lehet módosítani, azaz menet közben megváltoztathatjuk egy osztály leszármazási fáját! Nyilvánvalóan az adattagok öröklésére is szükség van egy objektum-orientált nyelvben, ez a Perlben az @ISA tömb segítségével megvalósítható.

```
package A;
sub new {
    my $type = shift;
    my $self = {};
    $self->{'a'} = 42;
    bless $self, $type;
}
package B;
@ISA = qw( A );          # A a B ősosztálya
sub new {
```

```

    my $type = shift;
    my $self = A->new;
    $self->{'b'} = 11;
    bless $self, $type;
}
package main;
$c = B->new;
print "a = ", $c->{'a'}, "\n";
print "b = ", $c->{'b'}, "\n";

```

Metódus

Semmi különös jelölés nincsen rájuk, kivéve a destruktorokat. Alapvetően kétfajta metódus van:

statikus metódus

Ez első paraméterében az osztály nevét kapja meg:

```

package ValamiUj;
sub holvagyok {
    my ($neve) = @_;
    print "holvagyok: $neve\n";
}

```

Ez a következőképpen hívható:

```
ValamiUj->holvagyok();
```

"rendes" metódus

Ez az első paraméterében egy referenciát vár.

```

package ValamiUj;
sub new {
    my $self = {};
    bless $self;
    $self->{elso} = 1;    # ez {"elso"}-vel egyenértékű
    $self->{ize} = 42;
    return $self;
}
sub kiir {
    my $self = shift;
    my @keys = @_ ? @_ : sort(keys(%$self));
    foreach $key (@keys) {
        print "\t$key => $self->{$key}\n";
    }
}

```

És ennek hívása:

```

use ValamiUj;
$obj = ValamiUj::new();
$obj->kiir();    # C++ stílus
kiir obj "elso";# "print" stílus

```

Destruktor

Destruktor is definiálható az osztályhoz, ha a modulban megadunk egy DESTROY nevű eljárást. Ez akkor lesz meghívva, amikor az utolsó hivatkozás is megszűnik az adott objektumra, vagy a program futása befejeződik.

Nincs rekurzív destruktork meghívás, vagyis az újraáldott objektum eredeti osztályára vonatkozó destruktort explicit meg kell hívni. Az objektumban tárolt adattagokra azonban a destruktork meghívása megtörténik.

PHP

Ebben a fejezetben a PHP programozási nyelvet, ezt az utóbbi időben egyre jobban elterjedt, szerveroldali szkript nyelvet próbálom meg bemutatni. A PHP a dinamikus, interaktív weboldalak létrehozásának egyik legegyszerűbb és leghatékonyabb eszköze. Használatával elenyésző mennyiségű kódolással egyszerű és hatékony szkripteket illeszthetünk web oldalunkba, amik a legalapvetőbb feladatoktól a legösszetettebb alkalmazásokig gyakorlatilag bármilyen feladat elvégzésére képesek. A PHP azonban egy jelentős ponton eltér az eddig ismert szkript nyelvektől. Szerveroldali szkript nyelv ez is, vagyis a szerveren fut le, azonban a kód maga a HTML-kódba beillesztve található meg. A PHP értelmező felismer a HTML-oldalba illesztett PHP kódot, értelmezi, lefuttatja és eredményét visszaadja a böngészőn keresztül. A könnyebb érthetőség kedvéért tekintsünk egy példát:

```
<html>
<head><title>Első PHP példánk</title></head>
<body>
<hr>
  <?php
    echo "Hello PHP!";
  ?>
<hr>
</body>
</html>
```

A kiemelt rész maga a PHP szkript. Nem túl bonyolult, de ahhoz elég, hogy ráérezzünk, miről is van szó. Ha a fenti HTML-kód keresztülfut a PHP értelmezőn, akkor az észreveszi a <?php tagról, hogy a következőkben nem hagyományos HTML-kódról van szó, hanem PHP programrészletről. A programot az értelmező végrehajtja és az echo utasításnak megfelelően a "Hello Moon" feliratot írja ki. Illetve nem pontosan kiírja, hanem a HTML-kódban a szkript helyére a kimenetet beilleszti. A HTML és a szkript együttes eredménye az alábbi kód lesz:

```
<html>
<head><title>Első PHP példánk</title></head>
<body>
<hr>
Hello PHP!
<hr>
</body>
</html>
```

A PHP rövid története

A PHP története 1994 őszére nyúlik vissza, amikor a munkát kereső Rasmus Lerdorf egy Perl CGI szkriptet használt a web oldalát felkeresők regisztrálására. A látogatókat naplózó kódot "PHP-tools for Personal Home Page"-nek nevezte el. Az első nyilvános változat úgy 1995 táján látott napvilágot. Ez még csak néhány egyszerűbb feladatra volt használható, többek között számlálót, vendégkönyvet tartalmazott.

A PHP fejlesztése a Torontoi Egyetemen folytatódott, ahol Rasmus Lerdorf olyan interfészt fejlesztett ki, aminek segítségével a HTML kódba ágyazott speciális utasítások közvetlenül érték el az egyetemi adatbázisokat. A rendszert Rasmus "Form Interpreter"-nek, FI-nek nevezte el. Az FI-ben használt elv már megegyezett a PHP alapelveivel, miszerint a HTML kódba beágyazott utasításokat értelmezte és hajtotta végre az FI értelmezője. Később a PHP és az FI összeházasításából született meg az első széles körben használt parancsértelmező a PHP/FI. Ez tartalmazta a PHP és az FI addigi szolgáltatásait, sőt az mSQL adatbázisok elérését is támogatta. Rasmus eleinte eljátszódott a gondolattal, hogy a PHP-t kereskedelmi terméké teszi, de olyan komoly mennyiségű visszajelzést kapott más programozóktól, különböző kiegészítéseket és hibajavításokat küldve a PHP-hez, hogy letett ebbéli szándékáról. A PHP fejlődéséhez és sokrétűségéhez nagymértékben hozzájárult külső programozók szabad és ingyenes részvétele a rendszer fejlesztésében. A PHP a mai napig is ingyenes termék, és ez valóban nagyon jó dolog. Az első verzió megjelenésétől kezdve a PHP felhasználói tábora töretlenül növekedett. 1996-ban közel 15.000 web odalon használták a PHP/FI-t, 1997-ben már több mint 50.000 web odalon. Ebben az évben kezdődött el a PHP sokkal jobban szervezett továbbfejlesztése. A PHP/FI-t értelmezőjét szinte az alapoktól kezdve újrairták, átemelve a PHP/FI-ben alkalmazott technikákat és kódot, de számos újat is hozzátéve. Így alakult ki a PHP 3-as változata, ami gyakorlatilag rendelkezett mindazokkal a képességekkel, amik a PHP népszerűségét megalapozták. Ugyanakkor természetes, hogy a PHP fejlődése nem állt meg. Jelenleg a 4.x változatnál tart a fejlesztés, de köszönhetően a "szabad szoftver" filozófiának nem valószínű, hogy itt megreked.

Alapok

A PHP szerveroldali beágyazott nyelv. A parancsok - tulajdonképpen maguk a programok - a szerveren futnak le oly módon, hogy a megfelelő kiterjesztésű állományt a PHP parancsértelmező átfésüli, értelmezi a PHP parancsokat és utasításokat, azokat megfelelő módon végrehajtja, és az esetleges kimeneteket (a példában látott módon) beilleszti a HTML-oldalba. Ez nagyon kellemes dolog, hiszen a felhasználó már a programunk által generált tartalmat látja, és nem magát a PHP programot. Ugyanakkor egyszerűbb feladatokat egy pár soros beillesztett programocskára is el tud látni, a viszonylag sokkal bonyolultabb CGI programok helyett.

A PHP rendkívül sokoldalú nyelv, gyakorlatilag minden olyan eszköz rendelkezésre áll, ami egy hatékony nyelvhez szükséges, bír a CGI programok készítéséhez kellő összes alapvető képességgel, de ezen túlmenően még nagyon sok minden tud. Nézzük meg a PHP legfontosabb lehetőségeit:

Változók rugalmas használata. A változók típusa nincs mereven rögzítve, azt a program futása közben a PHP a változó környezetétől függően állapítja meg. Ugyanakkor széles lehetőségeink nyílnak a változók egymásba történő konvertálására, mindezt természetesen automatikusan. Figyelemre méltó a tömbök rugalmas használata.

Kívülről átvehető változók. HTML-formokból meghívott PHP oldalak, a formokban felhasznált változókat egyszerűen, nevükkel történő hivatkozással vehetik át.

Hatékony filekezelés. Seregnyi filekezelő, fileinformációs és könyvtárkezelő függvény teszik lehetővé a fileokkal végzendő munkát.

Képek készítése és manipulációja. A PHP olvassa a GIF, JPEG és PNG állományokat, ezeket képes manipulálni. A manipulált vagy az újként előállított grafikus állományokat JPEG és a PNG formátumban képes kiírni, de lehetséges olyan PHP programok készítése, amik a statikus grafikus állományokhoz hasonlóan használhatók, tartalmukat azonban dinamikusan állítják elő.

Filefeltöltés kezelése. Lehetséges a felhasználó gépéről fileok feltöltését kezdeményezni a PHP programot futtató szerverre.

Távoli fileok elérése. Lehetőségünk van HTTP címek megnyitására, gyakorlatilag a helyi fileok megnyitásával azonos módon.

Reguláris kifejezések használata. A Perl nyelvben használatos reguláris kifejezések használhatók a PHP programokban a stringekben történő keresésekhez, karaktercserékhez, karakterrészletek másolásához. Bár kezdők részére a reguláris kifejezések használata eleinte okozhat bonyodalmakat, érdemes később elmerülni bennük, mert igen hasznos eszközök.

Adatbázisok elérése. A PHP egyik legfontosabb tulajdonsága az adatbázisrendszerek igen széles körének használhatósága. A teljesség igénye nélkül itt van néhány a támogatott adatbázisok közül: MySQL, mSQL, dBase, Oracle, Sybase, PostgreSQL, Ingres.

Változók PHP-ban

A PHP nyelvben a változókat a nevük elé tett '\$' jel jelöli. Az alábbi alapvető változótípusokat különböztetjük meg: egész számok, stringek és lebegőpontos változók. A változók legegyszerűbben valamilyen értékadással kaphatnak értéket, valahogy így:

```
$a = 10;  
$b = 'Ez most egy string';  
$pi = 3.1415;
```

Természetesen a változókkal az összes ismert és az adott változóra értelmezhető művelet elvégezhető, azonban léteznek különlegesebb operátorok is, illetve a változók értékeadásánál van még néhány érdekesség:

```
$c = ($a = 10);
```

Az értékadásnak önmagában is értéke van, vagyis a fenti esetben \$c változó ugyanazt az értéket kapja, amit \$a kapott meg. Ennek így a fenti esetben nem sok értelme van, de később látni fogjuk, hogy sok esetben hasznos lehet ez a fajta értékadás.

```
$c += 10;
```

A kifejezés megegyezik ezzel: $c = c + 10$; de hasonlóan működik a kivonás szorzás, osztás és a stringek összefűzése esetén.

```
++$c;
```

Az inkrementáló (növelő) operátor a változó értékét növeli meg eggyel. Hasonlóképpen működik csökkentésre $--c$; formában. Ezt az operátort leginkább ciklusokban használjuk gyakran valamely változó értékének léptetésére. A $++c$; kifejezést felírhatnánk így is $c++$; azonban ez nem egészen ugyanaz. Amennyiben a változó előtt van a növelés operátora bármely egyéb értékadást megelőző a növelés, egyébként nem. Ha \$c értéke 10, akkor a $a = (++c)$; értékadást követően \$c értéke 11 lesz és \$a értéke is. Ugyanakkor, a $a = (c++)$; értékadás hatására \$a felveszi \$c értékét, ami 10, majd \$c értéke megnövekedik, így 11 lesz.

Külön szót érdemes ejteni a stringek összefűzéséről:

```
$a = 'alma';  
$b = ' a fa alatt';  
$c = $a.$b;
```

\$c értéke 'alma a fa alatt' lesz. A pont operátor szolgál a stringek összefűzésére, ne használjuk az összeadás operátort ilyen esetekben. Viszont használható a $c .= a$ kifejezés a számokhoz hasonlóan, amikor is a kifejezés jobb oldalán lévő értéket a kifejezés bal oldalán állóhoz fűzi hozzá.

Mint már előbb utaltam rá, a változók típusa a PHP-ben nincsen rögzítve, azok a körülményekhez képest megváltozhatnak. Természetesen van rá mód, hogy a változók típusát pontosan beállítsuk - erre szolgálhat a `settype()` függvény - de az értékadásnál is pontosan definiálhatjuk a változó típusát, ha $c = (real) 10$; formában tesszük azt. Az értékadásnál használható típusok az alábbiak lehetnek:

(int) vagy (integer) - egész számok meghatározására

(real), (double), (float) - lebegőpontos számok meghatározására

(string) - szöveges változók meghatározására

(array) - tömbök meghatározására

(object) - objektumok meghatározására

A változók típusának rugalmasságára az alábbi példa mutathat rá:

```
$a = '10 körte';  
$b = 20;  
$c = $a + $b;
```

A példában szereplő \$c értéke 30 lesz, mert \$a-t a PHP számként értelmezi, gondolván azt akartuk. A változók típusának automatikus konverziója eleinte kicsit furcsa lehet olyanoknak, akik a változótípusokat szigorúan meghatározó nyelvekhez szoktak, de rövid idő után magától értetődően használhatjuk ezt az igen hasznos lehetőséget.

Sokszor hasznos lehetősége a PHP-nek a változó változók használata. Ilyen esetben a változó nevét, dinamikusan, egy másik változó értékéből eredeztetjük. Ha \$gyumolcs változó értéke "körte", és \$nev értéke "gyumolcs", akkor a \$\$nev változó értéke meg fog egyezni a \$gyumolcs változó értékével.

A PHP nyelvben egy egész sor függvény és utasítás kapcsolatos a változókval. Ezek a PHP referenciákban megtalálhatók, azonban itt kiemelek néhány fontosabbat.

Empty() - logikai függvény, mely igaz értékkel tér vissza, ha a változó értéke nulla, üres string, vagy a változó nincsen beállítva. Ezek szerint, ha \$a=10, akkor empty(\$a) értéke hamis lesz, ellenben ha \$b értéke 0 vagy \$b-nek még nem adtunk egyáltalán értéket, akkor empty(\$b) értéke igaz lesz.

Gettype() - visszadja egy változó típusát.

Print_r() - ember számára olvasható információt ad egy változó értékéről. Rendkívül hasznos függvény olyan esetben, amikor a változóink értékét a tesztelés során nyomon akarjuk követni, de tömbökről is nagyon jól áttekinthető képet ad. Később látható még példa a print_r() függvény használatára.

Az egyszerű skaláris változókat követően nézzük meg a tömböket.

Tömbök a PHP-ban

A tömbök, azok különböző lehetőségei és az azok köré felsorakoztatott függvények a PHP programozás egyik legerőteljesebb eszközrendszerét alkotják, ugyanakkor rendkívül egyszerűen és könnyedén használhatók. A tömb változók halmaza, melyeket a tömbön belül sorban tárolhatunk és a teljes adathalmazt egyszerre is kezelhetjük, ugyanakkor a tömb elemeihez külön-külön is hozzáférhetünk. Fontos tulajdonsága a tömböknek, hogy egy tömbön belül az elemek típusa különböző lehet. Egy tömb elemeit legegyszerűbben explicit módon, elemenként tölthetjük fel:

```
$tomb[1] = "dBase";  
$tomb[2] = "FoxPro";  
$tomb[4] = "Clipper";  
$tomb[5] = 42;
```

Látható, hogy a tömb elemeinek megadásakor nem szükséges a sorrendiséget szigorúan betartani.

Egy tömb elemeihez a fentieknél egyszerűbben is, a tömbindex használata nélkül is lehet elemeket adni:

```
$tomb[] = "Basic";  
$tomb[] = "FoxPro";
```

Ily módon a tömb végéhez kapcsolódnak az új elemek, az index értéke pedig az legutolsó indexelemnél eggyel magasabb lesz. Hasonlóan működik az array_push() függvény, azzal a különbséggel, hogy egy utasításon belül több értéket is hozzáfűzhetünk a tömbhöz:

```
array_push($tomb, "Cobol", "Fortran");
```

Szép lassan dagadó tömbünk a fenti utasításokat követően már így néz ki:

```
Array
(
    [1] => dBase
    [2] => FoxPro
    [4] => Clipper
    [5] => 42
    [6] => Basic
    [7] => FoxPro
    [8] => Cobol
    [9] => Fortran
)
```

Természetesen a tömbök értékeinek megadásához hasonlóan férhetünk hozzá a tömbelemekhez, azonban a fent említett `array_push()` függvény párja, az `array_pop()` függvény is rendelkezésünkre áll, mely azonban nemcsak egyszerűen a tömb utolsó elemét adja vissza értékül, hanem a tömb elemeinek számát is csökkenti az utolsó elemmel:

```
$nyelv1 = $tomb[1];
// $nyelv1 értéke "dBase"
$nyelv2 = $tomb[4];
// $nyelv2 értéke "FoxPro"
$nyelv9 = array_pop($tomb);
// $nyelv9 értéke "Fortran" és a tömb nyolc elem• lesz
```

Bonyolítsuk egy kicsit a dolgokat. Ezidáig a tömbünk egy dimenziós volt, azonban a PHP nyelvben a tömbök kettő vagy akár több dimenziósak is lehetnek. Az értékadás legegyszerűbb módja ilyen esetben is az explicit értékadás:

```
$auto[1][1] = "Maserati";
$auto[1][2] = "olasz";
$auto[2][1] = "Renault";
$auto[2][2] = "francia";
$auto[3][1] = "Mercedes";
$auto[3][2] = "német";
```

a tömb valahogyan így fog kinézni:

```
Array
(
    [1] => Array
        (
            [1] => Maserati
            [2] => olasz
        )
    [2] => Array
        (
            [1] => Renault
            [2] => francia
        )
    [3] => Array
        (
            [1] => Mercedes
            [2] => német
        )
)
```

Ilyen és ehhez hasonló tömbök létrehozására, azonban sokkal tömörebb és olvashatóbb módszer az `array()` függvény használata. Ez a függvény a paraméterként megadott értékeket tömb formában adja vissza. Így a fenti értékadással pontosan megegyező eredményt ad a következő:

```
$auto[1] = array( "Maserati" , "olasz" );
$auto[2] = array( "Renault" , "francia" );
$auto[3] = array( "Mercedes" , "német" );
```

Ahogy azonban a tömbelemek típusaira vonatkozóan nincsenek túl szigorú megkötései a PHP nyelvnek, ugyanúgy nem kezeli szigorúan a többdimenziós tömbök elemszámait sem a PHP. Az alábbi értékadás teljesen helyes eredményt ad:

```
$auto[1] = array( "Maserati" , "olasz" );
$auto[2] = array( "Renault" , "francia" , "406" , "206" );
$auto[3] = array( "Mercedes" , "német" , "E320",
                 "Vito" , "Sprinter kisteherautó" );
```

Természetesen az `array_pop()` és az `array_push()` függvények az `array()` függvénnyel ötvözve több dimenziós tömbök esetén is használhatók.

```
array_push( $auto, array("Citroen" , "francia" , "ZX" , "Xsara");
```

A fenti esetekben a tömb elemei azok sorszámaival voltak azonosítva. A PHP ismeri az asszociatív tömbök fogalmát is. Az asszociatív tömbök rendkívül hasznos és sokoldalú elemei a PHP nyelvnek. A PERL nyelvben használt hash típusú tömbökhöz hasonlóan működnek. A tömbelemekre való hivatkozás ilyen esetben nem sorszámmal, hanem egy indexelem (kulcs) segítségével történik, egyszerűen úgy, hogy a sorszám helyére, az indexelemet helyezzük.

```
$tomb["els•"] = "Kis Gedeon";
$tomb["második"] = "Nagy Elemér";
```

Függetlenül attól, hogy a tömb elemeinek milyen sorrendben adtunk értéket, az elemeket az indexkulcs segítségével érhetjük el, és ez nem függ attól, ha a tömbhöz hozzáfűzünk, vagy attól elveszünk egy elemet. Új elem bármikor hozzáfűzhető a tömbhöz:

```
$tomb["harmadik"] = "Kukonya Berkó";
```

Az asszociatív tömbök lehetnek egydimenziósak, mint a fenti példában, de lehetnek több dimenziósak is. A fenti példát kibővíthetjük több dimenzióssá:

```
$tomb["els•"]["neve"] = "Kis Gedeon";
$tomb["els•"]["kora"] = 27;
$tomb["második"]["neve"] = "Nagy Elemér";
$tomb["második"]["kora"] = 22;
```

Ha a "Nagy Elemér" értékű elemet a `$tomb["második"]["neve"]` hivatkozással tudjuk elérni, de ha `$sorszam` értéke "második" akkor akár `$tomb[$sorszam]["neve"]` hivatkozással is elérhetjük a keresett elemet.

A normál és az asszociatív tömbök létrehozására egyaránt használható az `array()` függvény, amit leginkább tömbök kezdő értékfeltöltése során használhatunk, egy értékadással kiküszöbölve többet. A fenti példákkal megegyezők az alábbi értékadások:

```
$tomb = array ( "els•"      => "Kis Gedeon",
               "második" => "Nagy Elemér" );
$tomb = array ( "els•"      => array ( "neve" => "Kis Gedeon",
                                       "kora"  => 27 ),
               "második" => array ( "neve" => "Nagy Elemér",
                                       "kora"  => 22 ) );
```

Mint az alábbi példa is mutatja, az értékadás esetén az index értékét nemcsak konkrétan, hanem változóval is megadhatjuk, így már meglehetősen rugalmasan tölthetjük fel tömbjeinket adatainkkal. A következő példa megmutatja a `print_r()` függvény használatát is, amit tetszőleges változó értékének kiírásához használhatunk, de mivel tömbváltozó esetében a komplett tömbstruktúrát is megjeleníti leginkább tesztelési célokra használható nagyon jól.

```
<?php
```

```
$nick1 = "X";
$nick2 = "Z";
$tomb = array (
    $nick1 => array("nev" => "X Y",
                  "email" => "xy@valami.hu"),
    $nick2 => array("nev" => "Z",
                  "email" => "z@bme.hu")
);
echo( "<PRE><b>" );
```



```

print_r($tomb);
echo("<HR>");
print_r($tomb["X"][$"nev"]);
echo("</b></PRE>");
?>

```

A program kimenete a következő lesz:

```

Array
(
    [X] => Array
        (
            [nev] => X Y
            [email] => xy@valami.hu
        )

    [Y] => Array
        (
            [nev] => Z
            [email] => z@bme.hu
        )
)

```

Asszociatív tömbök esetében azonban figyelemmel kell lenni arra, hogy ilyen tömb elemeit kizárólag a meghatározott indexértékkel érhetjük el, a tömb sorszámával nem. Ennek rendkívül egyszerű az oka. Az egyszerű sorszámozott tömb is asszociatív tömb, ahol a tömbindex maga a sorszám. Sőt egy tömbön belül keverhetjük is a sorszámozott és az indexelt elemeket, de azért ezt kerüljük, csak gondot okozunk magunknak. A normál és az asszociatív típusú tömbök a PHP programozás során rendkívül változatosan és hatékonyan használhatók, főleg akkor, ha tudjuk azt, hogy a PHP a tömbök elemeire, az elemszámokra és a tömbelemek típusaira vonatkozóan rendkívül szabad kezdet ad nekünk:

többszörös tömbön belül az egyik index lehet asszociatív, a másik normál

többszörös tömb esetében a tömbelem tömböknek nem kell feltétlenül azonos elemszámúaknak lenni, vagyis \$tomb[1] lehet öt elemű, míg \$tomb[2] lehet akár 8 elemű is.

egyszörös tömbök esetében a tömbelemek lehetnek különböző típusú adatok, de még többszörös tömbök esetében sem kell a tömbelem tömbök adatszerkezetének megegyeznie.

Vagyis elég nagy szabadsággal használhatjuk a tömbváltozókat, mégis érdemes szem előtt tartani, hogy ha lehet járjunk el következetesen a változók értékadásával és azok használatával.

Változók hatásköre

A változó hatásköre az a környezet, amelyben a változó definiált. A legtöbb esetben minden PHP változónak egyetlen hatásköre van. Ez az egyetlen hatáskör kiterjed az include és a require segítségével használt fájlokra is. Például:

```

$a = 1;
include "b.inc";

```

Itt az \$a változó elérhető lesz az beillesztett b.inc szkriptben is. A felhasználói függvényekkel a lokális függvényhatáskör kerül bevezetésre. Alapértelmezés szerint minden, függvényen belül használt változó ebbe a lokális függvényhatáskörbe tartozik, például:

```

$a = 1; /* globális hatáskör */
function Test ()
{
    echo $a; /* egy helyi változót vár */
}
Test();

```

Ez a szkript nem fog semmilyen kimenetet sem eredményezni, mivel az echo kifejezés az \$a változónak egy helyi - függvényen belüli - változatára utal, és ebben a hatáskörben ehhez nem

rendeltek értéket. Ez valamelyest különbözik a C nyelv filozófiájától, ahol a globális változók automatikusan elérhetők bármely függvényből, feltéve ha a függvényben újra nem definiáltad azt a változót. Ez problémák forrása lehet, ha az ember véletlenül megváltoztat egy globális változót. A PHP-ben a globális változókat global kulcsszóval kell deklarálni a függvényekben, például:

```
$a = 1;
$b = 2;
function Osszead()
{
    global $a, $b;
    $b = $a + $b;
}
Osszead();
echo $b;
```

A fenti szkript kiírja, hogy "3". \$a és \$b global-ként való deklarálásával minden utalás ezekre a változókra a globális változót fogja érinteni. Nincs megkötve, hány globális változót kezelhet egy függvény. Globális változók elérésének másik módja a PHP által definiált speciális \$GLOBALS tömb használata. Az előbbi példával egyenértékű megoldás:

```
$a = 1;
$b = 2;
function Osszead()
{
    $GLOBALS["b"] = $GLOBALS["a"] + $GLOBALS["b"];
}
Osszead();
echo $b;
```

A \$GLOBALS asszociatív tömb, ahol a globális változó neve jelenti a kulcsot, és a változó értéke a tömbelem értéke.

A változók hatáskörének másik fontos lehetősége a static (statikus) változó. A statikus változó csak lokális hatáskörben él - egy függvényen belül, de két függvényhívás közt nem veszti el az értékét, a változó hatásköréből való kilépés esetén is megmarad az értéke:

```
function Test()
{
    $a = 0;
    echo $a;
    $a++;
}
```

Ez nagyon haszontalan függvény, mivel nem csinál mást, mint minden meghívásakor \$a-t 0-ra állítja, aztán kiírja a 0-t. Az \$a++ teljesen felesleges, mert amint vége a függvény futásának az \$a változó megszűnik. Ahhoz, hogy ebből értelmes számlálófüggvény legyen - megmaradjon a számláló értéke -, az \$a változót statikusnak kell deklarálni:

```
Function Test()
{
    static $a = 0;
    echo $a;
    $a++;
}
```

Most már valahányszor meghívódik a Test() függvény, kiírja \$a értékét, majd azt megnöveli eggyel.

Fejezet. Konstansok

A konstans egy egyszerű érték azonosítója (neve). Mint ahogy az elnevezése is mutatja, a program futása során nem változik meg az értéke (a mágikus __FILE__ és __LINE__ konstansok

az egyedüli kivételek ez alól). A konstansok alapesetben érzékenyek a kis- és nagybetűs írásmódra. Megállapodás szerint általában csupa nagybetűs neveket adunk a konstansoknak.

Konstansok

A konstansok neveire a PHP más jelzőivel azonos szabályok vonatkoznak. Egy érvényes konstans név betűvel vagy aláhúzással kezdődik, amit tetszőleges számú betű, szám vagy aláhúzás követ. A konstansok bárhol elérhetőek. Konstanst a `define()` függvénnyel lehet létrehozni. Definiálása után később nem lehet törölni vagy megváltoztatni az értékét. Csak skaláris adat (boolean, integer, double vagy string típusú) lehet egy konstans tartalma. A konstans értékére a nevének megadásával lehet hivatkozni. A változókkal ellentétben nem szabad \$ jelet tenned a konstans neve elé. Használható még a `constant()` függvényt is, ha például a konstans nevét egy változó adja. A `get_defined_constants()` függvénnyel lehet a definiált konstansok listáját megkapni.

Megjegyzés: A konstansok és a (globális) változók különböző névtérben vannak. Ez azt jelenti, hogy a `TRUE` és a `$TRUE` két különböző dolgot jelent. Ha egy definiálatlan konstanst próbálsz meg használni, a PHP a konstans nevét veszi karaktersorozatként értékül. Ilyen esetekben egy notice szintű hiba keletkezik. A `defined()` függvény segítségével vizsgálható a konstans létezése.

```
<?php
define("KONSTANS", "Helló világ!");
echo KONSTANS; // kiírja, hogy "Helló világ!"
echo Konstans; // kiírja, hogy "Konstans" és hibát eredményez
?>
```

Kommentek

A PHP támogatja a 'C', 'C++' és Unix shell-szerű kommenteket. Például:

```
<?php
echo "Ez egy teszt"; // Ez egy egysoros c++ szerű komment
/* Ez egy többsoros komment
   Még egy sor komment */
echo "Ez egy másik teszt";
echo "Ez az utolsó teszt"; # Ez egy shell-szerű komment
?>
```

Az "egysoros" kommentek valójában csak a sor végéig, vagy az aktuális PHP kód végéig tartanak, attól függően, hogy melyik jön előbb.

```
<h1>Ez egy <?php# echo "egyszerű";?> példa.</h1>
<p>A fenti fejléc kiírja 'Ez egy egyszerű példa'.
```

Operátorok

Operátorok precedenciája

Az operátorok precedenciája azt határozza meg, hogy milyen "szorosan" köt össze két kifejezést. Például az $1 + 5 * 3$ kifejezésben, a kifejezés értéke 16, és nem 18, mert a szorzás operátorának, a `(*)`-nak nagyobb precedenciája van, mint az összeadásénak `(+)`. Zárójelek segítségével tetszőleges precedenciát lehet felállítani egy kifejezésen belül, ha szükséges. Például a $(1 + 5) * 3$ eredménye 18 lesz.

Az alábbi táblázat az operátorokat precedenciájuk szerint növekvő sorrendben tartalmazza.

Táblázat 10-1. Operátorok precedenciája

asszociativitás	operátorok
-----------------	------------

balról jobbra	,
balról jobbra	or
balról jobbra	xor
balról jobbra	and
jobbról balra	print
balról jobbra	= += -= *= /= .= %= &= = ^= ~= <<= >>=
balról jobbra	? :
balról jobbra	
balról jobbra	&&
balról jobbra	
balról jobbra	^
balról jobbra	&
nem köthető	== != === !==
nem köthető	< <= > >=
balról jobbra	<< >>
balról jobbra	+ - .
balról jobbra	* / %
jobbról balra	! ~ ++ -- (int) (float) (string) (array) (object) @
jobbról balra	[]
nem köthető	new

Aritmetikai operátorok

Táblázat 10-2. Aritmetikai operátorok

Példa	Név	Eredmény
\$a + \$b	Összeadás	\$a és \$b összege
\$a - \$b	Kivonás	\$a és \$b különbsége
\$a * \$b	Szorzás	\$a és \$b szorzata
\$a / \$b	Osztás	\$a és \$b hányadosa
\$a % \$b	Modulus	\$a / \$b maradéka

Az osztás operátor ("/") egész értékkel tér vissza (egy egész osztás eredményeképpen) ha a két operandusa egész (vagy string, ami egészszé konvertálódott) és a hányados is egész. Ha valamelyik operandus lebegőpontos szám, vagy az osztás eredménye nem egész, egy lebegőpontos szám a visszatérési érték.

Hozzárendelő operátorok

Az alapvető hozzárendelő operátor az "=". Elsőre azt hihetnénk, hogy ez az "egyenlő valamivel" jele. Valójában azt jelenti, hogy a bal oldali operandus [ami az egyenlőségjel bal oldalán áll] a jobb oldali kifejezést kapja értékül.

A hozzárendelő kifejezés értéke a bal oldalhoz rendelt érték. Vagyis a "\$a = 3" értéke 3. Ez lehetőséget ad néhány trükkös dologra:

```
$a = ($b = 4) + 5; // $a most 9, és $b 4
```

Az alapvető hozzárendelő operátoron felül vannak ún. "kombinált" operátorok is az összes kétoperandusú aritmetikai és sztring operátorok számára, amelyek lehetővé teszik, hogy használjunk egy változót egy kifejezésben, majd rögtön be is állítsuk a változót a kifejezés értékére. Például:

```
$a = 3;
$a += 5; // $a-t 8-ra állítja, mintha $a = $a + 5;-öt írtunk volna
$b = "X ";
$b .= "Y"; // $b "X Y" lesz, egyenértékű párja: $b = $b . "Y";
```

A hozzárendelés az eredeti változót az újba másolja érték szerint, így az egyiken elvégzett változtatások a másikat nem érintik. Ezt fontos tudni, például egy sokszor végrehajtott ciklus belsejében nagy tömbök másolásakor. A PHP 4 támogatja a \$var =&\$othervar; szintaxisú referencia szerinti érték hozzárendelést is, de ez PHP 3-ban nem működik. A 'referencia szerinti értékhozzárendelés' azt jelenti, hogy mindkét változó ugyanarra az adatra fog mutatni, és nem történik meg a változó értékének lemásolása.

Bitorientált operátorok

A bitorientált operátorok teszik lehetővé, hogy egész típusú számokon belül bizonyos biteket beállítsunk, vagy lefedjünk (maszkolás). Ha viszont az operátoron mindkét oldalán sztring típusú változó áll, akkor a bitorientált operátorok a sztringek karakterein dolgoznak úgy, hogy a karakterek ASCII kódjain végzik el a műveletet, és az eredményül adódó számot ASCII kóddal megadott karakternek értelmezi.

```
<?php
    echo 12 ^ 9; // '5' -öt ír ki
    echo "12" ^ "9"; // kiírja a visszaperjel karaktert (ASCII #8), mert
                        // ('1' (ASCII #49)) ^ ('9' (ASCII #57)) = (ASCII #8)
    echo "hallo" ^ "hello"; // eredmény: #0 #4 #0 #0 #0
                        // 'a' ^ 'e' = #4
?>
```

Táblázat 10-3. Bitorientált operátorok

Példa	Név	Eredmény
$\$a \& \b	És	Ott lesz '1' az eredményben, ahol \$a és \$b mindegyikében az a bit '1'-es. Minden más biten '0'.
$\$a \b	Vagy	Ott lesz '1' az eredményben, ahol \$a és \$b közül legalább az egyik azon a bitje '1'-es. Minden más biten '0'.
$\$a \wedge \b	Kizáró vagy	Ott lesz '1' az eredményben, ahol \$a és \$b közül csakis pontosan az egyikben '1' állt. Minden más biten '0'. [Más közelítésben ott lesz '1' az eredményben, ahol különböző bitek álltak \$a-ban és \$b-ben; megint más közelítésben \$a azon bitjei invertálódnak, amely helyeken \$b-ben '1' áll]
$\sim \$a$	Nem	\$a összes bitjét invertálja
$\$a \ll \b	Eltolás balra	\$a bitjeit \$b számú bittel balra tolja (minden bitnyi eltolás 2-vel való szorzást jelent [amíg el nem fogynak a bitek, utolsó helyen előjelbit van ?!])
$\$a \gg \b	Eltolás jobbra	\$a bitjeit \$b számú bittel jobbra tolja (minden bitnyi eltolás 2-vel való egész-osztást jelent. [Vigyázz, negatív számot inkább ne tolj jobbra!])

Összehasonlító operátorok

Az összehasonlító operátorok, mint nevük is sugallja, két érték összehasonlítására szolgálnak.

Táblázat 10-4. Összehasonlító operátorok

Példa	Név	Eredmény
<code>\$a == \$b</code>	Egyenlő	Igaz (TRUE), ha \$a és \$b értéke egyenlő
<code>\$a === \$b</code>	Azonos	Igaz (TRUE), ha \$a és \$b értéke egyenlő, és azonos típusúak (csak PHP 4)
<code>\$a != \$b</code>	Nem egyenlő	Igaz (TRUE), ha \$a és \$b értékei különböznek
<code>\$a <> \$b</code>	Nem egyenlő	Igaz (TRUE), ha \$a és \$b értékei különböznek
<code>\$a !== \$b</code>	Nem azonos	Igaz (TRUE), ha \$a és \$b értékei vagy típusai különböznek (csak PHP 4)
<code>\$a < \$b</code>	Kisebb mint	Igaz (TRUE), ha \$a szigorúan kisebb, mint \$b
<code>\$a > \$b</code>	Nagyobb mint	Igaz (TRUE), ha \$a szigorúan nagyobb, mint \$b
<code>\$a <= \$b</code>	Kisebb, vagy egyenlő	Igaz (TRUE), ha \$a kisebb, vagy egyenlő, mint \$b
<code>\$a >= \$b</code>	Nagyobb, vagy egyenlő	Igaz (TRUE), ha \$a nagyobb, vagy egyenlő, mint \$b

A feltételes "?" (ternális) operátor ugyanúgy működik, mint C-ben és sok más nyelvben.

```
(kif1) ? (kif2) : (kif3);
```

A kifejezés *kif2*-t értékeli ki, ha *kif1* igaznak bizonyul (**TRUE**), és *kif3*-at, ha *kif1* hamis (**FALSE**).

Hibakezelő operátorok

A PHP egy hibakezelő operátort támogat, az `@` jelet (@ - kukac). PHP kifejezés elé írva a kifejezés által esetlegesen generált hibaüzenete(ke)t figyelmen kívül hagyja a rendszer.

Ha a `track_errors` szolgáltatás be van kapcsolva, bármilyen a kifejezés által generált hibaüzenet a `$php_errormsg` globális változóba kerül tárolásra. Ez a változó minden hiba esetén felülíródik, ezért használható információk kinyerése érdekében a kifejezést követően ezt minél hamarabb ellenőrizni kell.

```
<?php
/* Szándékos állomány megnyitási hiba */
$file = @file ('nem_letezo_allomany') or
    die ("Nem lehet megnyitni, a hiba: '$php_errormsg'");
// bármilyen kifejezésre működik, nem csak függvényekre
$ertek = @$tomb[$kulcs];
// nem ad notice szintű hibát, ha a $kulcs kulcs nem létezik
?>
```

Megjegyzés: A `@` operátor csak kifejezésekre működik. Egyszerű ökölszabályként alkalmazandó, ha valaminek az értelmezett az értéke, akkor az elé a `@` operátor is oda tehető. Ekképpen például használható változók, függvények és `include()` hívások, állandók neve előtt és sok más esetben. Nem használható azonban függvény és osztály definíciók vagy nyelvi szerkezetek (mint például `if` és `foreach` utasítások) előtt.

Végrehajtó operátorok

A PHP-ban létezik egy program-végrehajtó operátor: a visszaidézőjel (```). Ezek nem szimpla idézőjelek! A PHP megpróbálja a sztring tartalmát parancssorból futtatandó utasításként végrehajtani, amelynek a kimenete lesz az operátor értéke. Ez nem egyszerűen a kimenetre kerül, hanem hozzárendelhető egy változóhoz.

```
$output = `ls -al`;
echo "<pre>$output</pre>";
```

Növelő/csökkentő operátorok

A PHP támogatja a C-szerű ún. elő- és utónövekményes ill. csökkentő operátorokat.

Táblázat 10-5. Növelő/csökkentő operátorok

Példa	Név	Hatás
<code>++\$a</code>	előnövekményes	Növeli \$a-t eggyel, majd visszaadja \$a értékét
<code>\$a++</code>	utónövekményes	Visszaadja \$a értékét, majd növeli \$a-t eggyel
<code>--\$a</code>	előcsökkentő	Csökkenti \$a-t eggyel, majd visszaadja \$a értékét
<code>\$a--</code>	utócsökkentő	Visszaadja \$a értékét, majd csökkenti \$a-t eggyel

Itt egy egyszerű példaprogram:

```
<?php
echo "<h3>Utónövekményes</h3>";
$a = 5;
echo "5-nek kell lennie: " . $a++ . "<br>\n";
echo "6-nak kell lennie: " . $a . "<br>\n";
echo "<h3>Előnövekményes</h3>";
$a = 5;
echo "6-nak kell lennie: " . ++$a . "<br>\n";
echo "6-nak kell lennie: " . $a . "<br>\n";
echo "<h3>Előcsökkentő</h3>";
$a = 5;
echo "5-nek kell lennie: " . $a-- . "<br>\n";
echo "4-nek kell lennie: " . $a . "<br>\n";
echo "<h3>Utócsökkentő</h3>";
$a = 5;
echo "4-nek kell lennie: " . --$a . "<br>\n";
echo "4-nek kell lennie: " . $a . "<br>\n";
?>
```

Logikai operátorok

Táblázat 10-6. Logikai operátorok

Példa	Név	Eredmény
<code>\$a and \$b</code>	És	Pontosan akkor igaz (TRUE), ha mind \$a mind \$b igazak (TRUE).
<code>\$a or \$b</code>	Vagy	Pontosan akkor igaz (TRUE), ha \$a és \$b között van igaz (TRUE).
<code>\$a xor \$b</code>	Kizáró vagy	Pontosan akkor igaz (TRUE), ha \$a és \$b közül pontosan egy igaz (TRUE).
<code>! \$a</code>	Tagadás	Pontosan akkor igaz (TRUE), ha \$a nem igaz (TRUE).
<code>\$a && \$b</code>	És	Pontosan akkor igaz (TRUE), ha mind \$a mind \$b igaz (TRUE).
<code>\$a \$b</code>	Vagy	Pontosan akkor igaz (TRUE), ha \$a és \$b között van igaz (TRUE).

String operátorok

Két string operátor van. Az egyik az összefűzés operátor ('.'), amely bal és jobb oldali operandusának összefűztjével tér vissza. A második az összefűző-hozzárendelő operátor ('.='), amely hozzáfűzi a jobb oldalon szereplő szöveges értéket a bal oldali operandus végéhez.

```
$a = "Para ";
$b = $a . "Zita"; // most $b értéke "Para Zita"
```

```
$a = "Dárdarázó ";
$a .= "Vilmos"; // most $a értéke "Dárdarázó Vilmos"
```

Vezérlési szerkezetek

Az összes PHP szkript utasítások sorozatából áll. Az utasítás lehet hozzárendelő utasítás, függvényhívás, ciklus, feltételes utasítás, vagy üres utasítás. Az utasítások általában pontosvesszővel végződnek. Ezenkívül az utasításokat csoportosítani lehet; utasításblokkba foglalhatók kapcsos zárójelek segítségével. Az utasításblokkok maguk is utasítások. A különféle utasítástípusokat ebben a fejezetben tárgyaljuk.

if

Az if szerkezet az egyik legfontosabb szerkezete a legtöbb nyelvnek - így a PHP-nek is. A PHP a C-ben megismerthez hasonló if szerkezettel bír:

if (kifejezés)

utasítás

Amint a kifejezésekről szóló fejezetben szerepel, a kifejezés logikai értéke értékelődik ki. Ha kifejezés TRUE, akkor a PHP végrehajtja az utasítást; ha FALSE, akkor figyelmen kívül hagyja. Arról, hogy mely értékek tekinthetők FALSE-nak, a Logikai értékke alakítás c. fejezetben olvashatsz.

Az alábbi példa kiírja, hogy a nagyobb, mint b, ha \$a nagyobb, mint \$b:

```
if ($a > $b)
    print "a nagyobb, mint b";
```

Gyakran sok utasítást kell feltételhez kötve végrehajtani. Természetesen nem kell minden utasításhoz külön if-et írni. Az utasításokat utasításblokkba lehet összefogni. Az alábbi kód például kiírja, hogy a nagyobb, mint b ha \$a nagyobb, mint \$b, és utána hozzárendeli \$a értékét \$b-hez:

```
if ($a > $b) {
    print "a nagyobb, mint b";
    $b = $a;
}
```

A feltételes utasítások vég nélkül további if utasításokba ágyazhatók, amely a program különböző részeinek feltételes végrehajtását igen hatékonyá teszi.

else

Gyakori, hogy egy bizonyos feltétel teljesülése esetén valamilyen utasítást kell végrehajtani, és valamilyen másik utasítást, ha nem teljesül a feltétel. Erre való az else. Az else kibővíti az if utasítást, hogy akkor hajtson végre utasítást, amikor az if kifejezés FALSE-ként értékelődik ki. Az alábbi kód például kiírja, hogy a nagyobb, mint b ha \$a \$b-nél nagyobb, egyébként az a NEM nagyobb, mint b üzenetet írja ki:

```
if ($a > $b) {
    print "a nagyobb, mint b";
} else {
    print "a NEM nagyobb, mint b";
}
```

Az else utasítás csak akkor hajtodik végre, ha az if kifejezés és az összes elseif kifejezés is FALSE értékű. Az elseif -ről most olvashatsz.

elseif

Az elseif, amint azt a neve is sugallja, az if és az else kombinációja. Az else-hez hasonlóan az if utasítást terjeszti ki, hogy különböző utasításokat hajtson végre abban az esetben, ha az eredeti if

kifejezés értéke FALSE lenne. Azonban az else-sel ellentétben csak akkor hajtra végre az alternatív kódrészt, ha az elseif kifejezés TRUE. Az alábbi kód például - \$a értékétől függően - üdvözlí Menő Manót, és Víz Eleket, vagy kiírja, hogy ismeretlen:

```
if ($a == "Menő Manó") {
    print "Szervusz Menő Manó! Rég láttalak!";
} elseif ($a == 'Víz Elek') { #szimpla idézőjel is használható
    print "Üdv Víz Elek!";
} else {
    print "Szervusz, idegen. Hát téged mi szél hozott ide?";
}
```

Egy if kifejezést több elseif követhet. Az első olyan elseif kifejezés hajtódik végre (ha van), amely értéke TRUE. A PHP-ban az 'else if' is (különírva) használható és ugyanúgy fog viselkedni, mint az 'elseif' (egybeírva). A szintaktikai jelentés 'kicsit' eltérő (ha ismered a C-t, nos ez pont úgy működik) de végülis ugyanaz lesz a végeredmény.

Az elseif ág csak akkor hajtódik végre, ha az öt megelőző if kifejezés, és az összes köztes elseif kifejezések FALSE értékűek, de az adott elseif kifejezése TRUE.

Vezérlési szerkezetek alternatív szintaxisa

A PHP bizonyos vezérlési szerkezeteihez egy alternatív szintaxist is nyújt; név szerint: az if, while, for, foreach, és switch számára. Minden esetben az alternatív szintaxisnál a nyitó kapcsos zárójel helyett kettőspontot (:) kell írni, a záró zárójel helyett pedig a vezérlési szerkezetnek megfelelő endif;, endwhile;, endfor;, endforeach;, vagy endswitch; utasításokat értelemszerűen.

```
<?php if ($a == 5): ?>
```

A most éppen 5.

```
<?php endif; ?>
```

A fenti példában az "A most éppen 5." egy alternatív szintaxisú if kifejezésbe van ágyazva. A HTML rész csak akkor íródik ki, ha \$a egyenlő 5-tel.

Az alternatív szintaxis az else-re és az elseif-re is alkalmazható. Az alábbi példa egy if szerkezet, amiben van elseif és else is alternatív formában:

```
if ($a == 0.5):
    print "a most fél.";
    print "De vajon kitől?";
elseif ($a == 8):
    print "Nekem nyolc, hogy mennyi az a.";
    print "Úgyis megváltoztatom az értékét.";
    $a++;
else:
    print "Ez így nem vicces, hogy a se nem fél, se nem nyolc";
endif;
```

while

A while ciklusok a PHP legegyszerűbb ciklusai. Éppen úgy viselkednek, mint a C nyelvbeli megfelelőik. A while általános szintaxisa:

```
while (kifejezés) utasítás
```

A while utasítás jelentése egyszerű. Azt mondja a PHP-nek, hogy mindaddig ismétlje az utasítás(ok) végrehajtását, amíg a while kifejezés TRUE. Iterációnak nevezzük azt, amikor a PHP egyszer végrehajtja az utasítást/utasításblokkot egy ciklus részeként. A kifejezés értéke a ciklus kezdetekor értékelődik ki, tehát még ha az utasításblokk belsejében hamissá is válik a feltétel, a blokk végrehajtása akkor sem áll meg, csak az iteráció végén [feltéve ha közben megint meg nem

változik a feltétel]. Amikor a while kifejezés értéke már az első vizsgálatkor FALSE, akkor az utasítás(blokk) egyszer sem kerül végrehajtásra.

Az if szerkezethez hasonlóan több utasítást csoportosítani lehet a while ciklusban kapcsos zárójelekkel, vagy az alternatív szintaxis használatával:

```
while (kifejezés): utasítás ... endwhile;
```

Az alábbi példák ugyanazt csinálják - 1-től 10-ig kiírják a számokat:

```
/* 1. variáció */
$i = 1;
while ($i <= 10) {
    print $i++; /* a kiírt érték $i, csak
                utána növelünk
                (post-inkrementáció) */
}
/* 2. variáció */
```

```
$i = 1;
while ($i <= 10):
    print $i;
    $i++;
endwhile;
```

do..while

A do..while ciklusok nagyon hasonlóak a while ciklusokhoz, a különbség mindössze annyi, hogy a kifejezés igaz volta itt az iteráció végén értékelődik ki, és nem az elején. A fő különbség a hagyományos while ciklushoz képest, hogy a do..while ciklus első iterációja garantáltan lefut (a kifejezés igazságértékét csak az iteráció végén ellenőrzi), amely nem garantált a hagyományos while ciklusnál (itt a kifejezés igazságértéke az iteráció kezdetén kerül kiértékelésre, ha értéke kezdetben FALSE, akkor a ciklus végrehajtása azonnal befejeződik).

Csak egy szintaxisa van a do..while ciklusnak:

```
$i = 0;
do {
    print $i;
} while ($i>0);
```

A fenti ciklus pontosan egyszer fut le, mert az első iteráció után, amikor a kifejezés igazságértéke vizsgálatra kerül, kiderül, hogy FALSE (\$i nem nagyobb, mint 0) és a ciklus végrehajtása befejeződik.

Haladó C programozók már bizonyára jártasak a do..while ciklus másfajta használatában. Például utasításblokk közepén ki lehet lépni a blokkból, ha az utasításblokkot do..while(0), közé tesszük, és break utasítást használunk. A következő kódrészlet ezt szemlélteti:

```
do {
    if ($i < 5) {
        print "i nem elég nagy";
        break;
    }
    $i *= $factor;
    if ($i < $minimum_limit) {
        break;
    }
    print " i most jó";
    ...i feldolgozása...
} while(0);
```

for

A for ciklus a legbonyolultabb ciklus a PHP-ben. Éppen úgy viselkedik, mint a C nyelvbeli párja.

A for ciklus szintaxisa:

```
for (kif1; kif2; kif3) utasítás
```

A fenti for szerkezettel megegyező az alábbi, remélhetőleg már ismerős kifejezés:

```
kif1;
while (kif2) {
    utasítás;
    kif3;
}
```

Az első kifejezés (kif1) a ciklus kezdetén egyszer kerül végrehajtásra. Minden iteráció elején kif2 kiértékelődik. Ha értéke TRUE, akkor a ciklus folytatódik, és az utasításra kerül a vezérlés. Ha értéke FALSE, akkor a ciklus véget ér. Minden iteráció végén kif3 is végrehajtásra kerül. Bármelyik kifejezést el lehet hagyni. Ha kif2 üres, az azt jelenti, hogy a ciklus a végtelenségig fut [hacsak nem jön a jó tündér break utasítás képében...] (A PHP implicit TRUE-nak feltételezi az üres kif2-t, mint a C.) Ez nem annyira haszontalan, mint elsőre amennyire elsőnek tűnik, hiszen gyakran fejezheted be a ciklust egy feltételes kifejezésbe ágyazott break kifejezéssel a for feltétel kifejezésének kiértékelése helyett. Nézd az alábbi példákat, mindegyikük kiírja a számokat 1-től 10-ig:

```
/* téma */
for ($i = 1; $i <= 10; $i++) {
    print $i;
}
/* 1. variáció */
for ($i = 1;; $i++) {
    if ($i > 10) {
        break;
    }
    print $i;
}
/* 2. variáció */
$i = 1;
for (;;) {
    if ($i > 10) {
        break;
    }
    print $i;
    $i++;
}
/* 3. variáció */
for ($i = 1; $i <= 10; print $i, $i++);
```

Természetesen "a téma" a legbarátságosabb (vagy esetleg a 3. variáció). Sok helyen hasznos azonban, hogy üres kifejezés is írható for ciklusba...

A PHP a for ciklus esetén is megengedi az alternatív szintaxishasználatát:

```
for (kif1; kif2; kif3): utasítás; ...; endfor;
```

Más nyelvekben létezik az ún. foreach szerkezet tömbök vagy hash-ek bejárására. A PHP 3-ban nincs ilyen, de a PHP 4-ben implementálták (lásd: foreach). PHP 3-ban a while, a list() és az each() szerkezeteket használhatod erre a célra.

foreach

A PHP 4-ben (nem a PHP 3-ban!) a Perlhez és más nyelvekhez hasonlóan létezik az ún. foreach szerkezet is. Ez jól használható eszközt ad a tömbökön végzett iterációkhoz. Két szintaxisa létezik, a második egy apró, de hasznos kiegészítéssel nyújt többet az elsőhöz képest.

```
foreach(tömb_kifejezés as $ertek) utasítás  
foreach(tömb_kifejezés as $kulcs => $ertek) utasítás
```

Az első forma végigmegy a tömb_kifejezés szolgáltatotta tömbön. Minden alkalommal az aktuális elem értéke a \$ertek változóba kerül, és a belső tömb mutató növelésre kerül. (A következő alkalommal tehát a soron következő elemet fogja venni). A második forma ugyanezt végzi el, de az aktuális elem kulcsa a \$kulcs változóba kerül. Megjegyzés: Amikor a foreach indul, a belső tömb mutató az első elemre áll. Ez azt jelenti, hogy nem kell meghívni a reset() függvényt egy foreach ciklus előtt.

```
reset ($tomb);  
while (list(, $ertek) = each ($tomb)) {  
    echo "Érték: $ertek<br>\n";  
}  
foreach ($tomb as $ertek) {  
    echo "Érték: $ertek<br>\n";  
}
```

Az alábbiak is azonos eredményt szolgáltatnak:

```
reset ($tomb);  
while (list($kulcs, $ertek) = each ($tomb)) {  
    echo "Kulcs: $kulcs, Érték: $ertek<br>\n";  
}  
foreach ($tomb as $kulcs => $ertek) {  
    echo "Kulcs: $kulcs, Érték: $ertek<br>\n";  
}
```

break

A break azonnal kilép az aktuális for, foreach, while, do..while ciklusból vagy switch szerkezetből. A break elfogad egy elhagyható szám paramétert, amely megadja, hogy hány egymásba ágyazott struktúrából kell egyszerre 'kiugrani'.

```
$tomb = array ('egy', 'kettő', 'három', 'négy', 'stop', 'öt');  
while (list(, $ertek) = each ($tomb)) {  
    if ($ertek == 'stop') {  
        break; /* írhattál volna ide 'break 1;'-et is */  
    }  
    echo "$ertek<br>\n";  
}  
/* Az elhagyható paraméter használata */  
$i = 0;  
while (++$i) {  
    switch ($i) {  
        case 5:  
            echo "5 esetén<br>\n";  
            break 1; /* csak a switch-ből lép ki */  
        case 10:  
            echo "10 esetén kilépés<br>\n";  
            break 2; /* a switch és a while befejezése */  
        default:  
            break;  
    }  
}
```

A continue ciklusok belsejében használható arra, hogy átugorjuk az aktuális iteráció hátralévő részét, és a végrehajtást a következő iterációval folytassuk. A continue elfogad egy elhagyható szám paramétert, amely megadja, hogy hány egymásba ágyazott struktúrának a hátralévő részét kell átugrani.

switch

```
if ($i == 0) {
    print "i most 0";
}
if ($i == 1) {
    print "i most 1";
}
if ($i == 2) {
    print "i most 2";
}
switch ($i) {
    case 0:
        print "i most 0";
        break;
    case 1:
        print "i most 1";
        break;
    case 2:
        print "i most 2";
        break;
}
```

A hibák elkerülése végett fontos megérteni, hogy hogyan kerül végrehajtásra a switch szerkezet. A switch vagyis utasításról utasításra hajtódik végre. Nem hajtódik végre semmilyen utasítás,

csak akkor, ha egy olyan case kifejezést talál a PHP, amely egyezik a switch kifejezés értékével. Ezután a PHP addig folytatja az utasítások végrehajtását, amíg el nem éri a switch blokk végét, vagy nem találkozik egy break utasítással. FONTOS! Ha nem nincs break egy case-hez tartozó utasítás(sorozat) végén, akkor a PHP végrehajtja a soron következő case-hez tartozó utasításokat is! Például:

```
switch ($i) {
    case 0:
        print "i most 0";
    case 1:
        print "i most 1";
    case 2:
        print "i most 2";
}
```

Itt, ha \$i értéke 0, akkor a PHP az összes kiíró utasítást végrehajtja! Ha \$i értéke 1, akkor a PHP az utolsó két print-et hajtja végre, és csak ha \$i értéke 2, akkor kapod a 'kívánt' eredményt (csak az 'i most 2' íródik ki). Tehát nagyon fontos nem elfelejteni a break utasítást (bár bizonyos körülmények között lehet, hogy pont ennek elhagyása a szándékos). A switch kifejezésben a feltétel csak egyszer értékelődik ki és a kapott eredmény lesz összehasonlítva a case kifejezések mindegyikével. Ha elseif kifejezéseket használsz, a kifejezések újra és újra kiértékelődnek. [és újra és újra be kell gépelni. Ez nem csak fárasztó, de hiba forrása is lehet.] Ha a kifejezés bonyolult, vagy egy ciklus belsejében van, a switch a gyorsabb. Egy eset (case) utasításlistája üres is lehet, így a vezérlés a következő case-címkére adódik.

```
switch ($i) {
    case 0:
    case 1:
    case 2:
        print "i 3-nál kisebb, de nem negatív";
        break;
    case 3:
        print "i pont 3";
}
```

Egy különleges eset a default [alapértelmezett] címke. Ez a címke bármivel egyezik, amivel a korábbi case elemek nem egyeztek. Ennek kell az utolsó elemnek lennie. Például:

```
switch ($i) {
    case 0:
        print "i most 0";
        break;
    case 1:
        print "i most 1";
        break;
    case 2:
        print "i most 2";
        break;
    default:
        print "i se nem 0, se nem 1, se nem 2";
}
```

A case kifejezés tetszőleges kifejezés, aminek egyszerű a típusa, vagyis egész vagy lebegőpontos szám, vagy string. Tömbök és objektumok itt nem használhatók, csakis egy-egy elemük ill. változójuk egyszerű típusként.

Az alternatív szintaxis működik a switch-ekkel is. Bővebb információért lásd: Vezérlési szerkezetek alternatív szintaxisa.

```
switch ($i):
    case 0:
        print "i most 0";
        break;
    case 1:
        print "i most 1";
        break;
    case 2:
        print "i most 2";
        break;
    default:
        print "i se nem 0, se nem 1, se nem 2";
endswitch;
```

declare

A declare egy kódblokk számára adott futtatási direktívák beállítását teszi lehetővé. A declare szintaxisa hasonló a vezérlési szerkezetekéhez:

declare (direktíva) utasítás

A direktíva rész a declare blokk működését szabályozza. Jelenleg csak egy direktíva használható, a ticks. (Lásd lejjebb a ticks részleteit) A declare blokk utasítás része mindig egyszer fut le. Az, hogy miképp, és milyen mellékhatásokkal, a direktíva részben megadottaktól függ.

Tick-ek

A tick egy olyan esemény, amely minden N db alacsony szintű utasítás végrehajtásakor bekövetkezik a declare blokkban. Az N értéket a ticks=N szintaxissal kell megadni a declare blokk direktíva részében. Az egyes tick-ekre bekövetkező esemény(ek) a register_tick_function() függvényen állítható(ak) be. Lásd az alábbi példát. Akár több esemény is bekövetkezik egy tick-re. Példa A PHP kód egy részének időmérése

```
<pre>
<?php
// Ez a függvény megjegyzi a hívása időpontjait
function idopontok ($visszaadni = FALSE)
{
    static $idopontok;
    // Visszaadja a $profile tartalmát, és törli
    if ($visszaadni) {
        $idok = $idopontok;
        unset ($idopontok);
        return ($idok);
    }
    $idopontok[] = microtime();
}
// A tick kezelő beállítása
register_tick_function("idopontok");
// Beállítjuk az első időpontot a declare előtt
idopontok();
// A kódblokk futtatása, minden második utasítás egy tick
declare (ticks = 2) {
    for ($x = 1; $x < 50; ++$x) {
        echo similar_text (md5($x), md5($x*$x)), "<br>";
    }
}
```

```

}
// Az időmérő függvény adatainak kiírása
print_r (idopontok(TRUE));
?>
</pre>

```

A fenti példa a declare blokkban lévő PHP kód sebességét méri, rögzítve minden második alacsonyszintű utasítás végrehajtásának időpontját. Ez az információ alkalmas lehet arra, hogy megtaláld a lassan futó részeket a kódodban. Ezt a hatást másképp is el lehet érni, de tick-eket használva sokkal kényelmesebb és könnyebben megvalósítható megoldást kapsz. A tick-ek kiválóan alkalmasak hibakeresésre, egyszerű multitasking megvalósítására, háttérben futattott I/O-ra, és sok más feladatra.

return

A return() utasítás függvényen belül használva azonnal befejezi a folyó függvény futását, és a paramétereként megadott érték szolgáltatja a függvény visszatérési értékét. A return() az eval() függvénnyel futatott kód vagy a szkript futását is leállítja. A globális érvényességi körben használva a folyó szkript futását szakítja meg. Ha ez a szkript az include() vagy a require() hatására lett futtatva, akkor a vezérlés visszaadódik arra a fájlra, ahol ezek az utasítások szerepelnek, valamint include() esetén a return() paramétere lesz az include() utasítás visszatérési értéke. Ha a return() a fő szkriptben lett kiadva, akkor befejeződik a szkript futása. Ha ez a auto_prepend_file vagy auto_append_file konfigurációs beállításban szereplő fájlok valamelyikében történik (lásd: konfigurációs fájl) akkor, (csak) ezeknek a futása fejeződik be.

require()

A require() beilleszti és feldolgozza a megadott fájlt. Ennek részletes mikéntjéről, lásd include()! A require() és az include() megegyezik egymással a hibakezelését leszámítva. Az include() nem fatális hibát, figyelmeztetést generál, a require() viszont fatális hibát jelez. Másszóval, ahol az igényelt fájl nemlétekor a futást meg kell szakítani, ajánlott a require(). Az include() nem így viselkedik, a hibától függetlenül a szkript futtatása folytatódik. Bizonyosodj meg, hogy a include_path helyesen van beállítva! Egyszerű require() példák

```

<?php
require 'prepend.php';
require $valamifajl;
require ('valamifajl.txt');
?>

```

Megjegyzés: PHP 4.0.2 előtt, a következők szerint működött. A require() mindig beolvasta a kívánt fájlt, még ha az a require()-t tartalmazó sorra soha nem is került vezérlés. A feltételes szerkezetek nem befolyásolták a működését. Mégis, ha a require()-t tartalmazó sorra nem került vezérlés a megadott fájlban lévő kód nem futott le. Ehhez hasonlóan, a ciklusok sem befolyásolták a működését. Habár a fájlban szereplő kód függött az azt körülölelő ciklustól, a require() maga csak egyszer történt meg.

include()

Az include() beilleszti és feldolgozza a megadott fájlt. Az alábbiak igazak a require()-ra is. A require() és az include() megegyezik egymással a hibakezelését leszámítva. Az include() nem fatális hibát, figyelmeztetést generál, a require() viszont fatális hibát jelez. Magyarán, ahol az igényelt fájl nemlétekor a futást meg kell szakítani, ajánlott a require(). Az include() nem így viselkedik, a hibától függetlenül a szkript futtatása folytatódik. Bizonyosodj meg, hogy a

include_path helyesen van beállítva! A fájl beillesztése során a megadott fájl öröklí az include() helyén érvényes változó hatáskört. Bármely változó, amely azon a ponton elérhető, elérhető a beillesztett fájlban is. Egyszerű include() példa

```
valtozok.php
<?php
$szin      = 'zöld';
$gyumolcs = 'alma';
?>

teszt.php
<?php
echo "Egy $szin $gyumolcs"; // Egy
include 'valtozok.php';
echo "egy $szin $gyumolcs"; // Egy zöld alma
?>
```

Függvény belsejében a megadott fájlban szereplő kód úgy fog viselkedni, mintha az magában a függvényben szerepelt volna. Ez azt jelenti, hogy a fájl öröklí a változók érvényességi körét. Függvényen belüli beillesztés:

```
<?php
function ize()
{
    global $szin;
    include 'valtozok.php';
    echo "Egy $szin $gyumolcs";
}
/* valtozok.php az ize() függvény hatóköréébe esik, *
 * így a $gyumolcs nem elérhető a függvényen kívül. *
 * A $szin igen, mivel globálisnak lett deklarálva. */
ize(); // Egy zöld alma
echo "Egy $szin $gyumolcs"; // Egy zöld
?>
```

Ha egy fájlt beillesztünk az include()-dal vagy require()-ral, akkor a cél fájl elején az elemző kilép a PHP módból HTML módba, majd visszaáll PHP módba a fájl végén. Ennek okán bármely beillesztendő fájlban levő PHP kódot közre kell fogni egy érvényes PHP kezdő- és zárójelöléssel. Ha az include()-dal hívott fájl HTTP-n keresztül érkezik az "fopen wrapper"-ek használatával, és a célszerver PHP kódként feldolgozza a fájlt, akkor átadhatsz változókat a hívott fájlban HTTP GET lekérési formában. Ez nem teljesen ugyanaz, mintha a include()-dal hívott fájl örökölné a helyi változókat, mivel a szkript valójában a távoli szerveren fut le, és a futási eredmény kerül beépítésre a helyi szkriptbe. Include() HTTP-n keresztül

```
/* Ezek a példák feltételezik, hogy a szerver be van állítva a .php *
 * fájlok feldolgozására és nincs beállítva a .txt fájlok feldolgozására *
 * A 'működik' azt jelenti, hogy az $ize és $bigyo változók elérhetőek *
 * a hívott fájlban. */
// Nem működik: a file.txt nem kerül feldolgozásra
include ("http://szerver/file.txt?ize=1&bigyo=2");
// Nem működik: egy 'file.php?ize=1&bigyo=2' nevű fájlt keres a helyi gépen
include ("file.php?ize=1&bigyo=2");
// Működik
include ("http://szerver/file.php?ize=1&bigyo=2");
$ize = 1;
$bigyo = 2;
include ("file.txt"); /* Működik */
include ("file.php"); /* Működik */
```

Mivel az `include()` és a `require()` különleges nyelvi elem, kapcsos zárójelekkel kell közrefogni, ha egy feltételes utasításon belül szerepel. `Include()` feltételes blokkon belül

```
/*Ez NEM JÓ, és nem a várt eredményt adja */
```

```
if ($feltetel)
    include($file);
else
    include($other);
/* Ez a HELYES */
if ($feltetel) {
    include($file);
} else {
    include($other);
}
```

`return` utasítást lehet elhelyezni egy `include()`-olt fájlban annak érdekében, hogy a kiértékelés ott befejeződjön, és visszaadjon egy értéket a hívó szkriptnek. A visszatérési értéket ugyanúgy használhatod, mint egy közönséges függvénynél. Megjegyzés: PHP 3, a `return` nem jelenhet meg függvény blokkon kívül máshol, amely esetben a függvényből történő visszatérést jelöli. Az `include()` és a `return()` utasítás

```
return.php
```

```
<?php
$var = 'PHP';
return $var;
?>
```

```
noreturn.php
```

```
<?php
$var = 'PHP';
?>
```

```
testreturns.php
```

```
<?php
$size = include 'return.php';
echo $size; // kiírja: 'PHP'
$bigyo = include 'noreturn.php';
echo $bigyo; // kiírja: 1
?>
```

`$bigyo` értéke 1, mert a beillesztés sikeres volt. Figyeld meg a különbséget a két fenti példa között. Az első a `return()` segítségével visszaadott egy értéket, a második nem. Létezik még néhány egyéb módja is változók beemelésének a `fopen()`, `file()` segítségével, vagy `include()` és `Kimenet` szabályzó függvények együttes használatával.

`require_once()`

Az `require_once()` beilleszt és feldolgoz fájlokat a program futása közben. Ez hasonló az `require()` működéséhez, azzal a fontos különbséggel, hogy ha a már egyszer beillesztésre került kódot a PHP nem próbálja meg ismét betölteni. A `require_once()` használatos azokban az esetekben, amikor ugyanaz a fájl esetleg többször kerülhet beillesztésre a szkript futása során, de biztosítani kell, hogy ez ténylegesen csak egyszer történjen meg, így megelőzve a függvények újradefiniálását, változók értékének átállítását, stb.

`include_once()`

Az `include_once()` beilleszt és feldolgoz fájlokat a program futása közben. Ez hasonló az `include()` működéséhez, azzal a fontos különbséggel, hogy ha a már egyszer beillesztésre került kódot a PHP nem próbálja meg ismét betölteni. Az `include_once()` használatos azokban az

esetekben, amikor ugyanaz a fájl esetleg többször kerülhet beillesztésre a szkript futása során, de biztosítani kell, hogy ez ténylegesen csak egyszer történjen meg, így megelőzve a függvények újradefiniálását, változók értékének átállítását, stb.

Függvények

Felhasználó által definiált függvények

Függvényeket a következő szintaxis szerint definiálhatod:

```
function foo ($arg_1, $arg_2, ..., $arg_n)
{
    echo "Példa függvény.\n";
    return $retval;
}
```

Bármely érvényes PHP kód megjelenhet egy függvényen belül, akár még más függvény vagy osztály definíciók is. PHP 3-ban a függvényeket definiálni kell, mielőtt hivatkozás történik rájuk (függvényhívás előtt). PHP 4-ben nincs ez a megkötés. A PHP nem támogatja a függvények polimorfizmusát (többalakúságát), a függvénydefiníciókat nem lehet megszüntetni vagy újradefiniálni egy már definiált függvényeket. A PHP 3 nem támogatja a változó számú függvényargumentumokat, bár az argumentumok kezdőértéke támogatott. Lásd az Argumentumok kezdőértéke című részt bővebb információért. A PHP 4 mindkettő lehetőséget támogatja. Lásd a Változó számú függvényargumentumok című részt és a func_num_args(), func_get_arg() és a func_get_args() függvényeket részletesebb leírásért.

Függvényargumentumok

Az információ a függvényekhez az argumentumlistán keresztül jut el, ami egy vesszővel határolt változó és/vagy konstanslista. A PHP támogatja az érték szerinti (ez az alapértelmezett) referenciakénti paraméterátadást is, és az argumentumok kezdőértékét. A változó hosszúságú argumentumlisták csak a PHP 4-ben jelentek meg. Lásd a változó hosszúságú argumentumlistákat és a func_num_args(), func_get_arg() és a func_get_args() függvényeket részletesebb leírásért. PHP 3-ban hasonló hatás érhető el a függvénynek tömb típusú változó paraméterként történő átadásával:

```
function tombot_kezel($input)
{
    echo "$input[0] + $input[1] = ", $input[0]+$input[1];
}
```

Referencia szerinti argumentumfeltöltés

Alapértelmezésben a függvény paraméterei érték szerint adódnak át (vagyis ha megváltoztatod a változót a függvényen belül, annak a függvényen kívülre nincs hatása). Ha szeretnéd megengedni, hogy a függvény módosítsa az átadott paramétereket, referencia szerint kell átadni azokat. Ha egy függvényargumentum mindig referencia szerint kell átadni, akkor a függvénydefinícióban az argumentum neve elé egy & jelet kell írni.

```
function fgv_extrakkal(&$string)
{
    $string .= 'és a szükséges plusssz.';
}
$string = 'Ez egy karakterfüzér, ';
fgv_extrakkal($string);
echo $string;    // kiírja, hogy 'Ez egy karakterfüzér, és a szükséges plusssz.'
```

Argumentumok kezdőértékei

Bármely függvény skalár-argumentumainak megadhatasz kezdőértéket a C++ szintaxisnak megfelelően:

```
function kavet_csinal ($tipus = "cappuccino")
{
    return "Csinálok egy pohár " . $tipus . "t.\n";
}
echo kavet_csinal ();
echo kavet_csinal ("espresso");
```

A fenti kód kimenete: Csinálok egy pohár cappuccinot.

Csinálok egy pohár espressot.

A kezdőértéknek konstans kifejezésnek kell lennie, nem lehet pl. változó vagy objektum. Figyelj arra, hogy a kezdőértékkel rendelkező argumentumok más argumentumoktól jobbra helyezkedjenek el; különben a dolgok nem úgy mennek majd, ahogy azt várnád. Lásd a következő kódot:

```
function joghurtot_keszit ($type = "acidophilus", $flavour)
{
    return "Készítek egy köcsög $flavour ízű $type-t.\n";
}
echo joghurtot_keszit ("eper"); // nem úgy működik, mint szeretnéd !?!
```

A fenti példa kimenete: Warning: Missing argument 2 in call to joghurtot_keszit() in

/usr/local/etc/httpd/htdocs/phptest/func_test.php on line 41

Készítek egy köcsög ízű eper-t.

Most hasonlítsd össze az alábbival:

```
function joghurtot_keszit ($flavour, $type = "acidophilus")
{
    return "Készítek egy köcsög $flavour ízű $type-ot.\n";
}
echo joghurtot_keszit ("eper"); // ez már jó
```

A fenti példa kimenete: Készítek egy eper ízű acidophilus-t.

Változó hosszúságú argumentumlista

A PHP 4 támogatja a változó hosszúságú argumentumlistát a felhasználók által definiált függvényekben. Valóban nagyon egyszerű kezelni ezt a func_num_args(), func_get_arg() és a func_get_args() függvényekkel. Semmilyen különleges szintakszist nem igényel és az argumentumlista lehet explicit módon adott és viselkedhet úgy is, mint egy normál függvény.

Visszatérési értékek

Az elhagyható return állítást használva adhatnak vissza értéket a függvények. Bármely típus visszaadható, beleértve a listákat és az objektumokat is. A függvény végrehajtása azonnal befejeződik, és a vezérlés visszakerül a függvényhívás utáni pozícióba. További részletes információkért lásd: return()!

```
function negyzete ($num)
{
    return $num * $num;
}
echo negyzete (4); // kiírja '16'.
```

Több értéket nem tud visszaadni a függvény, de hasonló hatás érhető el ezen többszörös értékek listába szervezésével.

```
function kis_szamok()
```

```

{
    return array (0, 1, 2);
}
list ($nulla, $egy, $ketto) = kis_szamok();

```

Ha a függvénynek referenciával kell visszatérnie, akkor az & referencia operátort kell alkalmaznod a függvény deklarációjánál és a visszatérési érték megadásakor is.

```

function &referenciat_ad_vissza()
{
    return &$valtozo;
}
$shivatkozas = &referenciat_ad_vissza();

```

Függvényváltozók

A PHP lehetővé teszi a függvényváltozók használatát. Ha egy változónevet kerek zárójelek követnek, akkor a PHP megkeresi a változó értékével azonos nevű függvényt, és megpróbálja azt végrehajtani. Ezt többek között visszahívandó (callback) függvények vagy függvénytáblák implementálására használható.

A függvényváltozók nem fognak működni az olyan nyelvi elemekkel, mint például az echo(), unset(), isset(), empty() vagy include(). Habár a print() nyelvi elem kivétel. Ez az egyik legjelentősebb különbség a PHP függvények és nyelvi elemek között. Függvényváltozó példa

```

<?php
function ize()
{
    echo "Az ize()-ben<br>\n";
}
function bigyo($param = '')
{
    echo "A bigyo()-ban; az argumentum:'$param'.<br>\n";
}
$func = 'ize';
$func();
$func = 'bigyo';
$func('Stex van Boeven');
?>

```

Fejezet. Osztályok, objektumok

class

Az osztály (objektumtípus) változók és rajtuk műveletet végző függvények [metódusok] együttese. Osztályt az alábbi szintakszis szerint lehet definiálni:

```

<?php
class Kosar
{
    var $dolgok;    // A kosárban levő dolgok

    // berak a kosárba $db darabot az $sorsz indexű dologból
    function berak ($sorsz, $db)
    {
        $this->dolgok[$sorsz] += $db;
    }
    // kivesz a kosárból $db darabot az $sorsz indexű dologból
    function kivesz ($sorsz, $db)
    {

```

```

        if ($this->items[$sorsz] > $db) {
            $this->items[$sorsz] -= $db;
            return true;
        } else {
            return false;
        }
    }
}
?>

```

Ez definiál egy Kosar nevű osztályt, ami a kosárban levő áruk asszociatív tömbjéből áll, és definiál 2 funkciót hozzá, hogy bele lehessen rakni, illetve kivenni a kosárból.

/* Egyik alábbi értékadás sem működik PHP 4-ben */

```

class Kosar
{
    var $mai_datum = date("Y. m. d.");
    var $nev = $csaladi_nev;
    var $tulajdonos = 'Ferenc ' . 'János';
    var $termek = array("Videó", "TV");
}
/* Így kell a fenti beállításokat elérni */
class Kosar
{
    var $mai_datum;
    var $nev;
    var $tulajdonos;
    var $termek;
    function Kosar()
    {
        $this->mai_datum = date("Y. m. d.");
        $this->nev = $GLOBALS['csaladi_nev'];
        /* stb. . . */
    }
}

```

Az osztályok típusok, vagyis az aktuális változók tervrajzai. A kívánt típusú változót a new operátorral hozhatod létre.

```

$kosar = new Kosar;
$kosar->berak("10", 1);
$masik_kosar = new Kosar;
$masik_kosar->berak("0815", 3);

```

Ez létrehozza a Kosar objektumosztály \$kosar és \$masik_kosar nevű objektumpéldányait. A \$kosar objektum berak() függvényét meghívtuk, hogy a 10-es számú árucikkből rakjon 1 darabot a kosárba. Három darab 0815 számú termék került a \$masik_kosar nevű kosárba. Mind a \$kosar, mind a \$masik_kosar objektumoknak megvannak a berak() és kivesz() metódusai, és tulajdonságai. Ezek azonban egymástól független metódusok és tulajdonságok. Az objektumokról hasonlóan gondolkozhat, mint a könyvtárakról az állományrendszerben. Lehetséges, hogy két különböző OLVASSEL.TXT állományod van, ha ezek két különböző könyvtárban vannak. Úgy mint a könyvtáraknál, ahol meg kell adnod a teljes elérési utat, hogy egy állományra szeretnél hivatkozni a gyökérkönyvtárban, a teljes metódusnevet meg kell adnod, hogy meg tudd azt hívni. A PHP nyelvben a gyökérkönyvtár analógiája a globális környezet, és az elérési út elválasztója a ->. Ezért a \$kosar->dolgok név és a \$masik_kosar->dolgok név két különböző változót ad meg. Figyeld meg, hogy a változót \$kosar->dolgok néven kell elérni, és nem \$kosar->\$dolgok néven, azaz a PHP változók neveiben csak egy dollárjelet kell tenned.

```

// helyes, egy dollárjel

```

```
$kosar->dolgok = array("10" => 1);
// helytelen, mivel a $kosar->$dolgok értelme $kosar->""
$kosar->$dolgok = array("10" => 1);
// helyes, de lehetséges, hogy nem a megcélzott eredmény
// $kosar->$valtozo értelme $kosar->dolgok
$valtozo = 'dolgok;
$kosar->$valtozo = array("10" => 1);
```

Egy osztály definiálásakor nem tudhatod, milyen néven lesz majd elérhető az objektumod a PHP programban: a Kosar osztály készítése idején nem volt ismert, hogy később \$kosar vagy \$masik_kosar néven nevezzük-e majd az objektumpéldányt. Ezért nem írhatod a Kosar osztályban, hogy \$kosar->dolgok. De hogy el tudjad érni az osztály saját metódusait és tulajdonságait az objektumpéldány(ok) nevétől függetlenül, használhatod a \$this kvázi-változót, amit 'a sajátom' vagy 'az aktuális objektumpéldány' értelemben alkalmazhatsz. Ezért a '\$this->dolgok[\$sorsz] += \$db' úgy olvasható, hogy 'adj \$db darab \$sorsz sorszámú terméket a saját dolgok tömbömhöz', vagy 'adj \$db darab \$sorsz sorszámú terméket az aktuális objektumpéldány dolgok tömbjéhez'.

extends

Gyakori, hogy szeretnél olyan osztályokat kialakítani, amelyek egy már meglévő osztályhoz hasonló tulajdonságokkal és metódusokkal rendelkeznek. Tulajdonképpen jó gyakorlat egy általános osztályt definiálni, amit minden projektedben használhatsz, és ezt az osztályt alakítani az egyes projektek igényeinek megfelelően. Ennek a megvalósítása érdekében az osztályok lehetnek más osztályok kiterjesztései. A kiterjesztett, vagy származtatott osztály minden tulajdonsággal és metódussal rendelkezik, ami a kiindulási osztályban megvolt (ezt nevezzük öröklésnek, bár senki sem hal meg a folyamat során). Amit hozzáadsz a kiindulási osztályhoz, azt nevezzük kiterjesztésnek. Nem lehetséges megcsonkítani egy osztályt, azaz megszüntetni egy metódust, vagy tulajdonságot. Egy leszármazott osztály mindig pontosan egy alaposztálytól függ, azaz egyidejűleg többszörös leszármaztatás nem támogatott. A kiterjesztés kulcsszava az 'extends'.

```
class Gazdas_Kosar extends Kosar
{
    var $tulaj;

    function tulajdonosa ($nev)
    {
        $this->tulaj = $nev;
    }
}
```

Ez definiál egy Gazdas_Kosar nevű osztályt, ami a Kosar összes változójával és metódusával rendelkezik, és van egy saját változója, a \$tulaj, no meg egy saját metódusa, a tulajdonosa(). A gazdas kosarat a hagyományos módon hozhatod létre, és a kosár tulajdonosát is be tudod állítani, le tudod kérdezni [ebben az esetben favágó módszerrel]. A gazdas kosarakon továbbra is lehet használni a Kosar függvényeit:

```
$gkosar = new Gazdas_Kosar; // Gazdas kosár létrehozása
$gkosar->tulajdonosa("Namilesz Teosztasz"); // a tulaj beállítása
print $gkosar->tulaj; // a tulajdonos neve
$gkosar->break("10", 1); // (Kosar-ból öröklött funkcionalitás)
```

Konstruktor

Figyelem A PHP 3 és PHP 4 konstruktorok különbözőképpen működnek. A PHP 4 megvalósítása erősen javasolt. A konstruktorok az osztályok olyan metódusai, amelyek automatikusan meghívásra kerülnek egy új objektumpéldány new kulcsszóval történő létrehozása során. A PHP 3-ban egy metódus akkor tekinthető konstruktornak, ha a neve megegyezik az osztály nevével. A PHP 4-ben egy metódus akkor lesz konstruktorrá, hogy a neve megegyezik annak az osztálynak a nevével, ahol definiálták. A különbség hajszálnyi, de kritikus (lásd lentebb).

// A PHP 3 és PHP 4 verziókban is működik

```
class Auto_Kosar extends Kosar
{
    function Auto_Kosar ()
    {
        $this->berak ("10", 1);
    }
}
```

Ez egy olyan Auto_Kosar nevű osztályt [objektumtípust] hoz létre, mint a Kosar, csak rendelkezik egy konstruktorral, amely inicializálja a kosarat 1 darab "10"-es áruval, valahányszor a new operátorral hozzuk létre az objektumot. [de csak akkor!!!] A konstruktoroknak is lehet átadni paramétereket, és ezek lehetnek elhagyhatók is, amely még hasznosabbá teszi őket. Ha paraméterek nélkül is használható osztályt szeretnél, állíts be minden paraméternek alapértéket.

// A PHP 3 és PHP 4 verziókban is működik

```
class Konstruktoros_Kosar extends Kosar
{
    function Konstruktoros_Kosar ($sorsz = "10", $db = 1)
    {
        $this->berak ($sorsz, $db);
    }
}
// Mindig ugyanazt az uncsi dolgot veszi...
$kiindulo_kosar = new Konstruktoros_Kosar;
// Igazi vásárlás
$masik_kosar = new Konstruktoros_kosar ("20", 17);
```

A PHP 3-ban a leszármazott osztályokra és konstruktorokra számos korlátozás van. Az alábbi példákat érdemes alaposan áttekinteni, hogy megértsd ezeket a korlátozásokat.

```
class A
{
    function A()
    {
        echo "Én vagyok az A osztály konstruktora.<br>\n";
    }
}
class B extends A
{
    function C()
    {
        echo "Én egy metódus vagyok.<br>\n";
    }
}
// PHP 3-ban semmilyen konstruktor sem kerül meghívásra
$b = new B;
```

PHP 3-ban semmilyen konstruktor sem kerül meghívásra a fenti példában. A PHP 3 szabálya a következő: 'A konstruktor egy metódus, aminek ugyanaz a neve, mint az osztálynak'. Az osztály neve B, és nincs B() nevű metódus a B osztályban. Semmi sem történik. Ez a PHP 4-ben ki van

javítva egy másik szabály bevezetésével: Ha az osztályban nincs konstruktor, a szülő osztály konstruktora hívódik meg, ha létezik. A fenti példa kimenete 'Én vagyok az A osztály konstruktora.'
' lett volna PHP 4-ben.

```
class A
{
    function A()
    {
        echo "Én vagyok az A osztály konstruktora.<br>\n";
    }
    function B()
    {
        echo "Én egy B nevű metódus vagyok az A osztályban.<br>\n";
        echo "Nem vagyok A konstruktora.<br>\n";
    }
}
class B extends A
{
    function C()
    {
        echo "Én egy metódus vagyok.<br>\n";
    }
}
// Ez meghívja B()-t, mint konstruktort
$b = new B;
```

A PHP 3-ban az A osztály B() metódusa hirtelen konstruktorrá válik a B osztályban, habár ez soha sem volt cél. A PHP 3 szabálya: 'A konstruktor egy metódus, aminek ugyanaz a neve, mint az osztálynak'. A PHP 3 nem foglalkozik azzal, hogy a metódus a B osztályban van-e definiálva, vagy öröklés útján áll rendelkezésre. Ez a PHP 4-ben ki van javítva egy másik szabály bevezetésével: 'A konstruktor egy metódus, aminek ugyanaz a neve, mint az osztálynak, ahol definiálták'. Ezért a PHP 4-ben a B osztálynak nincs saját konstruktora, így a szülő osztály konstruktora hívódik meg, kiírva, hogy 'Én vagyok az A osztály konstruktora.'
' . Sem a PHP 3, sem a PHP 4 nem hívja meg a szülő osztály konstruktorát automatikusan egy leszármazott osztály definiált konstruktorából. A te feladatod, hogy meghívod a szülő konstruktorát, ha szükséges. Nem léteznek destruktorkok sem a PHP 3 sem a PHP 4 verzióiban. Bár használhatod a register_shutdown_function() függvényt a destruktorkok legtöbb viselkedésének eléréséhez.

::

Az alábbiak csak PHP 4-ben érvényesek. Időnként hasznos az ősoosztályok metódusaira vagy tulajdonságaira hivatkozni, vagy olyan osztálymetódusokat meghívni, amelyek nem példányosított objektumokhoz tartoznak. A :: operátor erre használható.

```
class A
{
    function pelda()
    {
        echo "Én az eredeti A::pelda() metódus vagyok.<br>\n";
    }
}
class B extends A
{
    function pelda()
    {
        echo "Én a felüldefiniáló B::pelda() metódus vagyok.<br>\n";
        A::example();
    }
}
```

```

    }
}
// nincs semmilyen objektum az A osztályból
// ez azonban ki fogja írni:
//   Én az eredeti A::pelda() metódus vagyok.<br>
A::pelda();
// B egy objektuát hozzuk létre
$b = new B;
// ez ki fogja írni:
//   Én a felüldefiniáló B::pelda() metódus vagyok.<br>
//   Én az eredeti A::pelda() metódus vagyok.<br>
$b->pelda();

```

A fenti példa meghívja az A osztály pelda() metódusát, habár nincs konkrét példányunk az A osztályból, tehát ezt nem írhatnánk le az \$a->pelda()-hoz hasonlóan. Ehelyett a pelda() egy 'osztálymetódusként' viselkedik, azaz az osztály egy függvényeként, és nem egy példány metódusaként. Osztálymetódusok léteznek, de osztálytulajdonságok (változók) nem. Mivel a hívás pillanatában semmilyen objektum nem létezik, egy osztálymetódus nem használhat objektum változókat, és egyáltalán nem használhatja a \$this speciális referenciát. Egy objektummetódus azonban természetesen dolgozhat globális változókkal és lokális változókkal is. A fenti példa a B osztályban felüldefiniálja a pelda() metódust. Az A osztálytól örökölt eredeti definíció eltűnik, és többé nem érhető el, hacsak nem az A osztályban megvalósított pelda() függvényre hivatkozol közvetlenül, a :: operátor segítségével. Ennek eléréséhez A::pelda()-t kell használni (ebben az esetben írhatnál parent::pelda()-t is, ahogy az a következő szakaszban olvasható). Ebben a környezetben van aktuálisan használt objektum, és ennek lehetnek objektum változói (tulajdonságai). Ekképpen ha egy objektummetóduson belül használsz ezt az operátort, akkor alkalmazhatod a \$this-t, és felhasználhatod az objektum tulajdonságait.

parent

Gyakran van szükség arra, hogy a szülő tulajdonságaira vagy metódusaira hivatkozzunk leszármazott osztályokban. Ez különösen igaz, ha a leszármazott osztály egy finomítása, vagy specializálása az alaposztálynak. Ahelyett, hogy a szülő osztály nevét megadd minden ilyen meghíváskor (mint a hogy a :: operátor példája mutatta), használhatod a parent speciális nevet, ami tulajdonképpen a szülő osztály nevét jelenti, amit az extends kulcsszónál megadtál. Ennek a speciális névnek a használatával elkerülöd a szülő osztály nevének ismétlődését. Ha a megvalósítás során a leszármazási fát meg kell változtatni, csak egy helyen, az extends kulcsszónál kell átírnod a nevet.

```

class A
{
    function pelda()
    {
        echo "Én A::pelda() vagyok egyszerű funkcióval.<br>\n";
    }
}

class B extends A
{
    function pelda()
    {
        echo "Én B::pelda() vagyok több funkcióval.<br>\n";
        parent::pelda();
    }
}

```

```
$b = new B;
```

```
// Ez a B::pelda() metódust hívja, ami az A::pelda()-t hívja  
$b->pelda();
```

Objektumok szerializációja, objektumok session-ökben

Megjegyzés: A PHP 3-ban az objektumok elveszítik az osztály-hozzárendelésüket a szerializációs, és deszerializációs folyamat során. Az eredmény objektum típusú, de nem tartozik semelyik osztályhoz, és nincs egy metódusa sem, tehát eléggé használhatatlan (csupán egy tömb, furcsa szintakszissal). A következő információk csak a PHP 4-es változatra érvényesek. A `serialize()` egy karaktersorozatot ad vissza, ami az átadott érték byte-sorozatban megadott megfelelője. Az `unserialize()` visszaalakít egy ilyen karaktersorozatot az eredeti értéké. A szerializációs folyamat során egy objektum átadásával elmenthetjük az objektum minden tulajdonságát (változóját). A függvények nem kerülnek elmentésre, csak az osztály neve. Ahhoz, hogy az `unserialize()` segítségével vissza lehessen állítani egy objektumot, az objektum osztályának (típusának) már definiálva kell lennie. Ez a következőket jelenti egy példán keresztül megvilágítva. Ha az `elso.php` oldalon az `A` osztályú `$a` objektumot szerializálsz, akkor kapsz egy olyan karaktersorozatot, amely az `A` osztályra hivatkozik, és tartalmazza az összes `$a`-ban lévő változó (tulajdonság) értékét. Ha ezt a karaktersorozatot a `masodik.php` oldalon objektummá szeretnéd alakítani, újra létrehozva az `A` osztályú `$a` nevű objektumot, akkor az `A` osztály definíciójának rendelkezésre kell állnia a `masodik.php` oldalon is. Ez úgy érhető el, hogy az `A` osztály definícióját egy külső állományban tárolod, és ezt alkalmazod mind az `elso.php`, mind a `masodik.php` oldalon.

`aosztaly.inc:`

```
class A  
{  
    var $egy = 1;  
  
    function egyet_mutat()  
    {  
        echo $this->egy;  
    }  
}
```

`elso.php:`

```
include("aosztaly.inc");  
$a = new A;  
$s = serialize($a);  
// tároljuk az $s-t valahol, ahol masodik.php megtalálja  
$fp = fopen("tarolas", "w");  
fputs($fp, $s);  
fclose($fp);
```

`masodik.php:`

```
// ez szükséges, hogy a deszerializáció rendben menjen  
include("aosztaly.inc");  
$s = implode("", @file("tarolas"));  
$a = unserialize($s);  
// most már használható az egyet_mutat() metódus  
$a->egyet_mutat();
```

Ha session-öket alkalmazol, és a `session_register()` függvénnyel regisztrálsz objektumokat, ezek az objektumok automatikusan szerializálódnak minden PHP program futása után, és deszerializálódnak minden további programban. Ez egyszerűen azt jelenti, hogy ezek az

objektumok akármelyik oldalon feltűnhetnek, miután a session részévé váltak. Erősen javasolt, hogy minden regisztrált objektum osztály definícióját betöltsd minden oldalon, még akkor is, ha éppen nem használod azokat. Ha ezt nem teszed meg, és egy objektum úgy deszerializálódik, hogy nem áll rendelkezésre az osztály definíciója, el fogja veszteni az osztály hozzárendelését, és az stdClass osztály egy példánya lesz, metódusok nélkül, így használhatatlanná válik. Ezért ha a fenti példában az \$a a session részévé válik a session_register("a") meghívásával, akkor be kell töltened az aosztaly.inc külső állományt minden oldalon, nem csak az első.php és második.php programokban.

A speciális __sleep és __wakeup metódusok

A serialize() ellenőrzi, hogy van-e az osztályodnak __sleep nevű metódusa. Ha van, ez lefut a szerIALIZÁCIÓ előtt. Ez megtisztíthatja az objektumot, és végül egy tömbbel tér vissza, amely tartalmazza az adott objektum ténylegesen szerIALIZÁLANDÓ tulajdonságainak neveit.

A __sleep célja, hogy bezárjon minden adatbázis kapcsolatot, a várakozó adatokat lementse, és hasonló 'tisztító' jellegű tevékenységeket végezzen. Hasznos lehet akkor is, ha nagyon nagy objektumaid vannak, amelyeket külön szeretnél lementeni.

Ezzel szemben az unserialize() a speciális __wakeup nevű függvényt használja. Ha ez létezik, ez a függvény alkalmazható arra, hogy visszaállítsa az objektum erőforrásait.

A __wakeup célja lehet például, hogy visszaállítson egy adatbázis kapcsolatot, ami a szerIALIZÁCIÓ során elveszett, és hasonló beállítási feladatokat végezzen.

Referenciák a konstruktorban

Referenciák képzése konstruktorokban problémás helyzetekhez vezethet. Ez a leírás segít a bajok elkerülésében.

```
class Ize
{
    function Ize($nev)
    {
        // egy referencia létrehozása a globális $globalref változóban
        global $globalref;
        $globalref[] = &$this;
        // a név beállítása a kapott értékre
        $this->nevBeallitas($nev);
        // és kiírás
        $this->nevKiiras();
    }
    function nevKiiras()
    {
        echo "<br>", $this->nev;
    }
    function nevBeallitas($nev)
    {
        $this->nev = $nev;
    }
}
```

Nézzük, hogy van-e különbség az \$obj1 és az \$obj2 objektum között. Az előbbi a = másoló operátorral készült, az utóbbi a =& referencia operátorral készült.

```
$obj1 = new Ize('konstruktorban beállított');
$obj1->nevKiiras();
$globalref[0]->nevKiiras();
/* kimenete:
```

```

konstruktorban beállított
konstruktorban beállított
konstruktorban beállított */
$obj2 =& new Ize('konstruktorban beállított');
$obj2->nevKiiras();
$globalref[1]->nevKiiras();
/* kimenete:
konstruktorban beállított
konstruktorban beállított
konstruktorban beállított */

```

Szemmel láthatóan nincs semmi különbség, de valójában egy nagyon fontos különbség van a két forma között: az \$obj1 és \$globalref[0] `_NEM_` referenciák, NEM ugyanaz a két változó. Ez azért történhet így, mert a "new" alapvetően nem referenciával tér vissza, hanem egy másolatot ad. Nincsenek teljesítménybeli problémák a másolatok visszaadásakor, mivel a PHP 4 és újabb verziók referencia számlálást alkalmaznak. Legtöbbször ellenben jobb másolatokkal dolgozni referenciák helyett, mivel a referenciák képzése eltart egy kis ideig, de a másolatok képzése gyakorlatilag nem igényel időt. Ha egyik sem egy nagy tömb, vagy objektum, és a változásokat nem szeretnéd mindegyik példányban egyszerre látni, akkor másolatok használatával jobban jársz.

ZEND2

Referenciák

Mik a referenciák

A referenciák lehetőséget adnak PHP-ben azonos változó tartalom elérésére különböző nevek alatt. Ezek szimbólumtábla bejegyzések, nem olyanok, mint a C nyelv mutatói. PHP-ben a változók neve és tartalma két különböző dolog, tehát ugyanaz a tartalom kaphat különböző neveket. A legjobb hasonlat talán a UNIX állománynevek és állományok rendszere. A változóneveket könyvtár bejegyzéseként foghatod fel, a változók tartalmát állományokként. A referenciák olyanok, mint UNIXban a hardlinkek.

Mit lehet referenciákkal tenni

A PHP referenciák lehetőséget adnak arra, hogy egy értékhez két nevet lehessen rendelni. Ez azt jelenti, hogy a következő programban:

```
$a =& $b;
```

az \$a és \$b nevek ugyanarra az értékre hivatkoznak. Az \$a és a \$b nevek teljesen egyenrangúak. Nem arról van szó, hogy az \$a a \$b-re mutat, vagy fordítva, hanem arról, hogy az \$a és a \$b név ugyanannak az értéknek két elnevezése. Ugyanez a forma használható az olyan függvényeknél, amelyek referenciát adnak vissza, vagy a new operátor használatakor (a PHP 4.0.4 és későbbi verziókban):

```

$obj = & new valamilyen_osztaly();
$size =& valtozo_kereses ($valami);

```

Ha nem használod az & -t, akkor az osztálypéldány másolata adódik át. A \$this objektumon belüli használatával ugyanazon az objektumpéldányon dolgozol. Ha az értékadás során az & -t elhagyod, akkor az objektumról másolat készül és a \$this már ezen a másolaton fog dolgozni. Van, amikor ez nem kívánatos, mivel általában egy példányon szeretnénk dolgozni a jobb memóriahasználat és teljesítmény érdekében. A referenciákat paraméterátadáskor is

lehet használni. Ebben az esetben a meghívott függvény egy lokális változója és a hívó környezet egy változója ugyanazt az értéket fogja képviselni. Például:

```
function ize (&$valtozo)
{
    $valtozo++;
}
$a = 5;
ize ($a);
```

Ez a kód az \$a változó értékét 6-ra állítja. Ez azért történik meg, mivel az ize függvényben a \$valtozo egy referencia a \$a változó értékére.

Mit nem lehet referenciákkal tenni

Mint korábban írtuk, a referenciák nem mutatók. A következő konstrukció ezért nem a vártan megfelelően viselkedik:

```
function ize (&$valtozo)
{
    $valtozo =& $GLOBALS["valami"];
}
ize($valami);
```

A foo függvényben a \$valtozo változó a \$valami értékéhez lesz kötve, de utána ezt megváltoztatjuk a \$GLOBALS["valami"] értékére. Nincs lehetőség a referenciák segítségével a \$valami más értékhez kötésére a hívó környezetben, mivel a \$valami nem áll rendelkezésre az ize függvényben. Ott a \$valtozo reprezentálja az értéket, amely csak változó tartalommal bír és nem név-érték kötéssel a hívó szimbólumtáblájában.

Referenciakénti paraméterátadás

A függvényeknek változókat referenciaként is át lehet adni, így a függvény tudja módosítani a hívó környezetben definiált értéket. Ez a következőképpen oldható meg:

```
function ize (&$valtozo)
{
    $valtozo++;
}
```

```
$a = 5;
ize ($a);
// $a itt 6
```

Figyeld meg, hogy nincs referencia jelzés a függvényhíváskor, csak a függvény definíciójában. Ez önmagában elég a megfelelő működéshez. A következők szerepelhetnek referenciakénti paraméterátadásban:

- Változó, például ize(\$a)
- New utasítás, például ize(new osztaly())
- Egy függvény által visszaadott referencia, például:

```
function &valami()
{
    $a = 5;
    return $a;
}
ize(valami());
```

Minden más kifejezést kerülni kell referencia szerinti paraméterátadáskor, mivel az eredmény határozatlan lesz. A következő példákban a referencia szerinti paraméterátadás hibának minősül:

```
function valami() // Figyeld meg, nincs & jel!
```

```

{
    $a = 5;
    return $a;
}
ize(valami());

ize($a = 5) // Kifejezés, nem változó
ize(5) // Konstans, nem változó

```

Ezek a meghatározások a PHP 4.0.4 és későbbi verzióira érvényesek.

Referencia visszatérési-érték

A referencia visszatérési-érték pl. olyan változók megtalálásakor lehet hasznos, amelyekről referenciát kell készíteni. Ha referenciát kell visszaadni visszatérési értéként, akkor használd az alábbi formát:

```

function &valtozo_kereses ($param)
{
    ...kód...
    return $megtalalt_valtozo;
}

```

```

$size =& valtozo_kereses ($valami);
$size->x = 2;

```

Ebben a példában a `valtozo_kereses` egy objektumot keres meg, és a megtalált objektum egy tulajdonságát állítjuk át - helyesen. A referenciák használata nélkül a másolatának egy tulajdonságán tettük volna mindezt - hibásan. A paraméter átadással ellentétben, itt a `&` jelet mindkét helyen meg kell adnod a referenciavisszaadás jelöléséhez. Így nem egy másolatot kapsz, és az `$size` változóra nézve referencia hozzárendelés történik, nem pedig érték hozzárendelés (értékmásolás).

Referenciák megszüntetése

Amikor megszüntetsz egy referenciát, csak megszakítod a változónév és az érték közötti kapcsolatot. Ez nem azt jelenti, hogy a változó értékét töröld. Például:

```

$a = 1;
$b =& $a;
unset ($a);

```

nem fogja megszüntetni a `$b` nevet, csak az `$a` nevet, így az érték a `$b` néven továbbra is elérhető. Ismét érdemes a Unix `unlink` parancsával és az állományrendszerrel való hasonlatosságra gondolni.

A PHP által használt referenciák

Sok konstrukció a PHP-ben referenciák segítségével valósul meg, azért minden fentebb tárgyalt kérdés ezekre az elemekre is igaz. Néhány olyan konstrukciót, mint a referencia átadást vagy visszatérést már kifejtettünk, más referenciákat használó konstrukciók:

global referenciák

Amikor egy változót a global `$valtozo` formával globálisként használsz, tulajdonképpen egy referenciát képzelsz a megfelelő globális változóra, azaz a következő kódnak megfelelő történik:

```

$valtozo =& $GLOBALS["valtozo"];

```

Ez például azt is jelenti, hogy a `$valtozo` törlése nem fogja törölni a globális változót.

\$this

Egy objektum metódusban a \$this mindig az aktuális példányra egy referencia.

Hibakezelés

A hibáknak és figyelmeztetéseknek PHP-ben számos típusa van. Ezek:

PHP hiba típusok

Érték	Szimbólum	Leírás	Megjegyzés
1	E_ERROR	fatális futás-idejű hibák	
2	E_WARNING	nem fatális futás-idejű hibák	
4	E_PARSE	fordítás-idejű feldolgozási hibák	
8	E_NOTICE	futás-idejű figyelmeztetések (a notice a warning-nál gyengébb)	
16	E_CORE_ERROR	fatális hibák, amik a PHP elindulásakor lépnek fel	csak a PHP 4-ben
32	E_CORE_WARNING	nem fatális hibák figyelmeztetései (warning), amik a PHP elindulásakor lépnek fel	csak a PHP 4-ben
64	E_COMPILE_ERROR	fatális fordítás-idejű hibák	csak a PHP 4-ben
128	E_COMPILE_WARNING	nem fatális fordítás-idejű figyelmeztetések (warning)	csak a PHP 4-ben
256	E_USER_ERROR	felhasználó által generált hibaüzenetek	csak a PHP 4-ben
512	E_USER_WARNING	felhasználó által generált figyelmeztetések (warning)	csak a PHP 4-ben
1024	E_USER_NOTICE	felhasználó által generált figyelmeztetések (notice)	csak a PHP 4-ben
	E_ALL	az összes fent felsorolt elem	csak a PHP 4-ben

A fenti értékek (akár a numerikusak, akár a szimbolikusak) arra használhatóak, hogy felépíts egy bitmask-et, ami megadja, hogy mely hibákat kell jeleznie a PHP-nek. Használhatsz bitszintű operátorokat, hogy összeállítsd a fenti elemekből a neked megfelelő értéket, vagy letiltasz egyes hibákat. Csak a '|', '~', '!', és '&' operátorok használhatóak php.ini fájlban, és semmilyen operátor sem használható a php3.ini fájlban.

PHP 4-ben az alapbeállítású error_reporting érték E_ALL & ~E_NOTICE, ami azt jelenti, hogy minden hiba és figyelmeztetés megjelenik az E_NOTICE-szint kivételével. PHP 3-ban az alapbeállítás (E_ERROR | E_WARNING | E_PARSE), ugyanezt jelenti. Vedd figyelembe, hogy ezek a konstansok nem támogatottak a PHP 3 php3.ini fájljában, ezért az error_reporting beállítás a numerikus 7 érték.

Ezek a beállítások az ini fájl error_reporting direktívájával változtathatóak meg, vagy az Apache httpd.conf fájlban a php_error_reporting (php3_error_reporting PHP 3 esetén) direktívával vagy végül futásidőben egy szkriptben az error_reporting() függvénnyel. Ha a kódod vagy a szervered frissítéd PHP 3-ról PHP 4-re, jól teszed, ha ellenőrzöd ezeket a beállításokat és az error_reporting() függvényhívásokat, különben akaratlanul kikapcsolod az új hibatípusokat, különösen az E_COMPILE_ERROR-t. Ez üres dokumentumokhoz vezethet, amik nem tartalmaznak semmilyen utalást arra, hogy mi történt, vagy hogy hol kellene keresni a problémát. Minden PHP kifejezés írható a "@" előtaggal, ami kikapcsolja a hibajelentést arra a kifejezésre. Ha hiba lép fel a kifejezés kiértékelésekor, és a track_errors szolgáltatás be van kapcsolva, a hibaüzenet megtalálható a \$php_errormsg globális változóban. A @ hibakezelő operátor nem kapcsolja ki a szkriptek feldolgozása során előforduló hibák (parse error) jelentését.

Jelenleg a "@" hibakezelő operátor kikapcsolja azon kritikus hibák jelentését is, amik megállítják a szkript futását. Más problémák mellett, ha egy függvényből érkező hibaüzenetek elnyelésére használod a "@" jelet, meg fog állni a szkript futása, ha nem létezik a megadott függvény, vagy elírtad a nevét.

Az alábbiakban láthatsz egy példát a PHP hibakezelő képességeire. Definiálunk egy hibakezelő függvényt, ami tárolja a hibákat egy fájlba (XML formátummal) és email-t küld a fejlesztőnek ha a programban kritikus hiba történik. Hibakezelés használata egy szkriptben:

```
<?php
// saját hibakezelést építünk
error_reporting(0);
// felhasználó által definiált hibakezelő függvény
function sajátHibaKezelo ($hibaszam, $hibauzenet, $filenev, $sorszam,
$valtozok) {
    // időbélyeg a hibához
    $ido = date("Y-m-d H:i:s (T)");
    // Asszociatív tömb definiálása a hibaszövegeknek.
    // Valójában csak a 2,8,256,512 és 1024 elemeket
    // vesszük figyelembe
    $hibatipus = array (
        1 => "Error",
        2 => "Warning",
        4 => "Parsing Error",
        8 => "Notice",
        16 => "Core Error",
        32 => "Core Warning",
        64 => "Compile Error",
        128 => "Compile Warning",
        256 => "User Error",
        512 => "User Warning",
        1024=> "User Notice"
    );
    // azok a hibatípusok, amikre a változókat is el kell menteni
    $user_hibak = array(E_USER_ERROR, E_USER_WARNING, E_USER_NOTICE);
    $hiba = "<errorentry>\n";
    $hiba .= "\t<datetime>".$ido."</datetime>\n";
    $hiba .= "\t<errornum>".$hibaszam."</errornum>\n";
    $hiba .= "\t<errortype>".$hibatipus[$hibaszam."</errortype>\n";
    $hiba .= "\t<errormsg>".$hibauzenet."</errormsg>\n";
    $hiba .= "\t<scriptname>".$filenev."</scriptname>\n";
    $hiba .= "\t<scriptlinenum>".$sorszam."</scriptlinenum>\n";
    if (in_array($hibaszam, $user_hibak))
        $hiba .=
"\t<vartrace>".wddx_serialize_value($valtozok,"Variables")."</vartrace>\n";
    $hiba .= "</errorentry>\n\n";
    // teszteléshez
    // echo $hiba;
    // a hibanapló elmentése, email küldés ha kritikus hiba van
    error_log($hiba, 3, "/usr/local/php4/error.log");
    if ($hibaszam == E_USER_ERROR)
        mail("phpdev@example.com","Kritikus programhiba",$hiba);
}

function tavolsag ($vektor1, $vektor2) {
    if (!is_array($vektor1) || !is_array($vektor2)) {
        trigger_error("Helytelen paraméterek, tomboket varok", E_USER_ERROR);
        return NULL;
    }
    if (count($vektor1) != count($vektor2)) {
        trigger_error("A vektorok ugyanolyan dimenziojuak legyenek",
E_USER_ERROR);
        return NULL;
    }
}
```

```

    }
    for ($i=0; $i<count($vektor1); $i++) {
        $c1 = $vektor1[$i]; $c2 = $vektor2[$i];
        $d = 0.0;
        if (!is_numeric($c1)) {
            trigger_error("Az első vektor $i koordinátája nem szám, nullával
szamolok",
                        E_USER_WARNING);
            $c1 = 0.0;
        }
        if (!is_numeric($c2)) {
            trigger_error("A második vektor $i koordinátája nem szám, nullával
szamolok",
                        E_USER_WARNING);
            $c2 = 0.0;
        }
        $d += $c2*$c2 - $c1*$c1;
    }
    return sqrt($d);
}
$regi_hiba_kezelo = set_error_handler("sajatHibaKezelo");
// nem definiált konstans, warning-ot generál
$t = NEM_VAGYOK_DEFINIALVA;
// néhány "vektor" definiálása
$a = array (2,3, "ize");
$b = array (5.5, 4.3, -1.6);
$c = array (1, -3);
// user hiba generálása
$t1 = tavolsag($c, $b)."\n";
// újabb user hiba generálása
$t2 = tavolsag($b, "ez nem tömb")."\n";
// warning generálása
$t3 = tavolsag($a, $b)."\n";
?>

```

Adatbázis kezelés (MySQL függvények)

Az alábbi kis példa bemutatja, hogyan lehet MySQL adatbázisokhoz csatlakozni, kérést végrehajtani, kiírni az eredményt és megszüntetni a kapcsolatot. Példa 1. MySQL modul áttekintő példa

```

<?php
// Csatlakozás, adatbázis kiválasztása
$kapcsolat = mysql_connect("mysql_hoszt", "mysql_azonosito", "mysql_jelszo")
    or die("Nem tudok csatlakozni");
print "A kapcsolódás sikerül";
mysql_select_db("az_en_adatbazisom")
    or die("Nem sikerült kiválasztanom az adatbázist");
// SQL kérés végrehajtása
$keres = "SELECT * FROM az_en_tablam";
$eredmeny = mysql_query($keres) or die("Hiba a kérésben");
// Az eredmény kiírása HTML-ben
print "<table>\n";
while ($line = mysql_fetch_array($eredmeny, MYSQL_ASSOC)) {
    print "\t<tr>\n";
    foreach ($sor as $egy_oszlop) {
        print "\t\t<td>$egy_oszlop</td>\n";
    }
}

```

```
        print "\t</tr>\n";
    }
    print "</table>\n";
    // Kapcsolat lezárása
    mysql_close($kapcsolat);
?>
```