Webprogramozás I.

Szabó Bálint

MÉDIAINFORMATIKAI KIADVÁNYOK

Webprogramozás I.

Szabó Bálint

Eger, 2013

Korszerű információtechnológiai szakok magyarországi adaptációja

TÁMOP-4.1.2-A/1-11/1-2011-0021

Lektorálta:
Nyugat-magyarországi Egyetem Regionális Pedagógiai Szolgáltató és Kutató Központ

Felelős kiadó: dr. Kis-Tóth Lajos
Készült: az Eszterházy Károly Főiskola nyomdájában, Egerben
Vezető: Kérészy László
Műszaki szerkesztő: Nagy Sándorné

Tartalom

1. BEVEZETÉS

1.1 CÉLKITŰZÉSEK, KOMPETENCIÁK A TANTÁRGY TELJESÍTÉSÉNEK FELTÉTELEI

Ma már senki sem kérdőjelezi meg, hogy a world wide web megjelenése a könyvnyomtatáshoz hasonló, ha ugyan nem még azt is messze felülmúló hatással van az emberi kultúra fejlődésére.

Bár első változata csak az informálódás tér- és időkorlátait bontotta meg, napjainkra a média integrálódásának egyik elsődleges platformjává, virtuális világok hordozóközegévé, a globális emberi tudat neuronhálózatává vált.

Mindennapjaink egyre nagyobb hányadát töltjük számítógép közelben, online kapcsolatban, és bár észre sem vesszük, de ilyenkor mind gyakrabban, lassacskán szinte kizárólag webes felületen dolgozunk. Weblapokon olvasunk híreket, weben intézzük bankügyeinket, választjuk ki és vásároljuk meg a termékek egyre nagyobb hányadát, weben tanulunk, közösségi oldalakon tartjuk a kapcsolatot barátainkkal, szeretteinkkel. Semmiképpen sem feledkezhetünk meg arról sem, hogy webes felületen keressük meg a bennünket érdeklő információk jelentős részét. A web lassan-lassan egy a valódival szorosan összefonódó, mégis azzal párhuzamosan létező virtuális világgá alakul. Virtuális világgá, amelyről egyre nehezebb eldöntenünk, hogy részévé válik-e életünknek, vagy mi magunk kezdünk e beolvadni szövedékébe.

Az 1990-ben Tim Berners-Lee, a CERN munkatársa által megalkotott információs rendszer még távolról sem volt alkalmas mindannak a megvalósítására, amit ma a world wide web-fogalomhoz társítunk. A webböngésző, webszerver, a HTTP protokoll, a HTML nyelv, és az URL címzés együttesén alapuló szolgáltatás, alig volt több mint jó ötlet, eszköz, ami a tér és idő korlátokat lebontva biztosította az elektronikus dokumentumokban hordozott információ megszerezését.

Az eredeti tervek nem szóltak többről, mint egy interneten megvalósított hálózati szolgáltatásról, amely szöveges dokumentumok elosztott tárolását, gyors letöltését és megjelenítését, platformoktól függetlenül biztosította. A web szó arra utalt, hogy az egymástól tetszőleges távolságra lévő gépeken tárolt hipertext-dokumentumok, a szövegükbe ágyazott, más oldalakra mutató hivatkozások révén egyfajta gigantikus, ágas-bogas dokumentumszövedékké integrálódtak, amelyben a felhasználók hihetetlen gyorsasággal, szinte észrevétlenül navigálhattak.

A web páratlan karrierjét néhány véletlen és szerencsés epizód mellett a tudatos tervezés és átgondolt fejlesztés egyengette.

A '90-es évek elején a WWW nem az egyetlen hasonló képességekkel rendelkező internetes szolgáltatás volt. Azt, hogy a Gopher akár komoly ellenféllé is fejlődhetett volna, jól mutatja, hogy az első grafikus webböngésző, a Mosaic még web- és Gopher-kliens volt egyaránt. Nem tudjuk, mivé alakulhatott volna a Minnesotai Egyetem fejlesztése, ha maguk a fejlesztők nem ítélik halálra azzal, hogy 1993-tól licencdíjat kértek a Gopher-szerverek referenciakódjának felhasználásért. A Gopher-kiszolgálók és ezzel a szolgáltatás egészének sorsa megpecsételődött.

A CERN ugyanebben az évben jelen tette be, hogy a world wide web használata szabad és ingyenes.

Az első csata megnyerését Tim Berners-Lee tudatos, átgondolt, és messzire tekintő fejlesztőmunkája követte. Tim a World Wide Web Consortium (W3C) 1994-es megalapításával olyan nemzetközi szervezetet indított útjára, amely a mai napig koordinálja a web fejlődését, korrigálja a hibákat, és előre tekintő fejlesztésekkel igyekszik elérni, hogy a WWW továbbra is kiszolgálja a felhasználók egyre gyarapodó elvárásait, és a kor változó kommunikációs követelményeit.

A W3C munkájának volt köszönhető a mai web alapfeltételének megteremtése is. Az 1995-ben megjelenő HTML 2.0 űrlapok kialakítását tette lehetővé, ezzel pedig a kétirányú kommunikáció lehetőségét teremtette meg a world wide weben.

Ezzel a nem túl jelentősnek tűnő mozzanattal indult el az a folyamat, amelynek eredményeképpen a WWW-böngészők vékony klienssé alakultak, és alkalmazás szervereken futó, távoli alkalmazások webes elérését, és webfelületen történő vezérlését tették lehetővé.

A WWW igazi forradalma ezen a ponton kezdődött el.

Az egyszerű, bár szabadon bejárható statikus tartalmak helyét átvették a webfelületen futó a banki rendszerek, a keresőmotorok, a távoktatási portálok, kereskedelmi oldalak, online kezelhető dokumentumtárak, a közösségi portálok és a számtalan egyéb egyszerű feladatokat megoldó, vagy akár egész virtuális világot építő webes alkalmazások.

1.1.1 Célkitűzés

A számítógépet használók között alig van olyan, aki ne dolgozna nap mint nap webes alkalmazásokkal. Mindenki természetesnek veszi ezek ergonomikus

felületét, hibamentes, tökéletes, és gyors működését. Jóval kevesebben tudják azonban, hogy a csillogó, egyre több felhasználói élményt és funkcionális lehetőséget biztosító alkalmazások hátterében milyen komoly fejlesztői munka áll. Az átlagos webhasználónak fogalma sincs azokról a kiterjedt ismeretekről, amelyekre a webfelületen futó távoli alkalmazások készítéséhez szükség van.

Tananyagunkkal ezen ismeretek alapjait kívánjuk bemutatni az olvasónak. Reményeink szerint leckéinket átolvasva Ön megismeri a web kliens-szerver architektúrájának működését, megérti az állapotmentes protokollból adódó előnyöket és fejlesztési nehézségeket, valamint elsajátítja azokat a programozási alapismereteket, amelyek birtokában – tankönyvsorozatunk következő elemének áttanulmányozásával – önálló webes alkalmazások készítésére válik képessé.

1.1.2 Kompetenciák

A tananyag leckéinek elolvasásával Ön

- megismeri a WWW kezdeti, kétrétegű kliens-szerver architektúrájának működését,
- érteni fogja a webes kliens és a szerver közötti HTTP-üzenetváltásokat,
- elsajátítja a kliensoldali JavaScript-programozás alapjait,
- megismeri a szerveroldali programozás alapját biztosító háromrétegű kliens-szerver architektúrát,
- betekintést nyer a PHP nyelv kialakulásába és történetébe,
- megismeri egy Apache, PHP, MySQL programozási környezet telepítését és használatát,
- elsajátítja a PHP nyelvi elemeinek szintaktikáját és használatukat,
- megtanulja, hogy programjának megfelelő strukturálásával hogyan lehet úrrá a webalkalmazás kezdeti bonyolultságán,
- megismeri a PHP objektumorientált programozásának alapjait,
- áttekinti a webkliens és -szerver közötti adatcserét biztosító HTTP protokoll állapotmentességéből adódó problémák kiküszöbölésének lehetőségeit,
- megismeri a PHP hibakeresési és -kezelési lehetőségeit,
- megérti a négyrétegű kliens-szerver architektúra lényegét,
- megtanulja a PHP-MySQL kapcsolat kezelésének, a MySQL adatbázisokat használó webes alkalmazások készítésének alapjait.

1.1.3 A tantárgy teljesítésének feltételei

A webprogramozás talán a legtöbb ismertkört összefogó informatikai feladat. Tananyagunk ismertetése közben nem tudunk kitérni a kapcsolódó ismeretek tárgyalására, ezért nélkülözhetetlen, hogy Ön már rendelkezzen a megfelelő informatikai tudással.

- A tantárgy teljesítéséhez az alapvető informatikai kompetenciákon túl ismernie kell a HTML nyelvet,
- és a weblapok formázására használt CSS formátumleíró nyelv eszközeit.
- Rendelkeznie kell algoritmizálási ismeretekkel és szerencsés valamilyen magas szintű programozás nyelv ismerete is.
- Ismernie kell a relációs adatmodellt,
- a relációs adatbázisok kezelésére használható SQL nyelvet,
- és a MySQL adatbázis-kezelő rendszert!

1.1.4 A tananyag feldolgozásához szüksége eszközök

Leckéink példáinak kipróbáláshoz, és egyben az eredményes tanuláshoz szüksége lesz a **megfelelő fejlesztői környezetre**. Napjaink otthoni számítógépeinek döntő többségén Windows operációs rendszer fut, ugyanakkor a webes alkalmazások futását biztosító kiszolgálógépeken nem ilyen jelentős a Windows előnye. Ehhez hozzá kell tenni, hogy a tananyaguk témájául választott PHP nyelvet elsősorban Linux alatt működő webalkalmazások készítésére fejlesztették, és bár az Apache PHP MySQL környezet Windows alá is telepíthető, nem jellemző, hogy a PHP-ben fejlesztett web alkalmazásokat Windows-környezetben használnák.

Ebből következik, hogy nagy eséllyel Ön is Windows operációs rendszert futtató gépen dolgozik, ugyanakkor az elkészített web alkalmazásokat Linux-alapú számítógépeken fogja elérhetővé tenni leendő felhasználói számára. Informatikai zsargonnal élve, a fejlesztett alkalmazásokat Linuxos gép hostolja majd.

A probléma megoldására több lehetőség is kínálkozik.

1. A fejlesztés és felhasználás az alkalmazást véglegesen hostoló számítógépen történik.
2. A fejlesztés saját a gépen, a kód tárolása, a tesztelés és végleges használat is az alkalmazást hostoló számítógépen folyik.
3. A fejlesztés és tesztelés saját gépen zajlik, a kész alkalmazást a hostoló számítógépre telepítik.

Mindegyik módszernek van előnye és hátránya. Az első esetben például minden, a fejlesztésre használt szoftveres eszközt telepíteni kell a hostoló gépre, amin a fejlesztőnek megfelelő jogosultsággal is kell rendelkeznie. Ugyanakkor a fejlesztés a végleges futási környezetben történik, így telepítésre sem lesz szükség.

A második az első és a harmadik módszer lehetőségeit ötvözi. Bár nem kell a hostoló gépre telepíteni a fejlesztőkörnyezetet, változatlanul kiemelt jogosultságokra van szükségünk a szerveren.

A harmadik megoldás esetében semmilyen fejlesztőeszközt nem kell telepítenünk a hostoló gépre, minden ilyen programot saját gépünkön futtatunk. A fejlesztés közben gépünkön mintegy modellezzük a szerver nyújtotta környezetet. Ez a módszer meglehetősen kényelmes fejlesztést tesz lehetővé. Kisebb meglepetések akkor érhetnek bennünket, amikor a webalkalmazás telepítésekor derül ki, hogy a hostoló gép, és a saját gépünkön modellezett környezet nem teljesen azonos. Ez akár a kész alkalmazás kisebb-nagyobb módosítását is szükségessé teheti.

Ebben a tananyagban nem törekszünk komplex, működő webalkalmazások elkészítésére. Elsődleges célunk az, hogy Ön a web-programozás alapjait sajátíthassa el, így leginkább arra van szükségünk, hogy a kódrészleteket gyorsan megértse és ki is próbálhassa. Éppen ezért a harmadik módszert választjuk, azaz **Windows alatt alakítjuk ki a munkához szükséges fejlesztőkörnyezetet**.

Ezt a döntést az is indokolja, hogy manapság több olyan integrált programcsomag is létezik, ami csekély konfigurációs igény mellett is lehetővé teszi az Apache+PHP+MySQL környezet Windows operációs rendszerre telepítését. A tananyag példáinak kipróbálásra is egy ilyen ingyenes környezet, az **XAMPP** használatát javasoljuk. A szoftver telepítéséről a 3. leckében szólunk majd.

http://www.apachefriends.org/download.php?xampp-win32-1.7.7-VC9-installer.exe

1. link Az XAMPP webhelye

A PHP-kódok begépelése történhet az bármilyen szövegeditor segítségével is, azonban létezik néhány olyan fejlesztőkörnyezet is, amely debug-lehetőségekkel és számos egyéb a hasznos szolgáltatással (beépített webszer-

ver és böngésző, syntax highlight, tesztelés, fájlkezelés, adatbázis-kliens stb…) segíti a programozó munkáját. A professzionális eszközök között ilyen például a **Nusphere PhpEd**, ami ugyan kereskedelmi szoftver, de 14 napig ingyenesen használható.

http://www.nusphere.com/products/phped.htm

2. link A PhpEd honlapja

Tananyagunkban a hasonló képességekkel rendelkező CodeLobster nevű alkalmazást használjuk majd, amelynek van teljesen ingyenesen használható verziója is.

http://www.codelobster.com/download.html

3. link CodeLobster letöltése

1.2 A KURZUS TARTALMA

Tananyagunk összesen 12 leckére tagolódik. A most olvasott, 1. lecke bevezető információkkal, és tanulási tanácsokkal szolgál, az utolsó, 12. lecke pedig összegzi a tananyagban megszerezett ismereteket. A 2–11. leckék tartalmazzák a korábban felsorolt kompetenciák megszerzéséhez szükséges ismeretanyagot.

A 2. leckében a world wide web kialakulásáról, a kettő, három, és több rétegű kliens-szerver architektúrákról, a HTTP protokollról, a HTML nyelvről, az URL-címről, a statikus és dinamikus weboldalakról olvashat.

A „Kliensoldali programozás alapjai” című lecke a böngészőben futó, JavaScriptekkel manipulált dinamikus weboldalak készítésének alapjait ismerteti.

Az ezután következő leckében térünk át a szerveroldali programozásra. A 4. lecke a PHP programozási környezet telepítését, a PHP-program szerkezetét és futtatását mutatja be.

Az 5. lecke a programkóddal közvetlenül feldolgozott adatok tárolására használható nyelvi elemek, a literálok, a konstansok, az egyszerű változók, és a tömbök használatát, valamint kliens- és szerveroldali összetevők közötti adatcsere lehetőségeit taglalja.

A következő leckében ismerkedünk meg a PHP szelekciós és iterációs nyelvi elemeivel, a különböző elágazásokat és ismétléseket implementáló vezérlési szerkezetekkel.

A 8. lecke a PHP objektumorientált programozásra alkalmas nyelvi elemeit ismerteti. A leckében megismerheti az osztály, objektum, adattag, metódus, konstruktor, destruktor fogalmakat.

A 9. lecke a HTTP protokoll állapotmentességéből adódó programozási problémák megoldási lehetőségeiről, a munkamentek és a cookie-k kezeléséről szól.

Az utolsó előtti leckében a programkód hibáinak kereséséről, kiküszöböléséről, és a futási időben keletkező hibák kezeléséről tanulunk.

Tananyagunk programozásról szóló utolsó leckéjében a webalkalmazások adatbázis-kapcsolatának megvalósításáról, a PHP-MySQL kapcsolat fölépítéséről és kezeléséről, az adatbázis adatainak weblapokon történő megjelenítéséről olvashat.

1.3 TANULÁSI TANÁCSOK, TUDNIVALÓK

Tananyagunk 2–11. leckéje felöleli az web programozáshoz kapcsolódó alapismereteket. A leckék azonos szerkezetűek, fölépítésüket úgy igyekeztünk kialakítani, hogy a lehető legjobban segítsék az olvasót a megértésben, és a tananyag eredményes elsajátításában.

Minden lecke a **Célkitűzés, kompetenciák** szakasszal kezdődik. Ebben a bevezetésként is felfogható leckerészben találja meg az anyag áttanulmányozásával megszerezhető kompetenciákat, illetve itt olvashat a kitűzött célokról is. Célok alatt ne egyszerű felsorolást képzeljen el. Általában olyan problémákat, kérdéseket vetünk fel, amelyek az előző fejezetek alapján már Önben is megfogalmazódhattak. A lecke célja, hogy az új ismeretekkel megkeressük, és meg is adjuk a válaszokat a felsorolt problémákra. A bevezető kérdések ennek megfelelően, a lecke logikai gondolatmenetét is meghatározzák. Arra kérjük, gondol-

kodjon együtt a tananyag írójával. A szöveg olvasásakor keresse a válaszokat, és ne lépjen tovább a leckéből, amíg azokat meg nem találta.

A célok után a lecke ismeretanyaga következik. A szövegben eltérő formátummal jeleztük a valamilyen szempontból kiemelkedő bekezdéseket, szövegrészeket. Az alábbi formátumokkal találkozhat:

Alkalmazások menüelemei, menüparancsok

Definíciók, fogalmak

Fájlrendszerben használt elérési utak

Fontos szöveg

Grafikus felületen található vezérlő elemek, objektumok

```
Programkódok,
SQL-mondatok
HTML-források
```

Az adott szövegrész megértéséhez szükséges, a tananyaghoz mellékelt példaprogram neve.

Megjegyzések

Nyomógombok, **forró billentyűk**

Feladatok

☞ Gyakorlatok

Összefoglaló kérdések

- Válaszok

A leckében található **fogalmakat**, **definíciókat** igyekezzen a legpontosabban megtanulni. Természetesen nem a szó szerinti ismétlés, hanem a lényeg szabatos megfogalmazása a fontos.

Fordítson különös figyelmet a **fontos** szövegrészekre!

A **gyakorlatokat**, **feladatokat** minden esetben végezze el. Ezek ugyanis hozzásegítik ahhoz, hogy a szerzett ismereteket a gyakorlatban is képes legyen kamatoztatni.

A **kódokat**, SQL-mondatokat elsősorban a szemléltetés érdekében illesztettük be, azonban úgy igyekeztünk elhelyezni őket a tananyagban, hogy vágólapon keresztül, közvetlenül is bemásolhatók legyenek a felhasználás helyére.

Ha a kódok felhasználásának ezt a módját választja, legyen óvatos!

A vágólapról történő beillesztés során gyakran jelentkeznek kisebb-nagyobb hibák, amelyek a karakterek konverziójából adódnak. Tipikusan ilyen az idézőjelek fölcserélődése, vagy a sorvégjelek okozta hibák. Mielőtt futtatja a vágólappal másolt kódokat, minden esetben végezzen szintaktikai ellenőrzést!

Minden egyes lecke végén megtalálja **Összefoglalás** szakaszt, amely logikusan követhető sorrendbe szedve, tömören összegzi a leckében található ismereteket. Mielőtt elolvasná az összegzést, a lényeg kiemelésével foglalja össze fejben a tanultakat! Ha valami nem jut eszébe, olvasson vissza bátran a tananyagban! Csak az önálló összefoglalás után vesse össze saját gondolatait a lecke összefoglalásával.

Az összegzést a frissen szerzett tudás ellenőrzésére használható **önellenőrző kérdések** követik. Soha ne mulassza el ezek áttekintését! Minden kérdéshez megtalálja a helyes választ is. Ezt lehetőleg ne olvassa el mindaddig, amíg önállóan nem sikerült felelnie a feltett kérdésre. A válaszok csupán arra valók, hogy ellenőrizze saját megoldása helyességét.

1.4 FORRÁSOK

A tananyag elsajátításához leckénként elkülönített mappákban különböző forrásokat, állományokat biztosítunk. Amennyiben a tananyag mellé lemezmellékletet kapott, azon megtalálja ezeket az fájlokat. Ha tananyagot elektronikus környezetben sajátítja el, a letölthető fájlok elérhetők a kurzus felületén.

A szövegben jelezzük, hogy melyik példaprogram tartozik az adott szöveghez.

1. letöltés Források RAR formátumban

2. letöltés Források ZIP formátumban

2. LECKE: WEBALKALMAZÁSOK ELEMEI ÉS MŰKÖDÉSÜK

2.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

1989-ben Timothy Berners-Lee hozzákezdett egy új, számítógép-hálózatokon hozzáférhető dokumentumtípus és annak kezelésére alkalmas információs rendszer megvalósításához.

Az ötlet központi gondolata a **hipertextnek** nevezett, **szöveges** információt hordozó dokumentumtípus létrehozása. A „hiper" jelző arra utalt, hogy az ilyen dokumentumok a szövegbe ágyazott, **más hasonló fájlokra mutató hivatkozásokat**, úgynevezett linkeket is tartalmaznak. Az elképzelés szerint a hipertext-dokumentumok az **internet kiszolgáló gépein elosztottan** helyezkednek el, de a szövegben található hivatkozások révén **egymáshoz kapcsolódnak, és világméretű dokumentumszövedéket** (web), hatalmas összekapcsolt szöveget, **hipertextet alkotnak**.

A **World Wide Web** névre keresztelt rendszer gondolata **négy alappilléren nyugszik**. Az első az SGML-alapokon nyugvó dokumentumjelölő nyelv, a **HTML**. A második a HTML nyelven leírt dokumentumok, azaz weblapok elosztott tárolását, és interneten keresztül történő letöltését, megjelenítését biztosító **kliens-szerver architektúra**. A harmadik fontos pillér a **webszerver és kliens** közötti adatcsere szabványosságáért felelős protokoll, a **HTTP**. Az utolsó fundamentum a weblapok címezhetőségét, ezzel letölthetőségét, és a közöttük lévő kapcsolatok leírását biztosító **URL címzési rendszer**.

A kezdetben csak az egyszerűen formázott szövegek kezelésére kialakított hálózati információs rendszer, később jelentős fejlődésen ment keresztül. A statikus tartalmak letöltését és megjelenítését szolgáltató lehetőségen messze túllépve, távoli gépeken futó alkalmazások kezelőfelületének biztosítására vált alkalmassá.

Az addig statikus szövegekből álló dokumentumszövedékbe dinamikus tartalmak költöztek, amelyek a passzív befogadásra kárhoztatott felhasználót a web aktív résztvevőjévé, egyszersmind alkotójává is tették. Ez a különböző médiumok integrációjával együtt járó folyamat vezetett oda, hogy napjainkra a WWW az emberi kommunikáció alapvető platformjává, tudás globális hordozójává, gazdagításának, továbbfejlesztésének talán leghatékonyabb eszközévé fejlődött.

Tananyagunk első leckéjében az internet legnépszerűbb szolgáltatásnak kialakulását a webprogramozó nézőpontjából igyekszünk megvizsgálni. A lecke

végére ismerni a fogja HTTP protokoll jellemzőit, és statikus és dinamikus weboldalak közötti különbségeket, és megérti, hogyan teszi lehetővé, egyszersmind miért nehezíti meg a HTTP a szerveroldalon dinamikusan előállított tartalmak használatát.

2.2 A WORLD WIDE WEB ALAPJAI

A world wide web alapjait a HTML nyelv és az URL címzési rendszer definiálásával, a webszerver, a -kliens feladatainak meghatározásával, valamit a közöttük lebonyolítható kommunikáció szabályait rögzítő HTTP protokoll megalkotásával rakta le Tim Berners Lee.

2.2.1 A HTML

A HyperText Markup Language nyelvet a felsorolás, és a történeti hűség kedvéért említjük. Jelölőinek bemutatására, a webdokumentumok szerkezetének, a használható formátumoknak leírására nem térünk ki, a területre úgy tekintünk, hogy az ismert az olvasó előtt. A nyelv fontos jellemzőit az alábbiakban összegezhetjük.

A HTML az SGML szintaxisára épülő jelölőnyelv, amely azonban elődjének leíró jelölésével szemben funkcionális, azaz műveleti jelölést biztosít. A kisebb és nagyobb relációs jelek közé zárt (<>) HTML-jelölők nem a szöveg szemantikai értelmét, hanem a megjelenítésekor elvégzendő műveleteket írják le. A weblapok szövegébe ágyazva az egyes szövegrészek megjelenítési módját határozzák meg a böngésző számára.

A WWW kialakulásának érdekessége, hogy a Tim Berners-Lee által definiált HTML 1.0 verziója meglehetősen szegényes volt, mindössze 18 jelölőt tartalmazott. Csak szöveges tartalmak leírását tette lehetővé, és azokban is csak a legalapvetőbb szerkezeti elemek (a bekezdések, hivatkozások, felsorolások…) jelölését engedte meg.

http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html

4. link Az eredeti HTML jelölői

Az áttörést 1995-ben megjelent HTML 2.0 hozta, amely számos szövegformázási lehetőséget, és többek között képek csatolását is biztosította.

A továbbfejlesztett jelölőnyelv igazi jelentősége azonban nem a gazdagabb formátumjelölésben, hanem az űrlapok készítésére alkalmas, <FORM> elem megjelenésében rejlett. A FORM és a hozzá kapcsolódó egyéb elemek (INPUT, SELECT, TEXTAREA) ugyanis lehetővé tették a felhasználói adatbevitelt, ezzel megnyitották az utat a webfelületen bevitt adatok programozott feldolgozása előtt. **A HTML 2.0 biztosította először, hogy a weblapok háttérben futó alkalmazások kezelőfelületeiként működjenek!**

A HTML jelenlegi fejlesztése az 5.0 verziónál tart, amely mediális elemek beillesztést, animációk készítését, kliens oldali adattárolást, illetve a kliens és szerver kapcsolattartást egyszerűsítő elemeket is tartalmaz.

A webprogramozás tekintetében a HTML jelentősége leginkább abban áll, hogy a távoli gépeken futó alkalmazásoknak HTML-formátumú, a böngészőben megjeleníthető kezelőfelületet kell kialakítaniuk.

2.2.2 Webszerver, webkliens

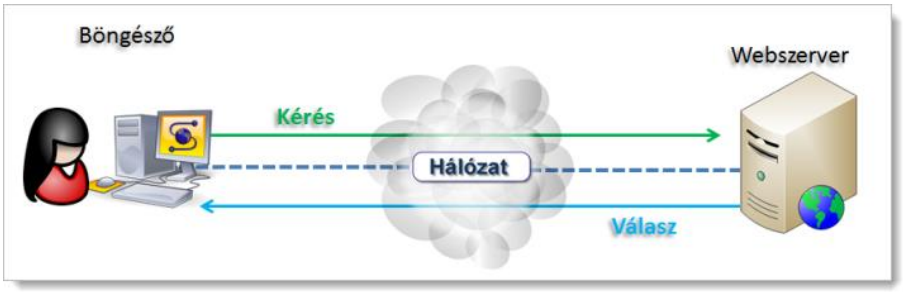
A WWW szolgáltatásnak az internet gépein elosztottan tárolódó, de bárhonnan elérhető, és bármely hoston megjeleníthető szöveges dokumentumok kezelését kellett biztosítania.

Tim Berners Lee kétszintű **kliens szerver architektúra** implementálásával oldotta meg feladatot. A **webszerver** feladata a hipertext oldalak tárolása és a **webkliensek** kiszolgálása. A kiszolgáló folyamatosan „figyeli” a kliensek felől érkező, weblapokra vonatkozó **kéréseket**, amelyeket a kért weblapot, vagy valamilyen egyéb adatot tartalmazó **válasszal szolgál ki**. A szerverek általában folyamatosan működnek, bármikor alkalmasak a bejövő kérések kiszolgálására.

A **webkliensek**, más néven **böngészők** biztosítják a **szolgáltatás és a felhasználók közötti kapcsolatot**. A felhasználó gépén futó böngésző, biztosítja a szolgáltatás kezelőfelületét.

- Amikor a felhasználó valamilyen weblap letöltését kéri, akkor a böngésző felveszi a kapcsolatot a dokumentumot tároló szerverrel, és elküldi a weblapra vonatkozó kérést.
- A kiszolgáló által visszaküldött oldalt, vagy üzenetet a felhasználó számára értelmezhető formában láthatóvá teszi képernyőn.

- A weblap megjelenítésekor természetesen értelmezi a szövegbe ágyazott HTML-jelölőket, és azoknak megfelelő formátumban jeleníti meg a dokumentumot.
- Ha a böngésző számára ismeretlen, vagy hibásan megadott jelölővel találkozik, akkor nem küld hibaüzenetet, hanem egyszerűen figyelmen kívül hagyja azt.



1. ábra Kétrétegű kliens-szerver architektúra

Az első webszervert és a WorldWideWeb névre keresztelt első böngészőt Tim Berners-Lee készítette. Mivel a HTML 1.0-ban még nem volt lehetőség képek, táblázatok űrlapok használatára a WorldWideWeb roppant egyszerű volt, mindössze a HTML 1.0 összesen 18 féle jelölőjével formázott weblapok letöltésére és megjelenítésére volt alkalmas.

2.2.3 URL-címzés

Az URL (Inified Resource Locator) olyan egyedi azonosító rendszer, amelynek segítségével a világ különböző web szerverein tárolt erőforrások – például weblapok – egyedi módon, egyértelműen azonosíthatóak.

Amikor a felhasználó egy weblap tartalmát szeretné látni képernyőjén, akkor annak URL-címét kell begépelnie a böngésző kezelőfelületén. A webkliens a cím alapján veszi fel a kapcsolatot a szerverrel, és küldi el a kívánt erőforrásra vonatkozó kérést.

Az URL címek általános leírása az alábbi:

```
[protokoll://]host[[/útvonal/][fájl][?paraméterek]]
```

Jól látható, hogy cím szinte minden eleme elhagyható. Ez nem azt jelenti, hogy ezekre nincs szükség, hanem azt, hogy minden elhagyható elemnek meg-

van az alapértelmezett értéke, amelyet vagy a böngésző, vagy a szerver helyettesít a megfelelő helyre.

- A *protokoll* jelöli a kliens és böngésző közötti **üzenetváltás szabályrendszerét**. Kezdetben ez csak HTTP lehetett, és a böngészők ma is ezt helyettesítik a cím elejére, ha a felhasználó elmulasztja begépelni a protokollt. Jelentőségéről később beszélünk.
- A *host* a webszerver számítógép IP-címe, vagy domain neve, a dokumentumot tároló **host egyértelmű azonosítója**. Ez a cím egyetlen kötelező része.
- Az útvonal határozza meg, hogy a szerver melyik **mappájában** található a kért állomány. Ha a felhasználó nem adja meg, akkor a **böngésző a főkönyvtár jelét** (/) helyettesíti be.
- *Fájl*: ez az opció jelzi, hogy a cím előző részeivel megjelölt gép és mappa, **melyik fájljára** van szükség. Ha a felhasználó nem gépelte be, akkor a szerver a számára **alapértelmezett fájlnevet** helyettesíti ide. Ez általában az `index.html`, de szerverenként szabályozható név.

Az URL-címek biztosítják a weblapok közötti kapcsolatok, linkek, hivatkozások leírását is. Minden hivatkozás a weblap egy fizikailag látható eleméből, és a hozzá kapcsolt, URL-címből áll.

Amikor a felhasználó egy hivatkozás fizikai részére kattint, a böngésző kiolvassa a dokumentumból a link URL-címét, és úgy viselkedik, mintha a címet a felhasználó gépelte volna be. Fölveszi a kapcsolatot a címben meghatározott szerverrel, és elkéri tőle a dokumentumot.

A hivatkozások URL-címe megadható abszolút és relatív módon is. Az relatív cím – a fent megadott formával szemben – nem tartalmazza a host címét. Mindig arra a szerverre vonatkozik, és abból a mappából indul, ahonnan a hivatkozást tartalmazó weblapot letöltötte a böngésző.

2.2.4 HTTP protokoll

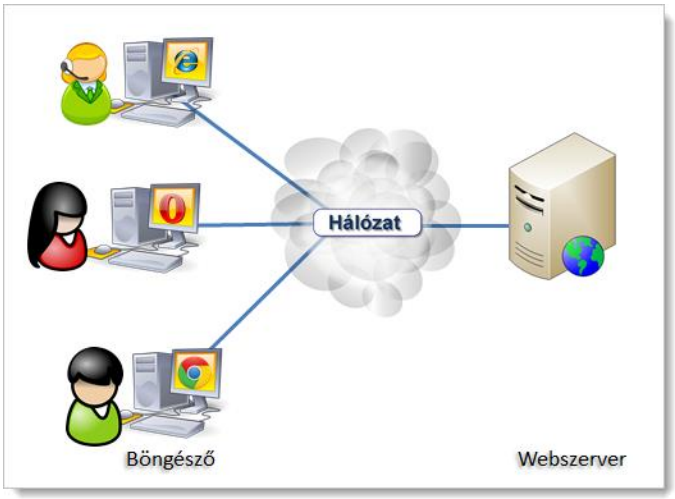
Bár a HyperText Transfer Protocol részletei általában a programozó előtt is rejtve maradnak, a HTTP a szolgáltatás legfontosabb eleme a webprogramozás szempontjából.

A HTTP **szöveges, kérés-válasz alapú, általános, állapotmentes protokoll,** amely a webkliens és -szerver közötti adatforgalom módját, az üzenetek lehetséges tartalmát határozza meg.

A böngésző és a webszerver úgynevezett **üzenetek** formájában kommunikálnak egymással. Az üzenetek **fejrészre** és **üzenettestre** bonthatók. Az **üzenet-**

test különböző **adatokat** (például a felhasználó által begépelt nevet és jelszót, vagy a szerver által visszaküldött fájlt) tartalmaz. A **fejrész** pedig a küldött adatok feldolgozására vonatkozó **utasításokat**, **metaadatokat** hordozza.

- A protokollra vonatkozó **szöveges** jelző arra utal, hogy az **üzenetek fejléce** emberi **értelmezésre alkalmas szövegből** épül fel.
- A **kérés-válasz** alapúság azt jelenti, hogy a **kapcsolatot mindig a kliens kezdeményezi**, egy úgynevezett **kérés**üzenet formájában, amely általában egy weblapra vonatkozik. A **szerver** előkeresi a kért oldalt, és **válaszában visszaküldi** azt a kliensnek.
- Az **általánosság** a kliens és szerver közötti üzenetek üzenettestének tartalmára utal. A HTTP azért általános protokoll, mert az üzenettest tartalmára semmiféle megkötést nem tartalmaz. Ez az a jellemző, amelynek köszönhetően a szerver és a kliens között bármilyen adat továbbítható. Egyszersmind **ez teszi lehetővé, hogy a world wide web alkalmas legyen a távoli gépeken futó alkalmazások, és a felhasználó közötti kapcsolat biztosítására.**
- Az **állapotmentesség** jelentése, hogy a szerver a válasz elküldése után bontja kapcsolatot, és annak utolsó állapotáról semmilyen információt nem tárol. Az **összeköttetés csak a kérés-válasz idején** áll fenn, azt követően mindkét oldal olyan állapotba kerül, mintha sohasem kapcsolódtak volna egymáshoz.
- A kapcsolat akkor is megszakad, amikor például képeket tartalmazó weblapot töltünk le. A letöltést követően a böngésző a weblapban abban talált jelölőknek megfelelően egyenként tölti le, majd jeleníti meg a képeket. Minden kép letöltésekor új kapcsolat épül fel, új kérés indul, amelyet a szerver mindig lezár a válasz elküldése után. A kapcsolatfölépítés, a kérés, a válasz, és a kapcsolatbontás minden egyes kép letöltésekor megismétlődik.
- Egy webszerver általában több kliens konkurens kéréseit szolgálja ki. **A kapcsolatbontás értelme az, hogy egyetlen kliens se tudja hosszú időre lefoglalni a szervert.**



2. ábra Konkurens kapcsolatok

Az állapotmentesség komoly problémákat vet fel a webprogramozásban. Minden alkalmazásra jellemző ugyanis, hogy a felhasználóval folytatott interakciók során adatokat olvasnak be, majd azokat feldolgozva újabb és újabb állapotba kerülnek. Amikor a felhasználó webes alkalmazást használ, a program egy távoli gépen fut, de kezelőfelülete a felhasználó böngészőjében jelenik meg. A program és a kezelőfelület között a webszerver tartja fenn a kapcsolatot. A felhasználó által megadott adatot a böngésző egy kérésben küldi el a webszervernek, ami a válaszában a megváltozott állapotnak megfelelő weboldalt küldi vissza. Ezután azonban elbontja a kapcsolatot és „elfelejti” az állapotot. Amikor a felhasználó újabb adatbevitelét követő kérés megérkezik, a szerver már nem „emlékszik” az előző állapotra.

Ez a gyakorlatban például azt jelentené, hogy amikor megpróbálunk bejelentkezni egy weboldalon, a böngésző elküldi a szervernek a begépelt felhasználói nevet és jelszót, a szerver pedig visszaküldi azt a felületet, amit a bejelentkezés után látnunk kell. Ez eddig rendben van. Mihelyt azonban kattintunk egyet az új felületen, a böngésző ismét kérést küld a szervernek, ami nem tárolja előző állapotunkat, tehát nem tudja, hogy már bejelentkeztünk, így nem engedélyezi elvégezni a következő műveltet.

Természetesen a problémára léteznek megoldások, amelyekről az állapotkezelésről szóló 9. leckében olvashatunk majd.

2.3 A HTTP-ÜZENETEK FÖLÉPÍTÉSE

A webprogramozás szempontjából talán a HTTP a web legfontosabb alapeleme. A szerver és kliens közötti üzenetváltások lehetséges tartalma, az üzenetek fölépítése fontos lehet a web programozásban, ezért vizsgáljuk meg kicsit részletesebben a HTTP üzenetek fölépítését.

A HTTP protokollnak eddig három verziója (0.9, 1.0, 1.1) látott napvilágot. Jelenleg természetesen az 1.1 van érvényben, de az egyszerűbb megértés kedvéért az 1.0-s verziót nézzük meg.

A HTTP **üzentek** a kliens által küldött **kérések** és a szerver által visszaküldött **válaszok** lehetnek. Mindkét üzenettípus alapvetően két részre, **fejlécre** és **üzenettestre** tagolódik.

2.3.1 A HTTP kérés formátuma

A kapcsolat fölépítése mindig a kliens kérésével indul. A kérés fejléce egy, vagy többsoros szöveg, amelynek sorait mindig kocsi vissza-soremelés (CRLF) karakter zárja.

Az üzenet kötelező, első sora az úgynevezett **kéréssor,** amelyet nulla vagy több, úgynevezett **header sor** követhet. Minden header sor egy **fejlécmező**-t tartalmaz. Az üzenet fejlécét, és az azután következő **üzenettestet**, üres sor (CRLF) választja el.

A kérés általános formátuma az alábbi:

```
1  kéréssor<CRLF>
   [header_sor<CRLF>…]
   <CRLF>
   [üzenettest]
```

Legegyszerűbb esetben a kérés egyetlen kéréssorból áll, nem tartalmaz sem fejlécmezőket, sem üzenettestet.

A **kéréssor** tudatja a szerverrel, hogy mit kér tőle a kliens. Első eleme egy **metódus**, melyet egy **URL-cím,** majd a HTTP protokoll **verziószáma** követ. A sor formátuma a következő:

```
1  metódus URL-cím HTTP/verzió
```

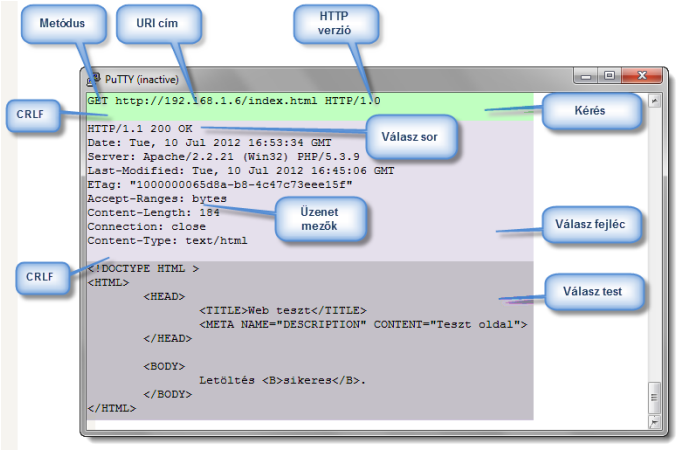
- A *metódus* **GET**, **POST**, vagy **HEAD** szavak valamelyike lehet. A GET egy fájl, a HEAD pedig a fájlról szóló információk visszaküldését kéri szerver-

től. A POST arról tájékoztatja a szervert, hogy az üzenettestben valamilyen, a szerveroldali programnak szóló adatok találhatók.

- Az *URL-cím* a GET vagy HEAD esetén a kért fájl URL-címe, POST esetén az az állomány, ami feldolgozza a küldött adatokat.
- A *verzió* pedig a HTTP protokoll kliens által használt verziójának száma (0.9, 1.0, 1.1).

Az alábbi példáinkban közvetlenül HTTP üzenetek segítségével kommunikáltunk a 192.168.1.6 című hoston futó, 80-as portot figyelő web-szerverrel. Feltételezzük, hogy a szerver főkönyvtárában két állomány az **index.html** nevű weblap és a **calc.php** nevű PHP-program található.

Az üzenetküldéshez a PuTTY nevű alkalmazást használtuk. A PuTTY SSH-kliens, de Telnet-ügyfélként is konfigurálható. A Telnet-kliensként a felhasználó által begépelt karaktereket egy előre beállított host meghatározott portjára küldi, majd megjeleníti a válaszként érkező karaktereket. Mivel a webszerverek alaphelyzetben a 80-as portot figyelik, példáinkhoz úgy konfiguráltuk a PuTTY-ot, hogy a begépelt karaktereket a 192.168.1.6 címen működő gép 80-as portjára, azaz a hoston futó web-szervernek küldje el. Ezekkel a beállításokkal a PuTTY kiválóan alkalmas a HTTP üzenetek tesztelésére.



3. ábra HTTP üzenetek

A fenti ábra „Kérés” feliratú sora (zölddel jelölve) GET metódussal kéri a **192.68.1.6** host webszerverétől a főkönyvtár **index.html** nevű állományát. A

kliens a HTTP 1.0 verzióját használja. A kérés nem tartalmaz sem header sorokat, sem üzenettestet.

2.3.2 A HTTP válasz

A fenti ábra lila színnel jelzett részei az alábbi kérésre érkező válaszüzenetet mutatják:

```
GET http://192.168.1.6 /index.html HTTP/1.0
```

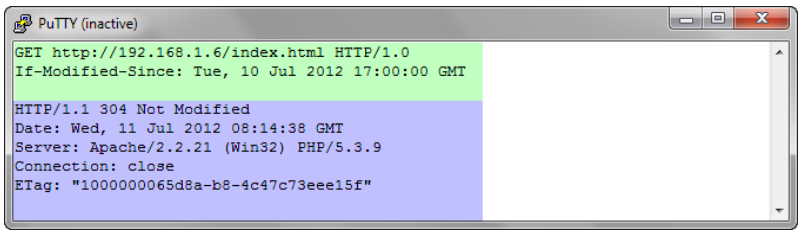
A szerver válasza is fejlécből és üzenettestből áll. A fejléc a **válaszsorral** kezdődik, amit kocsivissza-soremelés (CRLF) karakter zár. A válaszsort tetszőleges számú CRLF-karakterrel záródó, és **fejlécmezőket** tartalmazó header sor követheti. A visszaküldött adatokat tartalmazó **üzenettestet** most is üres sor (CRLF) választja el a fejléctől.

A **válaszsor** a szerver által használt **protokollverzióval** kezdődik (HTTP/1.1), amit a kérés teljesítésének sikerét jelző **válaszkód** (200) és az ahhoz tartozó **üzenet** (OK) követ.

A válaszsor után a **fejlécmezők** következhetnek. Ezek a **serverről** és a visszaküldött **adatokról** tartalmaznak **információkat**, és mindig *név*:*érték* formátumúak.

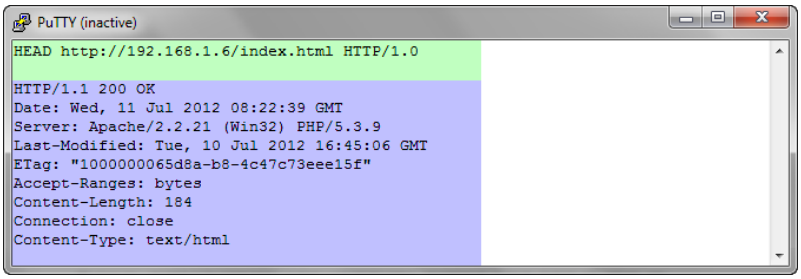
A fenti ábra **`Server: Apache/2.2.21 (Win32) PHP/5.3.9`** fejlécmezője például a webszerver adatait, a **`Date`** a kérés teljesítésének időpontját, a **`Last-Modified`** a kért fájl utolsó módosításának időpontját, az **`ETag`** a fájl ellenőrzőösszegét, a **`Content-length`** a válasz hosszát, az **`Accept-Ranges`** az előző érték mértékegységét, a **`Content-Type`** pedig az adatok típusát adja meg. A **`Connection`** mező jelzi, hogy a szerver le is zárta a kapcsolatot.

A **`Content-type`** fejlécmező szerepe nagyon fontos, hiszen a kliens ebből tudja, hogy milyen adatot kapott az üzenettestben. Ettől függ, hogy hogyan kell feldolgoznia a kapott adatokat.



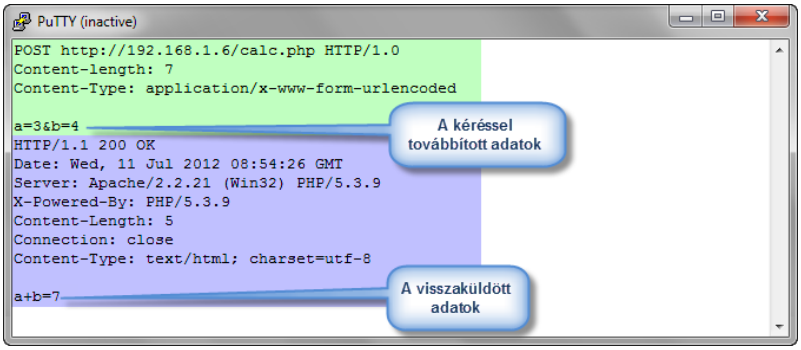
4. ábra Feltételes kérés

A fenti ábra szintén egy kérés-válasz párt mutat. A kérés fejlécben most találunk egy fejlécmezőt is (**`If-Modified-Since: Tue, 10 Jul 2012 17:00:00 GMT`**), ami arra szólítja föl a szervert, hogy csak akkor küldje vissza a kért fájlt, ha az 2012.07.10 17:00:00 óta megváltozott. A szerver a válaszsorban jelzi, hogy az **`index.html`** nem módosult ezért a válasz nem tartalmaz választestet, csak fejlécmezőket.



5. ábra Információkérés

Az ezen az ábrán látható példában csak a megadott fájlról szóló információkat kérünk, a dokumentumot nem akarjuk letölteni. A válasz tehát csak fejlécmezőket tartalmaz, üzenettestet nem.



6. ábra POST kérés

A fenti példa egy POST kérést tartalmaz. Figyeljük meg, hogy az URL-címben most nem egy weblap, hanem egy PHP kódot tartalmazó programfájlt jelöltünk meg (calc.php). Ennek kell majd feldolgoznia a POST-tal küldött adatokat. A kérés üzenettestében két változót és értéküket (a=3; b=4) küldjük el a szervernek. A **`Content-length`** üzenetmező a küldött üzenettest hosszát, a **`Content-type`** pedig az adattípust jelzi. Utóbbira azért van szükség, hogy a szerver tudja, miként kell feldolgoznia az adatokat. Mint tudjuk, a HTTP általános protokoll, az üzenettestben bármilyen adat elhelyezkedhet. Az **`application/x-www-form-urlencoded`** érték jelzi, hogy egy weboldal űrlapjáról származó URL-kódolással leírt változók találhatók az üzenettestben.

A kapott adatokat a szerver átadja a **calc.php** fájlban lévő PHP-programnak, amely feldolgozza azokat (**$c=$a+$b;**), és valamilyen kimenetet állít elő (**echo("{$a}+{$b}={$c}");**). A szerver válassza most nem egy előre elkészített és a szerveren tárolt weblap, hanem a PHP-program által előállított kimenet lesz.

A **calc.php** tartalmát az alábbi sorok mutatják be

```
<?php
  $c=$a+$b;
  echo("{$a}+{$b}={$c}");
?>
```

A PHP programozási nyelvről később tanulunk, az itt leírtak csak a példaként szolgálnak. Ha nem pontosan érti a kódot, ne bosszankodjon, később mindenről részletesen beszélni fogunk!

Elég elfogadnia, hogy a program fogadja a POST kéréssel feltöltött változókat, kiszámolja összegüket amit a $c változóba helyez, majd egy szöveget küld vissza, amelybe behelyettesíti az $a, $b és $c változók értékét.

A PHP programozási nyelvben a változóneveket $ jel előzi meg.

A fenti példaprogram csak akkor működik, ha a PHP konfigurációs fájljában **On** értékre állítjuk a **register_globals** direktívát, ami biztonsági okokból alapértelmezésként **Off** értékű.

2.4 STATIKUS ÉS DINAMIKUS WEBLAPOK

A HTML nyelv tetszőleges szöveges tartalom leírását, a szöveg szerkezetének, formátumának, a szövegben megjelenítendő mediális állományoknak, és a más weblapokra mutató hivatkozásoknak jelölését teszi lehetővé.

A **weblapok statikusak és dinamikusak lehetnek**. **Statikus** weblapról beszélünk akkor, ha annak tartalmát a fejlesztő előre meghatározza, leírja, és tárolja. Az ilyen weblapok tartalma **minden egyes letöltéskor, megtekintéskor ugyanaz**. A statikus weblap csak akkor változhat, ha a fejlesztő megnyitja és „kézzel” átírja a fájlt.

A **dinamikus weblapokat**, vagy azok egy részét nem a fejlesztő, hanem valamilyen **programkód** hozza létre. A program általában adatokat fogad, és azok

feldolgozása után **dinamikusan állítja elő** a weblap HTML-szövegét. A dinamikus weblapok tartalma ezért nem állandó, hanem **a program által feldolgozott adatoktól függ**.

> Az előző szakaszban látott **index.html** állomány statikus weblap, hiszen tartalma állandó. A **calc.php** által előállított rendkívül egyszerű oldal dinamikus weblap, hiszen tartalma a POST-kéréssel elküldött két változó értékének függvényében változhat.

A dinamikus tartalom előállítását, módosítását szerveroldalon, vagy kliens számítógépen futó programkód is végezheti. **Szerveroldali programról** beszélünk akkor, ha a dinamikus weblapot manipuláló **programkód** egy **távoli gépen működik**, és hozzánk már csak a kész weblap jut el.

Kliensoldali programról beszélünk akkor, ha a tartalmat manipuláló **kód** a **weblapba építve**, azzal együtt töltődik le, és utasításait **a böngésző hajtja végre**.

2.4.1 Kliensoldali programozás

A kliensoldali programozásra, a weblapokat manipuláló programok kliensoldali futtatására a HTML 2.0 megjelenése és böngésző programok fejlődése teremtette meg lehetőséget. Míg a HTML 1.0 csak a legalapvetőbb jelölők használatát engedte meg, a **HTML 2.0** számos újítást hozott. Közéjük tartozott, hogy a weblapok szövegébe közvetlenül, vagy hivatkozásokkal **Java, illetve JavaScript programkódokat** lehetett beilleszteni. Az új lehetőségek a böngészőprogramok fejlesztését is szükségessé tették, hiszen a régi kliensek nem tudták értelmezni a HTML 2.0 új jelölőit. Az 1995 után megjelent böngészők zöme már nemcsak a weblap megfelelő megjelenítésére, de a weblaphoz kapcsolt programkód futtatására is alkalmas.

> A modern böngészőket számtalan funkció elvégzésére teszik alkalmassá, sőt, általában úgy készítik őket, funkcionalitásuk bővítésére utólag is legyen lehetőség.

Amikor a böngésző olyan weblapot tölt le, amelyben programkódot talál, akkor kódot átadja a beépített értelmezőnek, ami végrehajtja a program utasításait. Az ilyen programok is alkalmasak a weblap tartalmának manipulálására.

*Az alábbi weblap csak az „**Üdvözöllek!**” szöveget írná ki. A **<SCRIPT>…</SCRIPT>** jelölők közé illesztett JavaScript kód azonban megváltoztatja az oldalt. A böngésző elindítja a kétsoros*

programocskát, amely bekér egy nevet, majd a névnek megfelelően változtatja meg a weblap tartalmát.

```
<!DOCTYPE HTML>
<HTML>
<HEAD>
    <TITLE>Klines oldali program</TITLE>
    <META HTTP-EQUIV="CONTENT-TYPE"
            CONTENT="TEXT/HTML; CHARSET=UTF-8">
    <SCRIPT>
        VAR nev=PROMPT("Mi a neved?","");
        DOCUMENT.WRITELN("Szia <B>" + nev + "</B>!
");
    </SCRIPT>
</HEAD>
<BODY>
    Üdvözöllek!
</BODY>
</HTML>
```

2.4.2 Szerveroldali programozás

A szerveroldali programozás esetén a programkód a szerveren tárolódik, és végrehajtása is ott történik. Ilyenkor nem a böngésző, hanem a távoli gépen elhelyezett alkalmazásszerver futtatja a programkódot.

Az **alkalmazásszerver** olyan keretrendszer (programozási nyelv, futtatókörnyezet, futásidejű könyvtárak, csatolófelületek együttese), amely egymástól függetlenül működő **programkódok párhuzamos futását** biztosítja. Az alkalmazásszerverek kiváló lehetőséget biztosítanak az adatbázis-szerverek által működtetett adatbázisok elérésére, és webalkalmazások készítésére is.

A webalkalmazás olyan szoftver, amelynek kódja távoli gépre telepített alkalmazásszerver segítségével fut, de kezelőfelülete a felhasználó gépén futó böngészőben jelenik meg. Mivel az alkalmazásszerveren futó program képes adatbázis-szerverek elérésére is, a webalkalmazások kiválóan alkalmasak adatbázisok web felületen történő kezelésére.

A webalkalmazások által megjelenített, kezelt adatokat szinte mindig valamilyen relációs adatbázis biztosítja. Adatbázisokban tárolódnak a webes levelezőrendszeren küldött leveleink adatai, a

webboltokban kapható termékek, és a leadott rendelések jellemzői, és a weben keresztül elért tanulmányi rendszer által kezelt adatok is.

A webalkalmazás futása az alábbi séma szerint zajlik:

- A webalkalmazáshoz tartozó programkódokat a webszerver tárolja.
- Az **indításkor** a felhasználó a webszerverrel lép kapcsolatba, de egy statikus weblap helyett **programkódot tartalmazó állományt** kér a szervertől.
- A webszerver a kért dokumentum visszaküldése helyett **fölveszi a kapcsolatot az alkalmazásszerverrel**.
- Átadja neki a programkódot, amelyet az **végrehajt**.
- A végrehajtás eredményeként **weblap keletkezik**, amely a **webalkalmazás pillanatnyi állapotának megfelelő** felhasználói felületnek felel meg. Tartalmazza azokat az adatokat, amelyeket a felhasználónak éppen látnia kell, de megtalálhatók benne azok a vezérlő elemek is (űrlapok, beviteli mezők, nyomógombok, hivatkozások, esetleg kliens oldali programkódok) amelyek biztosítják a felhasználó és az alkalmazás közötti további interakciót.
- Az alkalmazásszerver átadja **weblapot a webszervernek**.
- A **webszerver elküldi** a felhasználó gépére a weboldalt.
- Amikor a **felhasználó** az alkalmazás állapotának megváltoztatását igénylő **műveltet végez** (például valamilyen adatot gépel be, és elindítja annak feldolgozását) a **böngészője ismét a webszerverhez fordul**, átadja az adatokat és a feldolgozásukra alkalmas programkód indítását kéri.
- A webszerver ismét az alkalmazásszerverhez irányítja a kódot, és a program futtatásával **keletkező weblapot küldi vissza** a böngészőnek.

Webszerver alkalmazás szerver kapcsolata

A webszerverek kezdetben egy speciális interfész (**CGI** = Common Gateway Interface) **segítségével kommunikáltak** az alkalmazásszerverekkel. A webszerver kérésére a CGI indította el a futtató környezetet, átadta a végrehatandó programkódot, amit az alkalmazásszerver végrehajtott. Az eredményt a CGI-n keresztül visszaadta a webszervernek, és befejezte működését. A CGI-technológia úgy működött, hogy minden egyes felhasználói kéréskor újra és újra elindult **a futtató környezetet**, **amely önálló folyamatként** futott a feladat végrehajtásáig. Mivel egy webszerver általában több felhasználói kapcsolatot kezel egyidejűleg, a CGI használatával nagyon **megnőhetett a párhuzamosan**

futtatott folyamatok száma, ez pedig erősen **megnyújthatta a válaszidőt**, lelassíthatta a webalkalmazások futását.

A **modern webszervereket** úgy készítik el, hogy azok beépített **modulként** tartalmaznak különböző programozási nyelveken írt programok párhuzamos futtatására alkalmas **futtató környezeteket**. A **modulok** a webszerver indításakor betöltődnek a memóriába, így az egyes kérések esetén nem kell újra és újra elindítani őket. **Egyetlen modul képes kezelni a különböző programkódok párhuzamos futtatását**. Ezzel a technikával jelentősen gyorsul a webalkalmazások futása.

2.5 ÖSSZEFOGLALÁS, KÉRDÉSEK

2.5.1 Összefoglalás

Tananyagunk első leckéjében a webprogramozáshoz szükséges előismereteket tekintettük át. Megismertük a WWW szolgáltatás négy fontos komponensét, a HTML nyelvet, az URL-címeket, a webkliens és szerver szerepét, valamit a HTTP jelentőségét. A lecke arra igyekezett rávilágítani, hogy a web elsősorban a kliens és szerver közötti kommunikáció egyszerűségének köszönheti népszerűségét. A HTTP kérés-válasz alapú, szöveges, általános, állapotmentes protokoll. Szöveges volta lehetővé teszi, hogy egyszerű olvasással is megértsük az üzenetek fejlécét. Az általánosságnak köszönhetően a protokoll bármilyen adat átvitelére alkalmas, így kiválóan megfelel a böngészőben megjelenő kezelőfelület és a távoli gépen futó programkódok közötti kapcsolat fenntartására. A webszerver a kliens kérésére adott válasz visszaküldése után elbontja a kapcsolatot, és „elfelejti" a kliens állapotát. Ha csak egy árucikk megrendelésének folyamatára gondolunk, akkor is belátjuk, hogy a webalkalmazásoknak számos állapota lehet (bejelentkezés, áruk listázása, termék kiválasztása, termék megrendelése, fizetés, visszaigazolás elküldése stb.) A programozást egyedül a HTTP állapotmentessége nehezíti meg. Erre a problémára több megoldás is született, amelyekről - mint olvashattuk - később fogunk tanulni.

A kliens szerver kapcsolatban a böngészők kérésére a webszerver általában weblapok visszaküldésével válaszol. Ezek az oldalak statikusak vagy dinamikusak lehetnek. A statikus oldalak mindig állandóak, a fejlesztő által leírt szöveget és HTML-jelölőket tartalmazzák. A dinamikus oldalakat számítógépes programok állítják elő. Ezek a programok futhatnak a böngésző részét lépező értelmezőben, vagy távoli gépen alkalmazás szerver felügyelete alatt. Az első esetben kliensoldali, a másodikban szerver oldali programról beszélünk.

A webalkalmazásokat alkotó programkódokat a webszerverek a weblapokkal azonos módon, fájlokban tárolják. Amikor felhasználó egy programkódot

tartalmazó fájlt kér, a webszerver átadja a kódot az alkalmazás szervernek és a futás eredményeként keletkező weblapot küldi csak vissza a böngészőnek. A visszaküldött oldal mindig az alkalmazás pillanatnyi állapotának megfelelő kezelőfelület lesz. Tartalmazza az adott pillanatban megjelenítendő adatokat, és a továbblépéshez szükséges vezérlőelemeket.

2.5.2 Önellenőrző kérdések

1. Melyek a WWW alapvető komponensei?
   - A HTML nyelv, az URL-cím, a webszerver és kliens, valamint a HTTP protokoll.
2. Alkalmas volt-e a www első változata a webprogramozásra?
   - A www kezdetben csak néhány formátumot tartalmazó, elosztottan tárolódó, és egymáshoz kapcsolt szöveges dokumentumok letöltésére volt alkalmas. A HTTP protokoll első nyilvános változata a 0.9 még csak a GET metódust ismerte így a kliens kevés adatot tudott eljuttatni a szerverre. A protokoll ezen változatában nem volt mód az üzenettest adattípusának megjelölésére, így csupán weblapokat lehetett letölteni.
3. Mit jelent az, hogy a HTTP általános protokoll?
   - Azt, hogy a protokoll nem tartalmaz semmiféle megkötést arra az üzenettest szerkezetére nézve. A HTTP protokoll segítségével bármilyen adat küldhető kliens és szerver között. Leginkább ez a jellemző teszi alkalmassá a WWW-t arra, hogy weblapon megjelenő felületen alkalmazás szervereken futó programokat vezéreljünk.
4. A HTTP protokoll melyik jellemzője okozza a webprogramozás legtöbb problémáját?
   - Az állapotmentesség.
5. Mi a különbség statikus és dinamikus weblapok között?
   - A statikus weblapokat a fejlesztők készítik, és mindig azonos tartalommal jelennek meg. A dinamikus weblapokat szerver- vagy kliensoldalon futó programok állítják elő, és tartalmuk a program bemeneti adataitól függ.

3. LECKE: SZERVEROLDALI PROGRAMOZÁS PHP NYELVEN

3.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

A szerveroldali programozáshoz számos programozási nyelvet használhatunk, napjaink webalkalmazásainak döntő többsége azonban PHP nyelven készült. Ha programozott háttérrel rendelkező webhelyet kell készítenünk, a PHP több szempontból is jó választás lehet. A programozási nyelv és implementációja, a PHP értelmező ingyenes felhasználású, ugyanakkor professzionális lehetőségeket biztosít a webprogramozásban. Az alkalmazás kezelőfelületének előállításán túl a legkülönbözőbb műveletek elvégzésére, például adatbázis-kapcsolat lebonyolítására, hálózati szolgáltatásokkal való kommunikációra, szerveroldali állománykezelésre, grafikák előállítására, különböző adatformátumok kezelésére és számtalan egyéb feladat lebonyolítására alkalmas. Tananyagunk további leckéiben szinte kizárólag a PHP nyelvvel és lehetőségeivel foglalkozunk. Mai leckénkben megismerheti a nyelv rövid történetét, a szerveroldali programozáshoz szükséges szoftverkörnyezetet, és egy olyan programcsomagot, amellyel Windows operációs rendszerre is telepítheti a PHP használatához szükséges szoftvereket. Ezek mellett megismerhet egy a programkód előállításához szükséges ingyenes fejlesztőeszközzel, a lecke befejező szakaszában pedig a nyelv használatával kapcsolatos alapismeretekről olvashat.

3.2 A PHP TÖRTÉNETE

A PHP programozási nyelv megalkotója Rasmus Lerdorf, dán származású kanadai programozó. A nyelv „ősét” 1994-ben készítette saját munkája megkönnyítésére. Ez a verzió nem önálló programozási nyelv, csupán a weblapok programozott előállítását megkönnyítő, Perl-szkriptekből álló **makrógyűjtemény** volt. Lerdorf 1995-ben **Personal Home Page Tools** néven tette közzé az első publikus változatot. A **PHP**-t még abban az évben kibővítette egy **HTML-űrlap-feldolgozóval** (Form Interpreter) és **miniSql adatbázis-kezelő** rendszer elérését biztosító függvényekkel. A program nevét ez után változtatta **PHP/FI**-re.

1997-ben Lerdorf, Zeev Suraski és Andi Gutmans szinte **teljesen átírta a programkódot**. A PHP hivatalos neve ezzel egy időben **PHP: Hypertext Preprocessor**-ra változott. A fejlesztők ezzel is jelezték, hogy a PHP a fejlesztés nyomán **önálló**, elsősorban **webalkalmazások készítése** alkalmas, **szerveroldali programozási nyelvé** alakult. A nyelv szintaxisa hasonlít a C/C++, Java, JavaScript vagy Perl nyelvekhez, de olyan nyelvi elemeket építettek bele, amelyek

alkalmassá tették a böngészővel feltöltött adatok fogadására, feldolgozására, HTML- és egyéb tartalom dinamikus előállításra.

Az PHP-t azt követően is többször átírták. A 3-as és 4-es verziók után jelenleg a 5-ös sorozat a legelterjedtebb, de már készül a PHP 6 is. A nyelv funkciói egyre bővültek, ugyanakkor hatékonysága jelentősen nőtt. Minden bizonnyal ennek köszönhető, hogy napjainkra a szerveroldali webprogramozás legelterjedtebb programozási nyelvévé fejlődött.

Népszerűsége már 1997-től kezdődően folyamatosan növekedett. Felhasználóinak köre olyan rohamosan bővült, hogy 1999 végére már több mint 1 100 000 webhely működött PHP-kód alapján. 2007-ben közel 21 millió webalkalmazás programozott háttérét biztosította, napjainkra pedig 76 millió felé közelít a PHP-vel támogatott webhelyek száma. A PHP piaci részesedése toronymagasan kiemelkedik a szerveroldali programozási nyelvek mezőnyéből. A közel 77%-s arány jelentőségét jól mutatja, hogy a verseny második helyezettje, a Microsoft ASP.NET környezete is csak a 21%-ot mondhat magáénak.

http://w3techs.com/technologies/history_overview/programming_language

5. link A PHP piaci részesedése

3.3 A PHP SZOFTVERKÖRNYEZETE

A **PHP programozási nyelv implementációja a PHP-interpreter**, az a futási környezet, amely képes **értelmezni** és **végrehajtani** a kód utasításait.

Az évek során számos különböző operációs rendszeren futtatható változat készült, így a PHP használható többek között Linux, Windows, HP-UX, Solaris és OpenBSD operációs rendszereken is. Mivel maga nyelv minden platformon egységes, alkalmas platformfüggetlen alkalmazások készítésére is. A PHP használatával shell szkriptek, és (bizonyos megkötésekkel) önálló, grafikus felületen működő alkalmazások is készíthetők, azonban **elsődleges felhasználása ma is a webalkalmazások késztése, az alkalmazások kezelőfelületének dinamikus, szerveroldali előállítása**.

A fenti funkcióban a négyrétegű kliens-szerver architektúra alkalmazásszerverének feladatát tölti be a PHP-értelmező. A felhasználó gépen futó web

kliens valamilyen webszerverről tölti le a webalkalmazás kezelőfelületét. A webszerver modulként futtatja, vagy CGI-n keresztül indítja el a PHP-értelmezőt. A szerveroldali feldolgozást, és a kezelőfelület előállítását a PHP-program végzi, amely általában egy adatbázis-kezelő rendszer adatbázisban tárolja az alkalmazás adatait.

A PHP számos webszerverrel és adatbázis-kezelő rendszerrel képes együttműködni, jellemző szoftverkörnyezete azonban mégis a **Linux** operációs rendszeren futó **Apache** webszerver, és a **MySQL** adatbázis-kezelő rendszer. Ezt a szoftverkörnyezetet szokták a **LAMP** (Linux+Apache+MySQL+PHP) rövidítéssel jelezni.

A PHP nyelv megismeréséhez, megtanulásához, és a későbbi fejlesztésekhez célszerű saját gépünkön is kialakítani a megfelelő szoftverkörnyezetet. Bár a PHP „éles" használata leginkább LAMP környezetben jellemző, nyelvvel való ismerkedés, és a kezdeti lépések kényelmesebben tehetők meg Windows felületen. Tananyagunk következő szakaszában bemutatjuk a WAMP (Windows+Apache+MySQL+PHP) szoftverkörnyezet, és egy a kialakítására alkalmas telepítő csomagot.

Az Apache, a MySQL és a PHP különálló telepítésére is van lehetőség, azonban a szoftverek megfelelő konfigurálására, működésük összehangolására nem áll módunkban kitérni. Szerencsére számos olyan integrált szoftvercsomag létezik, amely Windows operációs rendszert futtató gépünkön egyszerű telepítőprogram segítségével helyezi üzembe be az AMP szoftvereket. Ilyen például az AppServ, az EasyPHP, a WAMP, vagy az XAMPP programcsomag is.

3.3.1 WAMP telepítése

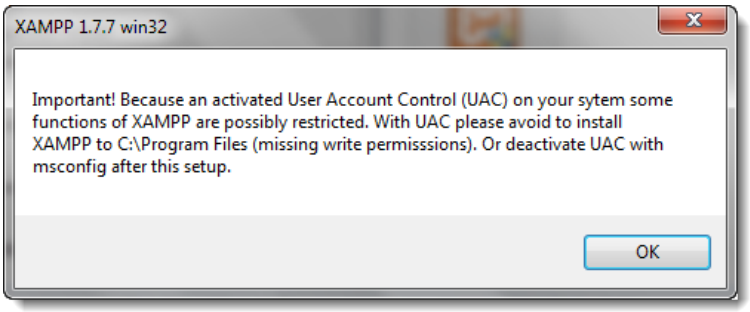
Tananyagunkban az XAMPP használatát javasoljuk, ugyanis a csomag telepítése meglehetősen egyszerű és a telepítőprogram futása után nem csupán az Apache-, MySQL-, PHP-környezet áll rendelkezésünkre, hanem FTP-szervert, SMTP-szervert, sőt, Java webalkalmazások futtatására alkalmas webszervert is találunk majd gépünkön. Habár ezek közül egyelőre csak az AMP-re lesz szükségünk, tanulmányaik folyatásakor jól jöhet a többi alkalmazás is.

http://www.apachefriends.org/en/xampp-windows.html

6. link Az XAMPP telepítő csomagja

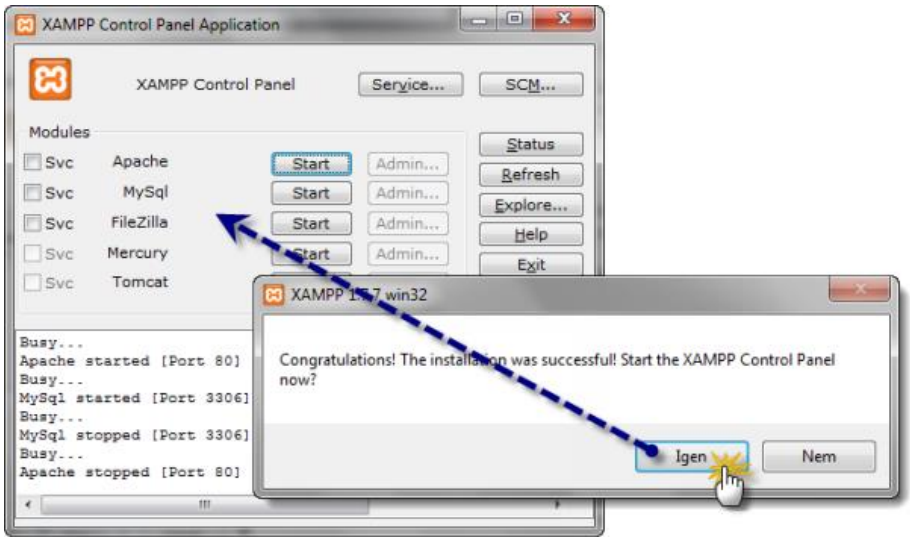
A telepítéshez kövessük az alábbi lépéseket!

- Töltsük le az XAMPP telepítő csomagját!
- Indítsuk el a telepítőt!
- Ha a számítógépünkön be van kapcsolva a Windows UAC szolgáltatása (UAC: User Access Control) akkor azonnal ijesztő üzenetet kapunk, ami azonban csak arról tájékoztat, hogy érdemes a **Program Files** mappa helyett a gyökérkönyvtárba telepíteni a csomagot.



7. ábra UAC esetén kapott üzenet

- Lépjünk tovább! A következő ablakban a telepítő (a fentiek miatt) a **C:\xampp** mappát ajánlja fel a csomag programkönyvtáraként. **Fogadjuk el!**
- Ezután arról dönthetünk, hogy a telepítő létrehozza-e az XAMPP indításához szükséges parancsikonokat, és hogy szolgáltatásként akarjuk-e használni a telepítésre kerülő szervereket.
  Az ikonokra szükségünk lesz, azonban ha gépünket nem dedikált kiszolgálónak szánjuk, akkor a szerverek állandó háttérben futtatása fölösleges. Elég, ha akkor indítjuk el őket, amikor valóban szükségünk van rájuk. **Ne változtassunk a felkínált beállításokon!**
- A telepítés végén megjelenő gratuláció ablakában válaszoljunk igennel a feltett kérdésre, és **indítsuk el** az XAMPP **Contol Paneljét**.



8. ábra Control Panel indítása a telepítés végén

- A panelen a **Start** gombokkal indíthatjuk az egyes szerveket, leállításukra pedig a **Start** helyén megjelenő **Stop** gombokkal lesz lehetőség.
- Indítsuk el az Apache webszervert!

Az futó szervereket a Stop gombokkal állíthatjuk le az XAMPP Control Paneljén, de az XAMPP leállításához nem elég bezárni a Control Panelt. A szabályos bezáráshoz használjuk az **Exit** gombot!

3.3.2 Konfigurálás

Tananyagunk példáit úgy állítottuk össze, hogy ne legyen szükség sem a webszerver, sem PHP, sem pedig a MySQL beállításainak megváltoztatására. A PHP konfigurációs állományra azonban több helyen hivatkozunk majd, ezért érdemes néhány szót ejtenünk a fájlról.

A PHP konfigurációs állománya a **php.ini**, egyszerű szövegfájl. Elhelyezkedése a php telepítésétől függ, de az XAMPP esetében a programcsomag **php** mappájában található. A dokumentum szögletes zárójelek közé zárt elnevezésekkel jelölt szakaszokra bomlik. A szakaszok sorokból állnak, amelyek egy része pontosvesszőkkel kezdődő megjegyzéssor, más részük pedig *tulajdonság* **=** *érték* formátumban megadott konfigurációs beállítás.

A konfigurációs szabályozása jelentheti egy érték megváltoztatását, új tulajdonság beillesztését, esetleg a php.ini egy-egy sorának eltávolítását. A beállítások a web szerver újraindítása után fejtik ki hatásukat.

Mint minden konfigurációs fájlról, a php.ini-ről is érdemes biztonsági másolatot készíteni, mielőtt bármilyen változtatást végzünk a fájlban.

A PHP lehetőséget biztosít egyes beállítások programból történő lekérdezésére, és dinamikus megváltoztatására is. Az erre a feladatra használható **`ini_set()`** és **`ini_get()`** függvényekről a 7. leckében olvashat.

A tananyagunk megfelelő helyein jelezni fogjuk a kapcsolódó beállításokat, de az alábbi hivatkozással elérhető oldalon minden konfigurációs direktíva leírását megtalálja.

http://www.php.net/manual/en/ini.list.php

7. link php.ini direktívái

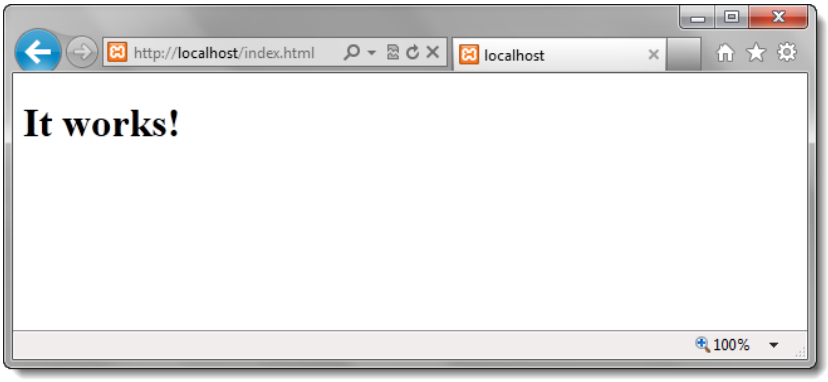
3.3.3 A Webszerver dokumentumkönyvtára

Telepítés után az XAMPP programkönyvtárában (**c:\xampp**) találjuk az összes telepített szerverkomponens könyvtárait, az **apache**, a **php**, a **mysql** és számos egyéb mappát. A webszerver indítása után természetesen nem tudunk majd bármelyik mappából weblapokat letölteni. Webböngészővel csupán egyetlen könyvtár, a **xampp\htdocs** mappa tartalma lesz elérhető. Ez a mappa a webszerver **DocumentRoot** könyvtára, ezt tekinti az Apache a **weben keresztül látható könyvtárszerkezet gyökerének**.

A mappában most (több más fájl és alkönyvtár mellett) megtaláljuk **index.html** nevű weblapot, amit már le is tudunk tölteni a böngésző segítségével.

```
http://localhost/index.html --> It works!
```

Nem jutunk feldolgozhatatlan mennyiségű információhoz, azt azonban megtudjuk, hogy a szerver működik!



9. ábra http://localhost/index.html

A **htdocs** mappa felel meg a gépünkön jelenleg elérhető egyetlen **webhelynek**. Állományainkat akár közvetlenül ebben **a** mappában is elhelyezhetnénk, de természetesen az igazán szép megoldás az lenne, ha saját webhelyet hoznánk létre, amelyre a **localhost** helyett saját névvel lehetne (például: **webprogramozas.ektf.hu)** hivatkozni.

Természetesen erre van is lehetőség, azonban a saját webhely létrehozásához a névfeloldás alapját biztosító **hosts** állomány szerkesztésére, és a webszerver konfigurációs állományának átírására lenne szükség. Az úgynevezett virtuális hostok létrehozása helyett most egy jóval egyszerűbb megoldást választunk. A **htdocs** mappában alkönyvtárt hozunk létre **webprog** névvel, és ebben helyezzük el fájljainkat. A mappa tartalmát ezt követően a **http://localhost/webprog** címmel fogjuk elérni.

Feladat:

- A **htdocs** könyvtáron belül hozzon létre a **webprog** nevű mappát!
- A másolja a **webprog** mappába az leckékhez kapcsolódó forráskódokat tartalmazó ZIP-fájlt és újabb mappa létrehozása nélkül bontsa ki az archív állományt!
- Minden lecke forrásai külön mappákban jelenek meg.
- Próbálja böngészővel megtekinteni a **lecke3/index.html**-t! **http://localhost/webprog/lecke3/index.html**
- Töltse le a böngészővel **test.php**-t is! **http://localhost/webprog/lecke3/test.php**

10. ábra index.html és test.php

index.html
test.php

3.4 PROGRAMOZÁSI KÖRNYEZET

Miután föltelepítettük és kipróbáltuk a webszervert, szükségünk lesz a weblapok és PHP-programok megírására alkalmas szerkesztőprogramra. A célra valójában bármilyen szöveg editor alkalmas (Jegyzettömb, NotePad++, TextPad, EditPlus,…) azonban sokat segíthet a munkában, ha kifejezetten programozásra fejlesztett alkalmazást használunk. Az úgynevezett IDE (Integrated Development Environment) kategóriába tartozó alkalmazások között is nagy a választék. Számos kereskedelmi forgalomban kapható, és ingyenesen használható rendszer között válogathatunk. Tananyagunkban a **CodeLobster** nevű, ingyenesen használható változattal is rendelkező PHP-szerkesztőt fogjuk használni.

http://www.codelobster.com/download.html

8. link A CodeLobster letöltése

3.4.1 IDE-eszköz telepítése és konfigurálása

A CodeLobster telepítése nem jelent problémát. Indítás után csak követni kell az utasításokat, beavatkozás sehol sem szükséges.

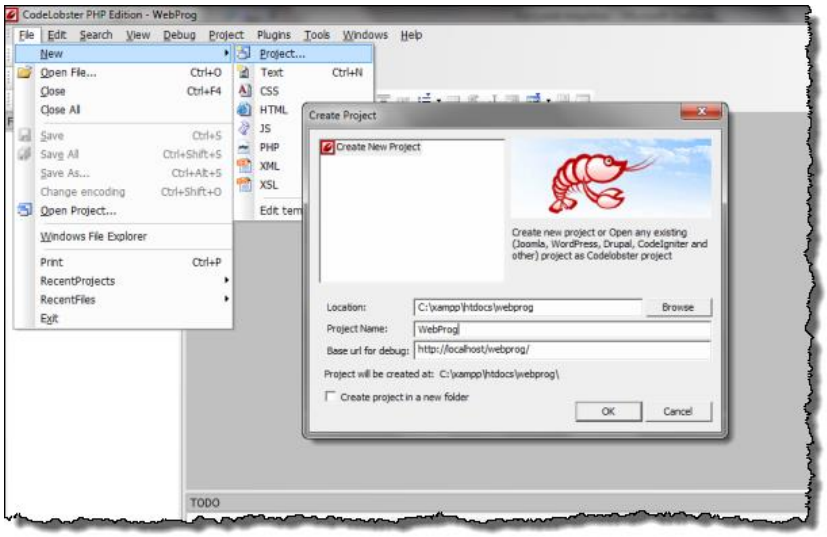
Feladat:

- Töltse le, telepítse, majd futtassa a CodeLobster IDE-eszközt!

A telepítés után a program automatikusan elindul. Azonnal alkalmas bármilyen szöveges állomány szerkesztésére, a weblapok, JavaScriptek, CSS-stíluslapok, XML-állományok és PHP-kódok készítését pedig számos segédeszközzel támogatja.

A Codelobster-t használhatjuk egyedi állományok, vagy teljes webhelyek szerkesztésére is. Utóbbi esetben minden webhelyhez úgynevezett **projektet** kell rendelnünk. A projektek segítségével könnyebben kezelhetők a webhelyek objektumai, egyszerűbben hozhatunk létre az új fájlokat, megnyithatjuk, szerkeszthetjük a meglévő állományokat, kipróbálhatjuk a weblapokat, tesztelhetjük a programkódokat.

Új projekt létrehozását a **File/New/Project** paranccsal kezdhetjük meg. A megjelenő ablakban meg kell adni a projekt nevét (**Project name**), azt, hogy fizikailag melyik mappában van a webhely (**Location**) és, hogy milyen URL-címmel érhetjük el a mappa tartalmát (**Base url for debug**).



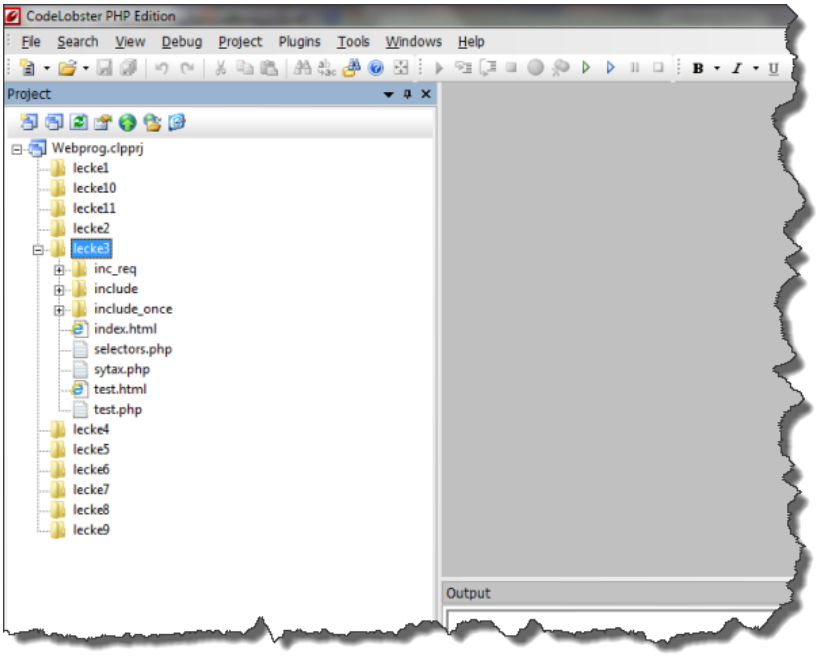
11. ábra Új projekt létrehozása

Feladat:

- Hozzon létre új projektet az alábbi beállításokkal!
- Location: **c:\xampp\htdocs\webprog**
- Project Name: **WebProg**
- Base url for debug: **http://localhost/webprog**

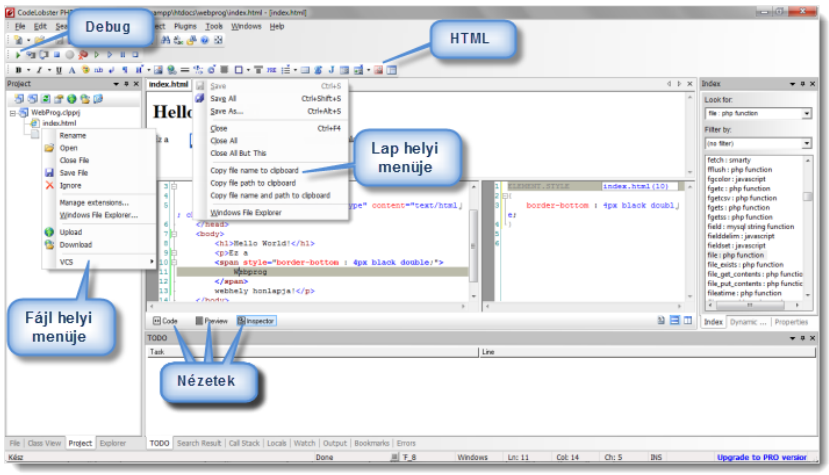
A projekt létrehozása után a képernyő bal oldalán felbukkanó **Project** ablakban megjelennek **Location** tulajdonságban megadott webhely fájljai, és alkönyvtárai, amelyeket legegyszerűbben a helyi menüvel kezelhetünk, de az **Open** parancs dupla kattintással is helyettesíthető.

- Nyissa meg a lecke3 mappát!



12. ábra A Project ablak tartalama

A megnyitott állományok különböző lapokon jelennek meg. Az egyes lapokkal a kiválasztásukra használható a lapfülek helyi menüiben végezhetünk műveleteket (pl. bezárás, tartalom mentése…) A lapok tartalmának forrása a **Code**, előképe pedig a **Preview** ablakban tekinthető meg. Elsősorban HTML-állományok megjelenítésekor használható **Inspector** ablak, amely egyszerre mutatja az előző két nézetet, és segít az előképben kijelölt elemek forrásának azonosításában, az azokhoz kapcsolódó CSS-formátumok szerkesztésében.



13. ábra A CodeLobster felülete

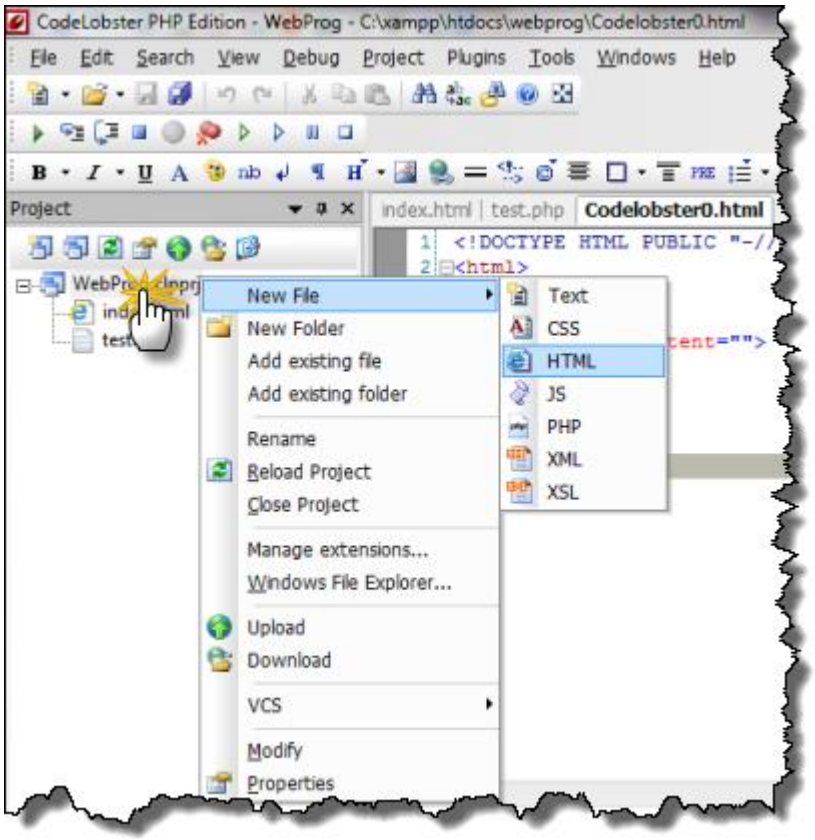
Az állományok előképe nemcsak a **Preview** lapon, hanem közvetlenül a böngészőben is megtekinthető. Ehhez a **Debug** eszköztár **Debug** gombját, vagy az **F5** billentyűt kell lenyomnunk.

Feladat:

- Nyissa meg az **index.html** állományt, és a **test.php**-t is!
- Nézze meg az **test.php** előképét, majd váltson vissza **Code** nézetbe!
- Nézze meg az **index.html**-t az **Inspector** ablakban!
- Figyelje meg, mi történik, ha az előképben, vagy a forrásban rákattint valamilyen elemre!
- Kattintson a **Webprog** szóra!
- Írja át az **Inspector** ablakban a **black** szót **blue**-ra! Figyelje a változást!
- Nézze meg az **index. html** tartalmát böngészőben is!
- Mentse a változásokat!

Új fájl létrehozásához ne használjuk a fájl menüt, mert hiába mentjük a fájlt a megfelelő helyre, nem fog megjelenni a **Project** ablakban!

A webhely új állományát a **Project** ablakban látható projektnév **helyi menüjében** készítsük el! A **New file** parancs almenüt nyit meg, amelyben kiválaszthatjuk a fájl típusát (Text, CSS, HTML, JavaScript, PHP, XML, XSL). Az így létrehozott fájl azonnal megjelenik a **Project** ablakban.



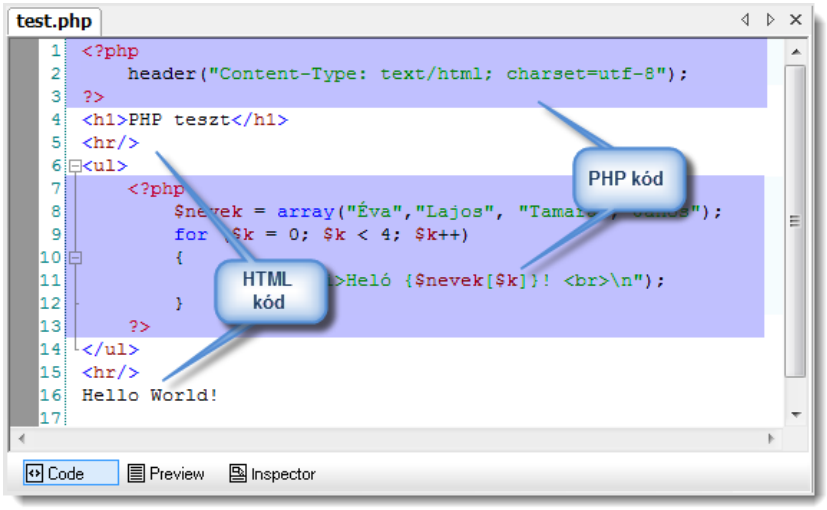
14. ábra Új fájl létrehozása

3.5 A PHP ÁLLOMÁNYOK

- Nyissa meg a test.php állomány!

A legtöbb programozási nyelvben az utasítások nem keveredhetnek más nyelvek kódjával, ami érthető is, hiszen az értelmező nem tudná eldönteni, melyik utasítást kell feldolgoznia és melyiket nem.

A PHP ebben a tekintetben kivételnek számít, ugyanis a programkód HTML-oldalakba ágyazható, sőt, az értelmező eleve abból indul ki, hogy a programsorokat HTML-szöveg veszi körül.



15. ábra HTML- és PHP-kód egy állományban

3.5.1 PHP fájlok kiterjesztése

A webszerver a **HTML**, vagy **HTM** kiterjesztés alapján különbözteti meg a „tisztán” HTML-állományokat, és a **PHP** kiterjesztésről ismeri fel azokat, amelyek PHP-kódot is tartalmaznak. A szerver a böngészők HTML-oldalakra vonatkozó kéréseire azonnal a weboldal elküldésével válaszol, a PHP-állományokat azonban **sohasem továbbítja közvetlenül a böngészőnek**. A programkódot tartalmazó állományokat előbb a **PHP-értelmező (interpreter)** kapja meg, ami feldolgozza a fájl, és **visszaadja** a webszervernek az utasítások által létrehozott **kimenetet**. A webszerver ezt a kimentet küldi vissza a böngészőnek.

A PHP-állományok értelmezésekor az interpreter **HTML- és PHP-módban** képes feldolgozni a szöveg egyes részeit.

A **HTML-módban** feldolgozott szöveg karaktereit egyszerű szövegként kezeli, ezért **változtatás nélkül a kimenetbe másolja**. A **PHP-mód**ban feldolgozott szövegrészeket azonban programkódnak tekinti, **értelmezi** és **végrehatja a talált utasításokat**. Az utasítások számtalan műveletet végezhetnek. **A PHP program** használhatja a fájlrendszerben elhelyezett fájlokat, kapcsolódhat adatbázisokhoz, hálózati kommunikációt folytathat, és adatokat, **HTML-részleteket írhat a kimenthez**. A teljes kimenetet tehát úgy lehet elképzelni, mintha a PHP-kódrészleteket helyettesítenénk azzal, a szöveggel, amit az utasítások kiírnak.

3.5.2 PHP-jelölők

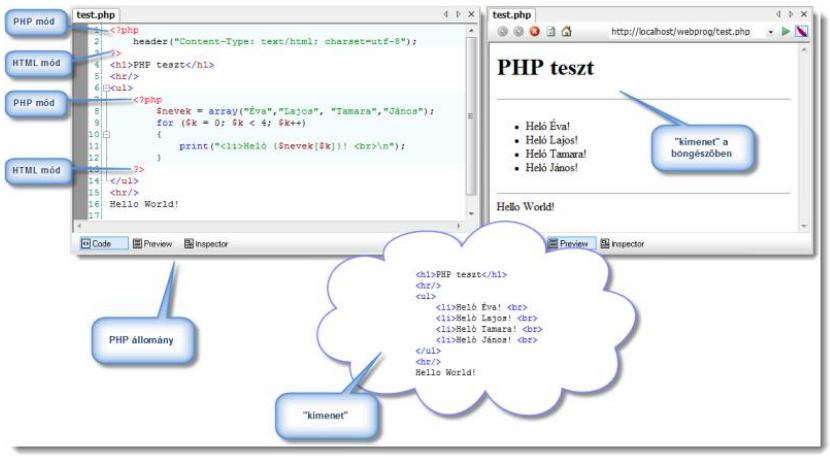
Természetesen ahhoz, hogy az értelmező a megfelelő pontokon tudjon váltani a két mód között, világosan jelölni kell a PHP-kódot tartalmazó részeket. Ennek érdekében a **program utasításait speciális jelölők** közé kell zárnunk. A legáltalánosabb jelölés szerint **<?php** karaktersorozat jelzi a PHP-kód elejét, a **?>** pedig a végét.

A HTML-jelöléséhez közelebb áll a **<script language="php">... </script>** jelölési mód, amit szintén használhatnak, de hosszúsága miatt ritkán alkalmaznak a programozók. A PHP-értelmező a **php.ini** nevű konfigurációs állományban tárolt beállításoktól függően további jelölőket is elfogadhat (**<? ... ?>, <% %>**). Tananyagunkban az első, **<?php**…**?>** jelölést alkalmazzuk.

selectors.php

Az interpreter **mindig HTML-módban kezdi** egy állomány feldolgozását, azaz az abból indul ki, hogy a dokumentum alapvetően HTML-szöveg, amelybe egy vagy több ponton PHP-kódot ágyaztak.

PHP-módba akkor vált át, amikor a szövegben a PHP-kód kezdetét jelző karaktersorozatot talál. Amikor a feldolgozásban eléri a kód végét jelző jelölőt, akkor ismét HTML-módba vált vissza. A PHP kiterjesztésű állomány végrehajtásakor a HTML-PHP-HTML váltás tetszőleges számban ismétlődhet, tehát a fájlban bármennyi PHP-kódrészlet lehet.



16. ábra PHP-állomány feldolgozása

3.5.3 A PHP-program szintaktikája

A PHP programok **pontosvesszővel** (;) zárt utasításokból épülnek fel. Az utasításokat általában egymás alá írjuk, tehát több sorban helyezzük el. Ez ugyan nem kötelező, de mindenképpen ajánlott, hiszen a program így könnyebben olvasható. Az utasítások esetleges paramétereit az **utasítás neve után vesszőkkel elválasztva adjuk meg**. A szükséges paraméterek száma az utasítás specifikációjában ismerhető meg. A függvénynek paramétereit mindig kerek zárójelek között kell megadni.

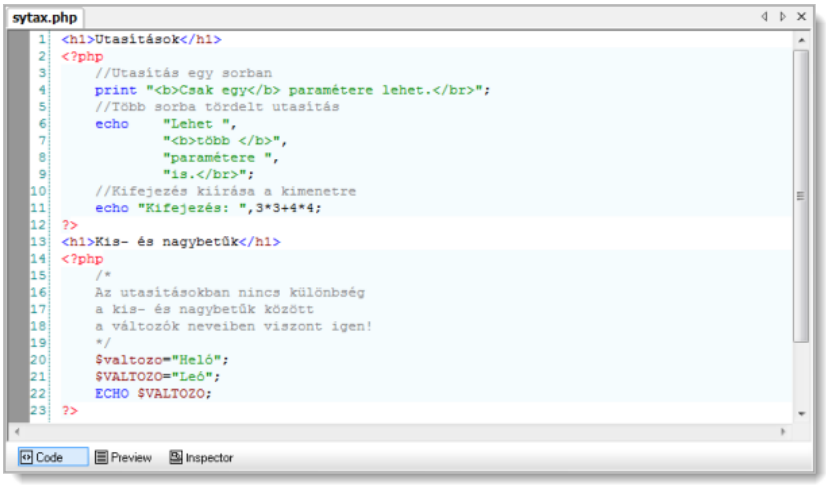
A nyelvi elemek többsége egyszerű utasítás, de a ciklusok, elágazások, eljárások más utasításokból álló blokkokat tartalmazhatnak. Az utasításblokkokat francia zárójelek közé zárjuk **{…}**.

A nyelvi elemeket elválasztó karakterek (szóköz, tabulátor, sorköz) **whitespace**-nek minősülnek, azaz nem jelent hibát, ha egy szóköz helyett tabulátort, vagy sorközt helyezünk el, illetve, ha többszörözzük az elválasztókat. A whitespace-karakterekkel **jól tagolhatjuk a programkódot**, így tovább növelhetjük az olvashatóságot.

A PHP az utasítások neveiben nem, de a változók nevében tesz különbséget **kis- nagybetűk** között. Az **echo** (kimenetre író utasítás) azonos jelentésű az **ECHO**-val, de a $**valtozo** nem azonos a $**VALTOZO**-val.

A programkódban egy- vagy többsoros megjegyzések helyezhetők el. Az egysoros megjegyzések **//** jellel kezdődnek, a többsorosokat pedig **/* */** karaktersorozatok közé kell tenni. A megjegyzéseket az értelmező figyelmen kívül hagyja, azonban a programozó számára fontos információkat tartalmazhatnak, aminek különösen a nagy terjedelmű alkalmazások áttekintésekor, illetve a régebben írt programkódok szerepének felidézésekor lehet nagy jelentősége.

syntax.php



17. ábra PHP kód

Itt érdemes említést tenni az **phpDocumentator** eszközökről, amelyek a PHP-kód többsoros megjegyzéseit használva automatikusan képes dokumentációt generálni az alkalmazásunkhoz.

(http://phpdocu.sourceforge.net/howto.php)

9. link phpDocumentator leírása

3.5.4 Adatok kiírása a kimenetre

Bár a PHP-programkód számtalan művelet elvégzésére alkalmas, az eredményeket azonban általában a kimentbe írja. A kiírásra a **`print`** illetve az **`echo`** utasításokat használhatjuk. Mindkettő a kimenetre írja az utasítás neve mellett feltűntetett adatot, azaz a kapott paramétert.

A különbség az, hogy a **`print`** csak egy, az **`echo`** pedig vesszőkkel elválasztott, több paraméter kiírásra is alkalmas.

```
1 <?php
    echo "Lehet ", "több ", "paramétere ", "is.";
    print "Csak egy paramétere lehet.";
  ?>
```

Mindkét utasítás paramétere lehet kifejezés. Ilyenkor a kifejezés eredménye kerül a kimenetre.

```
1 <?php
    echo "Kifejezés: ",3*3+4*4;
  ?>
```

A kiíró utasításokkal kiírt szövegek egyszerűen, egymást követően, minden formázás nélkül kerülnek a kimentre. Ha formázásukról, vagy akár többsoros megjelenítésükről gondoskodni akarunk, akkor HTML-jelölőkbe ágyazva kell kiírni őket!

```
1 <?php
    echo "Lehet ", "<b>több </b>", "paramétere ",
"is.</br>";
    print "<b>Csak egy</b> paramétere lehet.</br>";
    echo 3*3+4*4;
  ?>
```

A weblapok HEAD-elemében mindig jelezzük a weblap kódolását. Ha tisztán PHP-kóddal állítjuk elő a weblapot, akkor a PHP **header()** függvényével tudjuk elküldeni a kódolás típusát.

Példáinkban mindig UTF8 kódolást használunk, ezért a példaprogramokban – amennyiben a példa nem tartalmazza a HTML HEAD elemét – szerepelni fog a **header()** függvény.

```
header("Content-type: text/html; charset=utf-8");
```

A függvénnyel beállíthatjuk a kimenet kódolását, ami a válasz fejlécmezőjeként jut el a klienshez. Éppen ezért fontos, hogy a függvényt az első kimenetbe írás előtt szerepeltetni kell a programban!

3.5.5 A PHP-alkalmazás

A teljes **webalkalmazás** szinte **sohasem egyetlen PHP-fájlból** áll. A programkódot az alkalmazás tervezésekor fölmért feladatoknak és várt állapotoknak megfelelően **több különálló fájlban** tároljuk. A kliensoldali felület mindig **az alkalmazás aktuális állapotának**, és a felhasználó által kiadott utasításnak **megfelelő PHP-állományt** kéri a szervertől. Más fájlban tárolt kód állítja elő például az alkalmazás címoldalát, a bejelentkező felületet, vagy a felhasználói profil beállítására alkalmas oldalt. Az is jellemző gyakorlat, hogy az alkalmazás több különböző pontján is szükséges műveleteket függvényekkel valósítják meg, és a függvényeket különálló PHP-állományokban tárolják.

Az ilyen módon tagolt kód esetén gyakran van szükség arra, hogy több, különálló PHP-állományt ideiglenesen egy egységbe fűzzünk, és az így kapott kóddal kezeljük egy állapotot. A különálló fájlok ideiglenes összefűzésére használhatók az **`include`**, az **`include_once`** a **`require`** és a **`require_once`** utasítások.

Az **`include`**, és a **`require`** utasításoknak egyaránt a csatolandó PHP-fájl útvonalát és nevét kell megadni paraméterként. Amikor az értelmező **`include`** utasításhoz ér, akkor felfüggeszti az eredeti kód végrehajtását, és paraméterként megadott állomány feldolgozásával folytatja a munkát. Amikor végzett a becsatolt feldolgozásával, visszatér az eredeti kódba és abban dolgozik tovább.

A becsatolt fájlok kiterjesztése bármi lehet (akár HTML is), ugyanis az **`include`** végrehajtása előtt az interpreter mindig HTML módba vált, visszatéréskor pedig ismét PHP-módban folytatja a munkát. Pontosan ezért az **`include`**-dal **a csatolt állományban világosan jelezni kell a PHP-kódok helyét!** A PHP-jelölőkre akkor is szükség van, ha az include állományban csak PHP-kód van.

Előfordulhat, hogy az értelmező nem találja az **`include`**, vagy a **`require`** utasításokban megadott fájlt. A két utasítás között az ilyen esetek kezelésében van különbség. Az **`include`** ugyanis ilyenkor csak **figyelmeztetést** küld, de a program folytatódik (*lásd később: a Hibakeresés, hiba- és kivételkezelés leckében*), a **`require`** azonban **megszakítja** a program futását, és **hibaüzenetet** küld.

```
include/a.php
include/b.php
include/c.php
```

Ettől eltekintve a két utasítás azonos módon működik. Végrehajtásuk abban megegyezik, hogy alkalmasak ugyanannak a kódnak a többszöri beillesztésére is.

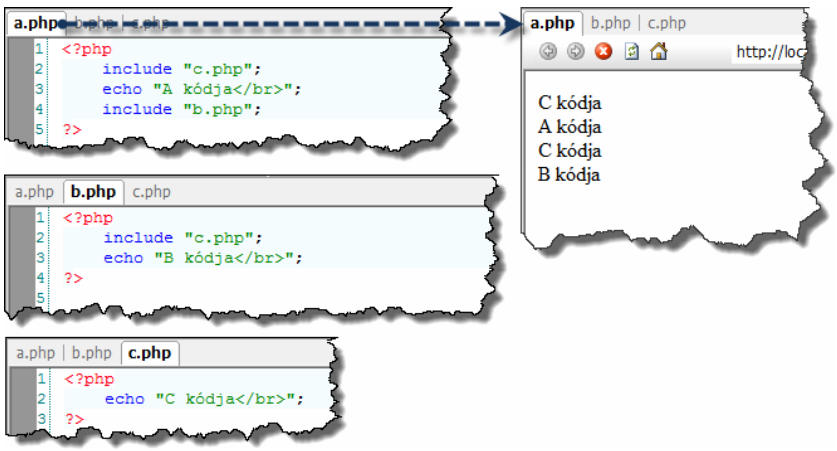
*Tételezzük fel, hogy három különböző kódot tartalmazó PHP-fájlunk van: **a.php**, **b.php** és **c.php**.*

*A **c.php** olyan kódot tartalmaz, amelyre az **a.php** és a **b.php** futásához is szükség van.*

*Az **a.php** kódját is úgy írjuk meg, hogy tartalmazzon egy **c.php**-t beillesztő sort, és a **b.php**-t is hasonlóan készítjük el.*

*Tegyük fel, hogy az **a.php**-nek nem csak a **c**-re hanem a **b.php**-re is szüksége van, tehát valahol tartalmaz egy **b.php**-t csatoló sort is.*

*Ezzel a lépéssel **c** kódja kétszer is bekerül az **a**-ba, hiszen közvetlenül, és **b**-n keresztül is csatoltuk.*



18. ábra Az include működése

Ha meg akarjuk előzni egy kód többszöri beillesztését (egyes esetekben erre feltétlenül szükség van), akkor használhatjuk az **include_once** és a **require_once** utasításokat. Ezek úgy működnek, mint az **include**, vagy a **require** azzal a különbséggel, hogy mielőtt elvégeznék a műveletet, megvizsgálják, hogy kódot korábban nem illesztettük-e be. Ha igen, akkor nem ismétlik meg a csatolást.

```
include_once/a.php
include_once /b.php
include_once /c.php
```

Ha biztosak akarunk lenni abban, hogy nem történik többszörös beillesztés, akkor használjuk az **include_once, require_once** utasításokat!

- Hasonlítsa össze a webhely include, és include_once mappáiban lévő fájlokat! Nézze meg az két a.php futásának eredményét!

3.6 ÖSSZEFOGLALÁS, KÉRDÉSEK

3.6.1 Összefoglalás

Mai leckénkben a PHP történetének rövid ismertetése után a szerveroldali programozásra használt nyelv lehetséges szoftverkörnyezetével ismerkedtünk meg. Letöltöttünk és telepítettünk a **WAMP** környezet kialakítására alkalmas csomagot, amelynek eredményeként számítógépünkön működőképes **Apache** webszerver és **MySQL** adatbázis-kezelő rendszer is rendelkezésre áll. A webszerver dokumentumkönyvtárában saját mappát hoztunk létre, amely alkalmas a példaállományok tárolására.

A szerverek telepítése után üzembe helyeztük a **CodeLobster** nevű IDE-alkalmazást, amely kiváló lehetőségeket biztosít a programkódok megírására és kipróbálására.

A lecke második felében a PHP-program szerkezetéről, a PHP- és HTML-kódok elkülönítéséről tanultunk. Megtanultuk, mit jelent a PHP és HTML feldolgozási mód, hogyan jelezhetjük a PHP-kód kezdetét és végét, hogyan állítja elő a PHP-interpreter a kimenetet.

Megtanultuk az alapvető szintaktikai előírásokat, megismertük az adatok kiírására alkalmas **echo** és **print** utasításokat, végül megtanultuk, hogyan lehet összekapcsolni egymással a több állományra tagolt kód részleteit.

3.6.2 Önellenőrző kérdések

1. Mi történik, ha a böngésző PHP kiterjesztésű fájlt kér a webszervertől?
   - A webszerver átadja a fájlt a PHP-interpreternek és csak a visszakapott kimenetet küldi el a böngészőnek.
2. Mit jelent a WAMP rövidítés?
   - A Windows+Apache+MySQL+PHP szoftverkörnyezetet.
3. Hogyan értelmezi a PHP-interpreter a PHP-állományokat?
   - Mindig HTML módban kezdi a feldolgozást, és mindaddig a kimenetre másolja a szöveget, amíg nem talál kezdő PHP jelölőt. Ilyenkor PHP-módba vált, értelmezi és végrehatja az utasításokat, és csak az echo illetve print utasításokkal kiírt szöveget küldi a kimentre. A befejező PHP-jelölő elérése után újra HTML-módba vált az értelmező. Ha több PHP-jelölőpár van a programban, akkor a váltás minden alkalommal megtörténik.
4. Írhatunk-e egy sorba több PHP-utasítást?
   - Igen. Az utasításokat pontosvesszőkkel kell zárni, de egy sorban több ilyen utasítás is lehet. Ugyanakkor érdemes kerülni az ilyen megoldást, és célszerű jól átlátható, világosan tagolt kódot írni. Ettől a programunk nem lesz hatékonyabb, de könnyebben tudjuk olvasni a kódot. A megfelelő tagolás mellett érdemes megjegyzéseket is tudunk elhelyezni a programban.
5. Mi a különbség az include és a require utasítások között?
   - Az include csak figyelmeztetést küld, ha nem találja a megadott fájlt, a require hibaüzenetet ad és le is állítja a program futását.

Feladat

Készítsen hello.php nevű, Hello World!, Hello Everybody! szöveget megjelenítő PHP-programot! A két mondat külön sorokban jelenjen meg a böngészőben, a World és az Everybody szavak legyenek félkövérek!

4. LECKE: LITERÁLOK, VÁLTOZÓK, TÖMBÖK

4.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

A számítógépes programokra hajlamosak vagyunk úgy gondolni, mint egymást követő műveletek, utasítások sorozatára, amelyeket a programozó a program megírásakor megfelelő formában rögzít, a számítógép (a számítógépen futó értelmező program) pedig egyenként végrehajt. A programok valóban így működnek, azonban ebben a megközelítésben kevéssé koncentrálunk a lényegre, azaz az utasítások végrehajtásának céljára. Miért hajt végre utasításokat a számítógép? Mi történik az utasítások végrehajtásakor, mi a program futtatásának célja.

Ha nagyon általánosan akarunk fogalmazni, akkor azt mondhatjuk, hogy egy program végrehatása során a számítógép egy kezdőállapotból kiindulva egy másik, befejező állapotot állít elő. A kezdőállapot a feladat, a befejező állapot a feladat megoldása. A számítógép számára minden egyes állapot adatokkal írható le, tehát amikor a számítógépes program dolgozik, egy adatokkal leírható állapotból egy másik, szintén adatokkal leírható állapotot állít elő. Eszerint a program tevékenysége az adatok megváltoztatásaként, azaz az adatkezelésként is felfogható.

Ha a program egy kezdőállapotból befejező állapotot állít elő, akkor a program egyes utasításai ennek az átalakításnak a részlépéseit, tehát szintén adatátalakítást, vagy ahhoz kapcsolódó műveletet végeznek.

Ahhoz, hogy az adatkezelés megvalósulhasson, megkerülhetetlen, hogy a programozó adatok formájában tudja ábrázolni az egyes állapotokat, és az adatokat úgy tárolja, hogy a program utasításai átalakításokat tudjanak végezni rajtuk.

Ez a megközelítés azért szerencsés, mert rávilágít arra, a programozás lényeges eleme az adatábrázolás, az adatatok megfelelő, számítógépes programmal kezelhető formában történő tárolása.

Mai leckénkben erről fogunk beszélni. Azt tanuljuk meg, hogyan ábrázolhatók, egyben hogyan tehetők feldolgozhatóvá az adatok a PHP programkódokban. A lecke ismerteti a PHP literáljainak, változóinak, konstansainak ábrázolási módját, bemutatja a használható elemi adattípusokat, és kiemeli a PHP univerzális, összetett adattípusát, a tömböt, illetve ismerteti az adatfeldolgozás elemi formáját a kifejezések írását.

4.2 LITERÁLOK

Az adatábrázolás legelemibb módja, amikor a programkód szövegében közvetlenül helyezünk el értékeket úgy, hogy azt azonnal föl is használjuk. Programkódban tárolunk adatot a például az alábbi utasítás leírásakor:

```
echo "Hello World";
```

Az értéket rögtön fel is használjuk az **echo** utasításban. Arra utasítjuk a PHP-értelmezőt, hogy egy adatot, mégpedig a **"Hello World"** szöveget írjon a kimenetre.

> **A programkódba közvetlenül begépelt, a programszövegben tárolt adatokat értékeknek vagy literáloknak nevezzük.**

Bár első pillantásra puszta értékeknek tűnnek, a literáloknak van egy fontos jellemzőjük, mindig meghatározott **adattípusok** van. A PHP-ben szám, szöveg és logikai típusú literálokat helyezhetünk el a kódban.

4.2.1 Szöveg literálok és ábrázolásuk

literals.php

A programban, különösen a képernyőre kerülő üzenetek miatt gyakran használunk szöveg literálokat. A PHP a legtöbb programozási nyelvhez hasonlóan, az **egyszeres (’), vagy dupla idézőjelek (")** között elhelyezett karaktereket tekinti szövegnek. A szövegliterálban bármilyen karakter előfordulhat, tehát bármit kiírhatunk a kimenetbe, azonban **a páros idézőjelek között elhelyezett szöveg** bizonyos karakterei speciális jelentéssel bírnak. A PHP az így megadott szöveget **kiértékeli** és az az abban elhelyezett speciális karaktereknek megfelelően **átalakítja**, így ezekkel a jelölésekkel formázott kimenet készítésére van lehetőség.

A speciális karakterek **** jellel kezdődnek, és valamilyen műveletre szólítják fel az értelmezőt:

\n helyére az interpreter újsor karaktert (ASCII 10) illeszt a kimenetre küldött szövegbe,

\r karakter hatása hasonló, de most kocsivissza karakter (ASCII 13) kerül a kimenetbe,

\t tabulátorkarakter beszúrását eredményezi,

**** backslash karaktert szúr be a szövegbe,

\" idézőjelet helyez el a szövegben,

\$ dollárjelet helyez el szövegben.

A \n és \r karakterekre azért van szükség, mert a Linux az ASCII 10, a Macintosh operációs rendszerek pedig a ASCII 13-t használják az új sor jelölésére. A Windowsban a két karakter együtt (ASCII 10, ASCII 13) jelzi egy sor végét. A \n,\r karaktereket attól függően használjuk, hogy milyen operációs rendszerre szánjuk a kimenetet.

A speciális karakterek által kifejtett hatás a kimentben, tehát a webböngészőnek visszaküldött weblap HTML-forrásában látható. A \r, a \n és a \t whitespace karakterek, tehát weblapon nem jelennek meg.

A weblap megjelenését csak a kimenetbe írt HTML-jelölőkkel befolyásolhatjuk!

A változók használatáról szóló szakaszban látni fogjuk, hogy a PHP-ban a változók neve **$** jellel kezdődik. Ha páros idézőjelek közé illesztett szöveg literálban szerepel egy változó neve ($ jellel kezdődő szó), akkor az értelmező a név helyett a változó értékét helyettesíti be a kimenetbe.

```
$vilag="World";       //$vilag áltozó értéke „World”
  echo "Hello $vilag!";//A Hello World! szöveget írja
ki
```

A hosszabb szövegliterálok ábrázolására alkalmas a Perlből átvett "heredoc" szövegmegadás, amit a PHP-ben a 4.0 verziótól kezdve használhatunk. Az ábrázolás lényege, hogy a szövegliterált két speciális sor között helyezhetjük el. A kezdősor három kisebb jelet (**<<<**) és egy szabadon választott jelölőszót tartalmaz, a befejező sorba pedig csak a kezdősorban megadott jelölőszót írjuk. Az alábbi példában a LATIN szó a szövegliterál jelölőszava.

```
echo <<<LATIN
   \nLorem ipsum \n dolor sit amet, consectetur
adipiscing elit. Nunc consequat orci non magna posuere
accumsan.
   Donec et nisi elit. Nulla facilisi.
   Sed tincidunt accumsan ligula, id varius leo
sollicitudin non.
   Duis semper nisl ut felis convallis at tincidunt
mauris interdum.
   Donec non adipiscing nisi. Cras et lacus leo.
Vivamus ac porta massa.
   LATIN;
```

Az értelmező a „heredoc” szöveget is értelmezi, tehát a vezérlő karakterek és a változó nevek itt is kicserélődnek.

A heredoc ábrázolás esetén fontos, hogy a szöveget nyitó jelölőszó után, a lezáró sorban pedig jelölőszó előtt semmi, még whitespace karakter sem lehet! Az alábbi megoldás tehát **rossz,** mert a záró sor jelölője előtt szóközök vannak.

echo <<<LATIN

Lorem ipsum

dolor sit amet, consectetur

adipiscing elit.

…LATIN;

4.2.2 A számok ábrázolása

A számok ábrázolásakor egész és valós típusú numerikus adatokat tárolhatunk literálban. A **valós szám literálokat** a számban elhelyezett pont (**.**) tizedes elválasztó különbözteti meg az **egész literáloktól**. Utóbbiak esetében az **oktális** és **hexadecimális** ábrázolásra is van lehetőség. Az oktális számokat **0 (011=>9) ,** a hexadecimális számokat **0x** karakterekkel (**0xff**=>255) kell kezdeni. A valós számokat legtöbbször egyszerűen tizedes elválasztóval jelezzük (3.14), de használhatjuk az exponenciális (normalizált) formát is (**314e-2**=> $314*10^{-2}$=>3.14).

Az alábbi példák megértéséhez elevenítsük fel, hogy az **echo** utasításnak több paramétere is lehet. Az **echo** sorban a kimentre írja őket.

```
echo "<h3>Szám literálok ábrázolása</h3>";
```

```
echo "Egész szám: ",314,"</br>";
```

```
echo "Oktális szám: ", 017,"</br>";
```

```
echo "Hexadecimális szám: ", 0xFF,"</br>";
```

```
echo "Exponenciális ábrázolás: ", 314e-
2,"</br>";
```

4.2.3 Logikai literálok

A logikai típusú adat mindössze kétféle, **igaz**, vagy **hamis** értéket vehet fel. A PHP-ben a **true** felel meg az igaz, és a **false** a hamis logikai literálnak.¨

4.3 KIFEJEZÉSEK

A számítógépes programban természetesen nem csak tároljuk az adatokat, de azok kezelésére, átalakítására utasítjuk a számítógépet. Az adatkezelésre, a számítógépes programnyelv utasításait, illetve úgynevezett kifejezéseket használhatunk.

> **A kifejezés egy adatkezelő művelet olyan leírása, amelyet az ember és a számítógép egyaránt képes értelmezni, elvégezni, és eredményét meghatározni.**

A kifejezések kétféle elemből, **tényezőkből** illetve **műveleti jelekből** épülnek fel. Ezeket **operandusoknak** illetve **operátoroknak** is nevezzük. Az operandus a feldolgozásra megadott adatot az operátor pedig a műveletet határozza meg.

Amikor az **értelmező egy kifejezést talál** a programkódban, akkor az abban a megadott **operandusokkal elvégzi az operátor által meghatározott műveletet,** majd az **eredményt** a kifejezés **helyére helyettesítve** dolgozik tovább.

A behelyettesítés természetesen nem jelenti a programkód átírását, a kifejezés eredményét csupán a végrehajtáshoz használja az értelmező.

Az alábbi sor hatására a 12 érték kerül a kimenetre:

```
echo 3*4;
```

Kifejezések mindenhol elhelyezhetők, ahol literálokat is használhatunk, és az értékekhez hasonlóan **típusuk** is van. Utóbbit a kifejezés **eredményének típusa határozza meg**. A fenti példa kifejezése numerikus, hiszen eredménye szám (12).

Az eredmény típusa természetesen a használt műveleti jel alapján dől el. A numerikus kifejezésekben aritmetikai (matematikai) operátorokat, a szöveges kifejezésekben sztringoperátorokat használunk.

4.3.1 Matematikai operátorok

Az elemi **matematikai műveletek** leírásához aritmetikai operátorokat használhatunk. A PHP öt ilyen műveleti jelet bocsát rendelkezésre:

Operátor	Művelet	Példa	Eredmény
+	Összeadás	6+3	9
-	Kivonás	6-3	3
*	Szorzás	6*3	18
/	Osztás	6/3	2
%	Maradékképzés	7%3	1

4.3.2 Sztring operátor

A sztringek vagy szövegek kezelésére csak a konkatenáció, **szövegösszefűzés** operátorát használhatjuk. Az értelmező a pont (.) operátor hatására egyetlen szöveggé illeszti össze a két operandust.

Operátor	Művelet	Példa	Eredmény
.	Összefűzés	"Hello " . "World"	"Hello World"

4.3.3 Összehasonlító operátorok

Az **összehasonlító operátorok** az úgynevezett **logikai kifejezések** leírásához használhatók. A logikai kifejezések a két operandus megadott operátor szerinti összehasonlításával megfogalmazott **állítások.** Eredményük ennek megfelelően **igaz/true, vagy hamis/false** érték lehet. Használatukra a különböző vezérlési szerkezetekben, elágazások, ciklusok feltételeinek leírásakor lesz szükség.

Operátor	Állítás	Igaz, ha	Példa	Eredmény
==	egyenlő	a két operandus **értéke megegyezik**	4 == 5	false
!=	nem egyenlő	a két operandus **értéke különbözik**	4 != 5	true
===	azonos	a két operandus **értéke és típusa is** megegyezik	„5”=== 5	false
!==	nem azonos	a két operandus **értéke vagy típusa** különböző	„5”!== 5	true
>	nagyobb, mint	a bal oldali operandus **értéke nagyobb** a jobb oldalinál	5 > 4	true
>=	nagyobb vagy egyenlő	a bal oldali tényező **értéke nagyobb, vagy egyenlő** a jobb oldalival	4 >= 4	true
<	kisebb, mint	a bal oldali operandus **értéke kisebb** a jobb oldalinál	5 < 4	false
<=	kisebb vagy egyenlő	a bal oldali operandus **értéke kisebb** a jobb oldalnál **vagy egyenlő** azzal	3 <= 4	true

4.3.4 Az értékadás operátora

A változók értékeinek beállításhoz is kifejezéseket használunk. Az ilyen kifejezéseket **értékadó vagy hozzárendelő kifejezésnek** nevezzük, operátoruk az **egyenlőségjel**. Ezek a kifejezések annyiban speciálisak, hogy **bal oldalukon mindig egy változó**, jobb oldalukon pedig egy értéket meghatározó **operandus** (literál, kifejezés, változó, függvényhívás) áll. A művelet eredményeként a bal oldalon található változó fölveszi a jobboldali operandus értékét.

Az alábbi példában a $világ változó értéke a „World” szövegre, a $pi változó értéke pedig a 3.14 double típusú értékre változik:

```
$vilag="World";
```

```
$pi=3.14;
```

Az „egyszerű” értékadás nem veszi figyelembe a változó addigi értékét, az esetleges korábbi értéket az új érték felülírja. A PHP-ben léteznek olyan **értékadó operátorok**, amelyek **a változó meglévő értékével és a jobboldali operandussal elvégzett művelet eredményét** helyezik a változóba.

Az alábbi táblázatban feltételezzük, hogy a kifejezésben szereplő változó már rendelkezik értékkel, amit a bal szélső oszlopban találunk.

Régi érték	Operátor	Példa	Egyenértékű kifejezés	Új érték
10	**+=**	$a **+=** 5	$a = $a + 5	15
10	**-=**	$a -= 5	$a = $a - 5	5
10	***=**	$a ***=** 5	$a = $a * 5	50
10	**/=**	$a **/=** 5	$a = $a / 5	2
10	**%=**	$a **%=** 5	$a = $a % 5	0
"Hello "	**.=**	$koszon**.=**"World"	$koszon = $koszon . "World"	"Hello World"

A fenti értékadó operátorok közül kettőnek speciális, úgynevezett egyoperandusú változatai is vannak. Az egyoperandusú műveleti jelek esetében, az operátor jobb oldalán nem áll operandus, helyesebben az értelmező úgy számol, mintha annak helyén 1 lenne.

A ++ operátor 1-et hozzáad, a -- 1-et elvesz a bal oldali operandus értékéből.

Régi érték	Operátor	Példa	Egyenértékű művelet	Új érték
10	++	$a++	$a=$a+1	11
10	--	$a--	$a=$a-1	9

4.3.5 Bináris operátorok

binary_operators.php

A bináris operátorok segítségével 2-es számrendszerben ábrázolt számokkal végezhetők műveltek.

Operátor	Művelet	Leírás
&	ÉS, azaz AND	Az eredmény a mindkét operandusban bekapcsolt bitekből álló szám.
|	VAGY, azaz OR	Az eredmény a legalább az egyik operandusban bekapcsolt bitekből álló szám.
^	kizáró vagy, azaz XOR	Az eredmény a csak az egyik vagy csak a másikoperandusban bekapcsolt bitekből álló szám.
~	NO	Egyoperandusú operátor. Az operandus bitjeit ellenkezőre fordítja.
<<	balra tolás	A jobb oldali operandus bitjeit a bal oldali operandusnak megfelelő helyi értékkel lépteti balra, a belépő bitek helyi értéke 0 lesz.
>>	jobbra tolás	A bal oldali operandus bitjeit a jobb oldali operandusnak megfelelő helyi értékkel lépteti jobbra, a belépő bitek helyi értéke 0 lesz.

4.3.6 Összetett kifejezések

Az összetett kifejezések az egyszerű kifejezésekhez hasonlóan műveleteket írnak le, de több, egymást követő kifejezésből állnak. Úgy is fogalmazhatnánk, hogy az **összetett kifejezések operandusai is kifejezések**. Feldolgozásuk alapértelmezés szerint balról jobbra haladva történik. Az értelmező veszi az első egyszerű kifejezést és eredményét operandusnak tekintve, folytatja az összetett kifejezés feldolgozását.

Ha például az összetett kifejezés 10+5+3-4, akkor előbb a 10+5 => 15, majd a 15+3 => 18, végül a 18-4 műveletet végzi el az értelmező. A kifejezés eredménye tehát 14 lesz.

Az alábbi utasítás hatására a 14 kerül a kimentre:

```
echo 10+5+3-4;
```

A balról jobbra haladó feldolgozási sorrendet a műveleti precedencia, illetve a zárójelezés változtathatja meg. Két egymás mellett lévő művelet közül mindig a magasabb precedenciájút hajtja végre először az interpreter. Azonos precedencia esetén a haladási sorrend dönt.

Az alábbi táblázatban fentről lefelé haladva soronként csökken a műveletek precedenciája, az azonos sorban lévő műveleteké pedig azonos.

```
++ --
!
* / %
+ - .
<< >>
< <= > >= <>
== != === !==
&
^
|
&&
||
? :
= += -= *= /= .= %= &= |= ^= <<= >>= =>
and
xor
or
```

Zárójelezés esetén a zárójelbe zárt rész együtt kerül kiértékelésre.

```
echo 10+5*3;
```

```
==> 25
```

```
echo (10+5)*3;
```

```
==> 45
```

4.3.7 Összetett logikai kifejezések

A logikai kifejezések is lehetnek összetettek. Az ilyen kifejezések **logikai operátorokkal elválasztott, egyszerű logikai kifejezésekből** épülnek fel, eredményük pedig szintén **igaz,** vagy **hamis** lehet. Az eredményt az egyszerű logikai kifejezések értéke és a használt logikai operátorok alapján határozza meg az interpreter.

Az alábbi összetett logikai kifejezés eredménye igaz, mert igaz hogy 6<7 ÉS az is igaz, hogy 7<8.

```
6<7 AND 7<8
```

A logikai operátorok az alábbiak lehetnek:

Operátor	Jelentés	Igaz ha
||	vagy	az egyik operandus igaz
or	vagy	az egyik operandus igaz
xor	kizáró vagy	csak az egyik operandus igaz
&&	és	mindkét operandus igaz
and	és	mindkét operandus igaz
!	tagadás	ha az egyetlen operandus hamis

Minden logikai operátor kétoperandusos, kivéve az utolsó, tagadás operátort. Ez ellenkezőjére fordítja az őt követő logikai operandus értékét.

4.4 VÁLTOZÓK

A literálok csak a programkód írásakor már ismert értékek tárolására alkalmasak, hiszen a programozónak bele kell írnia értéküket a kódba. A program futása során keletkező adatokat nem tárolhatjuk literálként. A **futás közben létrehozott adatokat változókban helyezzük el.**

A PHP-értelmező a futó program számára lefoglalja a számítógép memóriájának egy részét. A **változó** ennek a **memóriának egy névvel azonosított szakasza**, ahol valamilyen **érték**, adat tárolódik. A név a változó hivatkozási neve, az adat pedig az értéke. A programkódban a hivatkozási névvel hivatkozhatunk memóriában elhelyezett adatra. Az aktuális feladattól függően kimenetre írhatjuk, vagy megváltoztathatjuk a változóban tárolt értékét.

Ha egy literált többször is használunk a programkódban, akkor értékét célszerűbb változóban tárolni, és a változó nevével hivatkozni az értékre. Ez megkönnyíti a kód értelmezését, és egyszerűbbé teszi a programkód esetleges átírását. Az adatok programkódban történő tárolására tehát literálok helyett lehetőleg használjunk változókat!

4.4.1 Változók deklarálása

Mielőtt egy változóra hivatkoznánk, azaz használnánk az értékét:

1. le kell foglalni a helyét a memóriában,
2. hozzá kell rendelni a hivatkozási nevét (nevet kell adni a változónak),
3. meg kell adni a változó kezdőértékét.

Az 1-es és 2-es pontot együtt **deklarációnak** nevezzük, a 3. pontot **értékadásnak** hívjuk. Egy változónak csak a deklaráció után adhatunk értéket, az értékre pedig csak értékadás után hivatkozhatunk a változó nevével.

Változók nevei

A változók lehetséges neveit minden programozási nyelv szigorúan szabályozza. PHP-ben minden **változónév dollár jellel ($) kezdődik**, **betűkből**, **számjegyekből** és **aláhúzás** karakterekből (_) állhat. Megkötés, hogy a $ jelet követő első karakter nem lehet számjegy!

Betűnek számít az angol ábécé bármely betűje a-z-ig és A-Z-ig, valamint az ASCII karakterek a 127-es kódtól 255-ös kódig (0x7f-0xff).

A változónevek nem tartalmazhatnak szóközt, és bár az ASCII 127-255 karakterek között megtalálunk néhány nemzeti karaktert is, ezek használata könnyen vezethet félreértésekhez, ezért célszerű őket kerülni.

Az "ü" például megtalálható az ASCII-táblázatban (129) az "ő" azonban nem. Ha nem akarjuk örökké az ASCII-táblát nézegetni, akkor csak az angol ábécé betűit, aláhúzás karaktereket és számjegyeket használjunk!

A PHP – mint tudjuk – az utasítások neveiben nem különbözteti meg a kis- és nagybetűvel írt változatokat.

A változónevek kis- nagybetű érzékenyek!

Az $adat nevű változó nem azonos az $ADAT-tal!

Változók típusa

Az úgynevezett **szigorúan típusos** nyelvekben **explicit deklarációra** van szükség, ami azt jelenti, hogy a hivatkozás név mellett a változó típusát is meg kell adni a deklarációkor. **A típus meghatározza** a változó lehetséges **értékeit**, a vele végezhető **műveleteket** és az adat tárolására használt **memória méretét** is befolyásolja.

A PHP nem tartozik a szigorúan típusos nyelvek közé, de az adattípusoknak itt is van jelentőségük. A **egyszerű**, **összetett** és **speciális** típusú változókat használhatunk.

Az egyszerű típusok az alábbiak:

- **boolean:** logikai adat tárolására használt változók típusa.
- **integer**: egész számok típusa.
- **double/float**: tizedes törtek tárolására alkalmas változók típusa.
- **string**: szöveg típus.

Két összetett típus áll rendelkezésre:

- **array**: több, akár különböző típusú adat tárolását lehetővé tévő típus.
- **object**: adatok és műveletek tárolása alkalmas típus.

Speciális típusok:

- **resource:** erőforrás típus, a MySQL adatbázisokkal foglalkozó leckében dolgozunk ilyen változókkal.
- **NULL**: speciális típus, az értéket nem tartalmazó, ÜRES változó típusa.

Implicit deklaráció

A PHP úgynevezett **gyengén típusos nyelv**, ami azt jelenti, hogy egy változó deklarálásakor nem kell megadni annak típusát. A változót egyszerű értékadással, azaz **implicit deklarációval** hozhatjuk létre.

```
$a=10;
```

A fenti sor **lefoglalja a változó helyét** a memóriában (ha az még nem létezett), **hozzárendeli** a **$a hivatkozási** nevet és elhelyezi a változóban a 10-es **értéket**. A deklarációt azért nevezzük **implicitnek**, mert a **változó típusát a tárolt érték típusa határozza** meg, explicit megnevezésére nincs mód. Emiatt – bár nem jeleztük – az **$a** változó integer, azaz egész szám típusú lesz.

A típust tehát alapvetően a tárolt érték határozza meg, de arra is van lehetőség, hogy a változót a program különböző pontjain más-más típusú adatként kezeljük, sőt típusát meg is változtathatjuk a futás során.

Az alábbi programocska a 10 értéket írja a kimentre, holott a 4. sorban egy szöveg és egy egész szám típusú változót szoroztunk össze. A PHP a környezetnek, kifejezésben magadott műveletnek megfelelően, számként kezeli a szöveg változót.

```
1. <?php
2. $a="2";
3. $b=5;
4. echo $a*$b;     //A * művelet miatt az $a
   itt szám!
5. ?>
```

vars.php

Ez a fajta kötetlenség nagy szabadságot ad a programozónak, ugyanakkor számos hibához vezethet a figyelmetlen felhasználás során. Ezért lehetőleg kerüljük a típusok keverését, és csak azonos típusú változókat helyezzünk közös kifejezésbe! Ha erre nincs lehetőség, használjuk a típusátalakítás lehetőségeit!

Változó típusának meghatározása

Mivel a PHP-ben nem kell, és nem is lehet explicit deklarációval létrehozni a változókat, ráadásul a típus meg is változhat a program futása közben, nem lehetünk biztosak egy változó pillanatnyi típusában. A PHP számtalan függvénynyel támogatja a programozó munkáját (A függvényekről bővebben az alprogramokról szóló leckében olvashatunk). Közöttük megtalálhatók a típusok meghatározásával és konverziójával kapcsolatos függvények is.

A **gettype** függvény egy változót vár paraméterként és visszaadja annak típusát.

```
1 <?php
2 //gettype.php
3 $a=123;
4 echo gettype($a),"<br/>";      //integer
  $b=3.14;
  echo gettype($b),"<br/>";      //double
  $c="Hello World";
  echo gettype($c),"<br/>";      //string
  $d=$a===$b;
  echo gettype($d),"<br/>";      //boolean
  ?>
```

Forrás: gettype.php

Típusátalakítás függvénnyel

A PHP gyengén típusos mivolta nemcsak jelenti, hogy deklaráláskor nem kell megadnunk a típust, hanem azt is, hogy a változók típusa egyszerűen átalakítható. A két paramétert váró **settype** függvénnyel explicit módon adhatjuk meg egy változó típusát. A függvény első paramétere egy változó, a második pedig a céltípus szöveg formájában megadott neve. A függvény eredménye **true**, ha a sikeres az átalakítás, **false** ha a konverzió közben hiba keletkezett.

```
<?php
//settype.php
$b=3.14;
echo "\$b típusa: ",gettype($b),"<br/>";  //double
settype($b,"integer");
echo "\$b típusa: ",gettype($b),"<br/>";  //integer
?>
```

Forrás: settype.php

A változó típusának megváltoztatásakor sérülhet az abban tárolt érték. Ha egy double változót integer típusúra állítunk a törtrész elvész.

Típus átalakítása casting-gal

A **`settype()`** függvény ténylegesen megváltoztatja a változó típusát.

Ha változót csak egy adott ponton akarjuk más típusú adatként használni, akkor típus ideiglenes átalakítása, az úgynevezett **type casting** technikával végezhető el. Ilyenkor – kerek zárójelek között – a változó neve el kell írni a kívánt típus nevét!

```
<?php
 //casting.php
 $b=3.14;
 echo "\$b=$b; típusa: ",gettype($b),"<br/>";
 echo "(integer) \$b típusa:
 ",gettype((integer)$b),"<br/>";
 echo "\$b típusa: ",gettype($b),"<br/>";
?>
```

Forrás: casting.php

4.4.2 Változók törlése

Előfordul, hogy nagyobb PHP-programban törölni szeretnénk egy korábban létrehozott változót. Erre használhatjuk PHP **`unset()`** függvényét, ami teljes egészében eltűnteti a memóriából a paraméterként megadott változót. Ha ezután hivatkozunk a változóra, a hibát jelző megjegyzés jelenik meg a kimeneten, de a futás nem szakad meg.

```
1 $a=300;
2 unset($a);
3 echo $a; //Undefined variable: t
```

Ha csak az értékét akarjuk törölni, adjunk NULL értéket a változónak!

4.5 KONSTANSOK

A program elkészítése során szükségünk lehet olyan adatokra, amelyekre alfanumerikus nevekkel akarunk hivatkozni, de értéküket soha nem akarjuk változtatni. Az ilyen adatokat konstansoknak, állandóknak nevezzük.

Létrehozásuk a **`define()`** függvénnyel történik, amelynek első paramétere a konstans neve, a második az értéke.

Az állandók neve a változókkal szemben nem kezdődik $ karakterrel, ellenben szokás (nem kötelező), hogy ezeket a neveket nagybetűkkel adjuk meg.

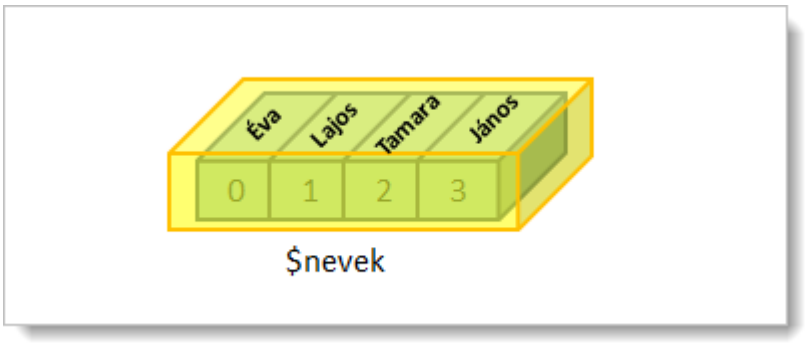
```
1 <?php
    //constant.php
    define("PI",3.14);       //A konstansok nevei
konvencionálisan nagybetűsök
    $r=10;
    echo "Terület: ",$r*$r*PI;   //A konstans értékére
is nevével hivatkozunk
  ?>
```

Forrás: constant.php

4.6 TÖMBÖK

A tömb a PHP meglehetősen flexibilis, összetett adattípusa, amely több érték egyidejű tárolására is alkalmas változók létrehozását teszi lehetővé.

Ha egy egyszerű változót úgy lehet elképzelni, mint egy névvel fölcímkézett fiókot, amely adatot tárol, akkor a tömb egy névvel ellátott, többrekeszes fiók, aminek rekeszeiben különböző értékek helyezkedhetnek el. A tömb egyes rekeszeit **tömbelemeknek** nevezzük. Egy tömbváltozóban annyi érték tárolható, ahány eleme van a tömbnek, illetve egy tömbnek annyi eleme van, amennyi értéket elhelyezünk a tömbben



19. ábra Tömbváltozó

A változókra egyszerűen a nevükkel hivatkozhatunk. A tömbök esetében azonban a név mellett, a hivatkozott tömbelem sorszámát, indexét is meg kell adnunk.

Ha például van egy ***$nevek*** *nevű tömbváltozónk, aminek elemei az "Éva","Lajos", "Tamara","János" szövegeket tárolják, akkor az "Éva" nevet tároló elemre a következőképpen hivatkozhatunk:*

```
$nevek[0];
```

4.6.1 Tömbök deklarálása

Egyes programozási **nyelvekben** a **tömb típus homogén és statikus** adattípus. A homogenitás azt jelenti, hogy minden tömbelemnek azonos típusúnak kell lennie, a statikus jelző pedig arra utal, hogy deklarálás után nem változhat a tömb elemeinek száma.

A **PHP tömbjei ezzel szemben inhomogén, dinamikus tömbök**, azaz elemeik típusa egymástól független, és a tömb elemszáma is szabadon változtatható.

Egy tömb deklarálása, és egyben a kezdőértékek megadása az **`array()`** függvénnyel történik, amelynek tetszőleges számú paramétere lehet. Az új tömbben annyi elem lesz, ahány paramétere van az **`array()`** függvények, és minden elem egy-egy paraméter értékét tárolja majd.

Az alábbi deklaráció például létrehozza a négyelemű $nevek tömböt, amelynek elemeiben az "Éva", "Lajos", "Tamara", "János" szövegek tárolódnak.

```
$nevek=array( "Éva","Lajos", "Tamara","János");
```

A **tömbelemek azonosítására** sorszámaikat, azaz **indexüket** használjuk.

A PHP-ben tömbök legelső elemének indexe **nem 1, hanem mindig** 0**! Az ilyen tömböket 0, vagy zéró bázisú tömböknek nevezzük.**

A paraméterek nélküli **`array()`** üres tömböt hoz létre, amelynek még nincsenek elemei.

```
$ures=array();
```

A tömb létrehozása után, akár az érték használatához, akár megváltoztatásához, a változó nevének és az elem szögletes zárójelekbe zárt indexének, megadásával, *$változó* **[***index***]** formában hivatkozhatunk egy tömbelemre.

```
$nevek=array( "Éva","Lajos", "Tamara","János");
$nevek[3]="John";     //3-as elem megváltoztatása
echo $nevek[3];       //3-as elem értékének kiírása
```

Forrás: array.php

4.6.2 Új elem hozzáfűzése a tömbhöz:

Mivel a PHP tömbjei dinamikusak, bármikor új elemet adhatunk a tömbhöz. Ehhez egy **index nélküli tömbelemnek** kell **értéket** adni! Az új elem ilyenkor mindig a tömb végére kerül.

```
$nevek[]="Károly"     //Két új elem kerül a tömb
végére;
$nevek[]="Vilma";
```

A fenti utasítás hatására a **`$nevek`** tömb két elemmel bővült: "Éva","Lajos", "Tamara","John","Károly","Vilma"

Az index nélküli hivatkozás üres tömb létrehozására, tehát deklarálására is alkalmas, de akár azonnal elemet is illeszthetünk az új tömbbe. Üres tömb létrehozásakor a speciális NULL értéket, egyébként bármilyen értéket rendelünk az index nélküli elemhez.

```
//az üres, $nincselem tömb létrehozása
$nincselem[]=NULL;
//egy elemű, de dinamikusan tovább bővíthető tömb
$evszamok[]=1962;
```

4.6.3 Többdimenziós tömbök

A tömbök lehetnek egy- vagy többdimenziósak. Az egydimenziós tömbök, vagy más néven **vektorok** egyetlen elemsorozattal rendelkeznek. (Fiók egymás mellett lévő rekeszekkel.)

A kétdimenziós tömböket úgy tudjuk elképzelni, mintha egy fiókban több sorban helyezkednének el a rekeszek. A kétdimenziós tömb elemei sorokat és oszlopokat alkotnak, ezért az ilyen tömböt **mátrixnak** nevezzük.

A mátrix létrehozásához egy apró trükköt használunk: olyan vektort készítünk, aminek minden eleme tömb.

```
1 $matrix=array(
2   array("Éva","Lajos",
   "Tamara","John","Károly","Vilma"),
3   array(1962,1970,1981,1978,2001,1989)
4   );
```

A többdimenziós tömbök logikailag összetartozó, de eltérő típusú adatok tárolására és kezelésére alkalmasak.

A fenti példa kétsoros, hatoszlopos mátrixban tárolja emberek neveit és születési évüket.

A **mátrixok elemeire** a **sor- és az oszlopindex** együttes megadásával hivatkozhatunk, az indexeket **$***tömb* **[***sor***] [***oszlop***]** formában egymás mellett elhelyezett szögletes zárójelpárokba írjuk:

Ha meg akarjuk jeleníteni az első nevet és a hozzá tartozó születési évet, akkor mindkét esetben sor és oszlop indexet is meg kell adnunk.

Az alábbi kódrészlet az Éva nevet és az 1962-es évet írja ki.

```
  <?php
     //matrix.php
     //Kétdimenziós tömbb
     $matrix=array(
        array("Éva","Lajos",
"Tamara","John","Károly","Vilma"),
        array(1962,1970,1981,1978,201,1989)
        );
     echo "Név: ",$matrix[0][0],"<br/>";
     echo "Született: ",$matrix[1][0],"<br/>";
     echo  $matrix[0][0]," ",$matrix[1][0]," évben
született.<br/>";
  ?>
```

Forrás: matrix.php

4.6.4 Asszociatív tömbök

Az eddig tárgyalt tömböket **numerikus tömböknek** nevezzük, mert elemiket **számokkal indexelhetjük**. A PHP másik tömbtípusában, az úgynevezett **asszociatív tömbökben** az **elemeket** nem sorszámok, hanem **alfanumerikus azonosítók**, úgynevezett **kulcsok** indexelik.

Az asszociatív tömböket másképpen kell deklarálni, de értékeikre ugyanúgy hivatkozunk. Bár a deklaráláshoz itt is az **`array()`** függvényt használjuk, az egyes elemeket *kulcs***=>***érték* formában adjuk meg.

Az alábbi példa létrehozza az $ember nevű, négyelemű asszociatív tömböt. Az elemek kulcsai ***„nev”, „szulev”, „szulhely”, „testsuly”.***

```
$ember=array(
   "nev"=>"Molnár Éva",
   "szulev"=>1962,
   "szulhely"=>"Eger",
   "testsuly"=>58
);
```

Ez az asszociatív tömb tehát négyelemű, az elemekre a kulcsukkal hivatkozhatunk. Az alábbi példa kiírja „nev" tömbelemet, megváltoztatja, majd kiírja a"testsuly" elem értékét is.

```
$ember=array(
   "nev"=>"Molnár Éva",
   "szulev"=>1962,
   "szulhely"=>"Eger",
   "testsuly"=>58
);
echo $ember["nev"],"<br/>";
$ember["testsuly"]-=2;
echo $ember["testsuly"],"<br/>";
```

Az asszociatív tömbök alkalmasak a világ valamely dolga, egyede, objektuma tulajdonságaink egy változóban való tárolására. Más nyelvekben a rekord típus felel meg leginkább ennek a funkciónak.

Az asszociatív tömböket leginkább numerikus tömbök elmeiként alkalmazzuk. Ha például több ember adatait egyetlen **`$csoport`** nevű változóban akarjuk tárolni, akkor a **`$csoport`** változót, mint tömböt hozzuk létre, de minden elemét asszociatív tömbként adjuk meg.

Forrás: assoc.php

4.6.5 Hivatkozás nemlétező tömbelemre

A PHP úgynevezett kezelhető hibát generál, és alapbeállításban hibaüzenet jelenít meg a kimeneten, ha nem létező tömbelemre hivatkozunk.

Forrás: array_error_notice.php

Bár az ilyen hibák nem végzetesek, a tömbelem-hivatkozásokat mindig ellenőrzés alatt kell tartani. Ez ehhez kapcsolódó függvényekről a 7. leckében olvashat.

4.7 ÖSSZEFOGLALÁS, KÉRDÉSEK

4.7.1 Összefoglalás

Mai leckénkben nagyot léptünk előre, hiszen áttekintettük a PHP fontosabb adattároló struktúráit. Elsőként a programszövegben közvetlenül tárolt értékekről beszéltünk. Megállapítottuk, hogy a literálok legfontosabb jellemzője az érték, azonban az értékek típussal is rendelkeznek, amelyet az ábrázolás módja határoz meg. A szövegliterálokat páros vagy páratlan idézőjelek közé írjuk, az egész számok csak számjegyekből, esetleg előjelből állnak, a valós számok pedig tizedes elválasztót tartalmaznak.

A literálok után a program futása közben keletkező adatok tárolására használható változókról esett szó. Megállapítottuk, hogy a PHP gyengén típusos nyelv, mert a változók deklarációjakor nem kell, és nem is lehet explicit módon megadni a típust. A változók típusát értékük, és felhasználásuk környezete határozza meg.

A PHP string, integer, double, és boolean típusai mellett megismertük a programozási nyelv egyik legrugalmasabban használható adattípusát, a tömböt. Megtanultunk, hogy a PHP tömbjei inhomogén, dinamikus adatszerkezetek, azaz a tömbváltozók elemszáma változhat, elemeik pedig eltérő típusúak lehetnek.

A tömbök kapcsán beszéltünk az egy- és többdimenziós, valamit a numerikus és asszociatív tömbökről. Megállapítottuk, hogy az asszociatív tömbök egyedek tulajdonságainak tárolására és kezelésére alkalmas adatszerkezetek, amelyek elemeit numerikus indexek helyett szöveges kulcsokkal azonosítjuk.

A lecke anyagának jelentős hányadát tette ki műveletek leírására alkalmas kifejezések ismertetése. A kifejezések kapcsán megismertük a PHP különböző műveletekre használható, két-, illetve egyoperandusos operátorait, és a kifejezések kiértékelésének módját.

4.7.2 Önellenőrző kérdések

1. Mi kerül a kimenetre az alábbi kódrészlet végrehajtásakor?
   ```
   $nev=”Tóth Áron”;
   echo ’Név: $nev\r\n’;
   ```
   - Pontosan a ’Név: $nev\r\n’ szöveg, a literált ugyanis páratlan idézőjelek között adtuk meg, így az interpreter értelmezés nélkül a kimenetre írja a szöveget.
2. Mi a hiba az alábbi kódrészletben?
   ```
   echo <<<TXT
   Ez
   egy
   többsoros
   szövegTXT;
   ```
   - A kódrészlet az úgynevezett heredoc szövegliterált használná, azonban ebben szabály, hogy a literált lezáró jelölőszó sorában semmi sem állhat a jelölő előtt. A TXT;-nek tehát külön sorba kellene kerülnie.
3. Mit ír a kimenetre az alábbi kifejezés?
   ```
   $x=11.11;
   echo (interger) $x;
   ```
   - A 11 számot. Az $x típusa ugyan double, de a kiíráskor integerre „type casingoltuk”. Ilyenkor a törtrész természetesen elvész.
4. Mit ír ki az alábbi sor?
   ```
   echo "Eredmény: ".(12+3*4>5%2*3);
   ```
   - Eredmény: 1
     Azért, mert a zárójeles kifejezésben a logikai kifejezés bal oldala 24 (12+3*4), a jobb oldal pedig 3 (5%2*3). A logikai kifejezés igaz, amit a PHP 1-ként ábrázol.
     Az "Eredmény: ".1 kifejezésben az 1 szövegként viselkedik, amit a . operátor hozzáfűz az „Eredmény: „ szöveghez.
5. Mit jelent az, hogy a PHP tömbjei dinamikusak és inhomogének?

- Azt, hogy a tömbtípusú változók elemszáma változhat, és az elemek eltérő típusúak lehetnek.

Hozzon létre $konyv nevű asszociatív tömböt, amelynek elemei egy könyv címét, szerzőit, kiadóját és a megjelenés évét tárolják!

Készítsen konyvtar.php nevű programot, amely 5 könyv adatait tárolja egy tömbben, és a program $peldany nevű változójában megadott sorszámnak megfelelő könyv adatait kiírja a kimenetre! Az adatok címkézve jelenjenek meg! (Pl: Cím: Virágot Algernonnak!) Ha a $peldany értékét átírjuk, és a programot újraindítjuk, másik könyv adatai jelenjenek meg.

5. LECKE: A PHP VEZÉRLÉSI SZERKEZETEI

5.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

Miután megismertük a PHP adatszerkezeteit, itt az ideje, hogy végre valódi programokat készítsünk. Az elmúlt leckében arról beszéltünk, hogy a program különböző állapotai adatokkal írhatók le. Az egyes állapotok adatait utasításokkal és kifejezésekkel manipulálva előállíthatók a következő állapotot leíró adatok. Ha a programok készítése csak ennyiből állna, akkor valójában készen állnánk az alkalmazások készítésére, hiszen adatokat tudunk tárolni, és megismertük az adatok manipulálására használható kifejezéseket is. Mi hiányzik még?

Természetesen nagyon sok minden, de ahhoz, hogy működő webes programokat tudjunk készíteni, alapvetően két dologra van szükségünk.

Az első problémát az adatbevitel jelenti. Bár megtanultuk a feldolgozás alatt álló adatok tárolására használható változók kezelését, arra nem gondoltunk, hogy a legtöbb adat közvetve, vagy közvetlenül a felhasználótól származik. A programozási nyelvek többségében viszonylag egyszerűen tudunk adatokat átvenni a felhasználótól, a webalkalmazások esetén azonban szembesülni fogunk néhány nehézséggel!

A Böhm–Jacopini-tétel kimondja, hogy minden program felépíthető három alkotóelemből. Ezek a szekvencia, a szelekció, és az iteráció. Szekvenciák készítésére természetesen képesek vagyunk, hiszen ez gyakorlatilag utasítások sorfolytonos leírását jelenti. Ami hiányzik, az a szelekciók és iterációk megvalósításra alkalmas nyelvi elemek, az úgynevezett vezérlési szerkezetek ismerete.

Ez a második problémánk. Ha meg tudjuk valósítani, akár alapszinten is a felhasználói adatbevitelt, és megismerjük az elágazások, ciklusok szervezésének lehetőségeit, hatalmas lépünk előre a webprogramozás ismereteinek elsajátításában.

Mai leckénkben a második probléma megoldására vállalkozunk. A megtanuljuk a kettő és többágú elágazások, a léptető, az elől, és a hátul tesztelő ciklusok megvalósításának technikáját, és megismerjük a tömbök kezelésének egyszerű és kényelmes módját.

A lecke végére elegendő ismerettel rendelkezünk ahhoz, hogy megbirkózzunk a felhasználói adatbevitel kihívásaival, amire majd a követező leckében térünk rá.

5.2 A PHP VEZÉRLÉSI SZERKEZETEI

Az adatok tárolására használható nyelvi elemek ismerete elengedhetetlenül szükséges a programozáshoz, de programozási nyelv vezérlési szerkezetei, az elágazások és szelekciók megvalósítására alkalmas nyelvi eszközei nélkül semmire sem jutunk.

5.3 ELÁGAZÁSOK

Míg az utasításszekvenciák végrehajtása egyetlen utasításág mentén, azaz lineárisan halad, addig a szelekciók, vagy más néven elágazások esetén a kód több utasításágra bomlik. Minden ág további utasításokat tartalmazhat, azonban a szelekció után az ágak újra találkoznak, és a program végrehajtása ismét közös szekvenciában folytatódik.

Az elágazásban az interpreternek döntenie kell, hogy melyik ág utasításait hajtja végre, azaz melyik utasítások fogják továbbvezérelni a programot. Ezért az értelmező az elágazási pontban (minden szelekciós vezérlési szerkezet esetén) megvizsgál egy tesztadatot, és annak értékei alapján dönti el, hogy melyik utasításágon halad tovább. Miután az ágak befejeződtek, a vezérlés az elágazást követő első utasításra kerül, és a végrehajtás újra egyetlen ágon folytatódik.

Az elágazási pontokból induló ágak alapján kétágú vagy többágú elágazásokról beszélhetünk.

5.3.1 Kétágú elágazások

A kétágú elágazások szervezésére az **`if`** utasítást használhatjuk, ami az alábbi formában teszi lehetővé az ágak leírását:

```
if(tesztkifejezés){
   igaz_ág
} else {
   hamis_ág
}
```

Az **`if`** kulcsszót kerek zárójelek között, egy tesztkifejezésként használt **logikai kifejezés** követi. Ezután két, kapcsos zárójelek között elhelyezett **utasítás blokk** következik, amelyek között az **`else`** kulcsszó áll.

Az első blokkot **igaz**/ha, a másodikat **hamis**/különben **ágnak** nevezzük, mert az értelmező az első ágat hajtja végre, ha tesztkifejezés értéke **igaz**, ellenkező esetben azonban a második ág kerül végrehajtásra. A különben ág után a végrehajtás újra egy szekvenciában folytatódik.

Forrás: if_alap.php

if_alap.php

```php
<?php
    //if_alap.php
    header('Content-Type: text/html; charset=utf-8');
    $a=10;
    $b=5;
    if($a>$b){
        echo "$a a nagyobb<br/>";
    } else {
        echo "$b a nagyobb<br/>";
    }
    echo "$a és $b összege=",$a+$b;
?>
```

20. ábra if…else

A fenti példa két változóban tárolt szám közül a nagyobbat írja ki.

Az olvasó nyilván észreveszi, hogy a program szépséghibája, hogy nem kezel minden lehetőséget. Nincs felkészítve arra az esetre, amikor a kész szám azonos értéket tartalmaz.

Az **`if`** utasítás egyik speciális formája, amikor a **hamis ág hiányzik**. Ez is kétágú elágazás, csupán arról van szó, hogy a hamis ág üres. Ha a feltétel teljesül, akkor az interpreter végrehajtja az igaz ágat, majd tovább folytatja a programot, ha hamis, akkor az igaz ág utasításait átlépve azonnal az elágazás után folytatódik a program.

```
if(tesztkifejezés){
   igaz_ág
}
```

Előfordul, hogy az igaz ág egyetlen utasításból áll. Ilyenkor az utasítást nem kötelező kapcsos zárójelek közé tenni. Az alábbi forma is megfelelő.

```
if(tesztkifejezés) igaz_utasítás;
```

Az **`if`** segítségével tehát kétágú elágazások szervezhetők, azonban akár az igaz, akár a hamis ágban is további elágazásokat helyezhetünk el. Így a vezérlési szerkezet alkalmazásával a többágú elágazások készítésére is van lehetőség.

Forrás: if_beagyazott.php

if_beagyazott.php

```php
<?php
    //if_beagyazott.php
    header('Content-Type: text/html; charset=utf-8');
    $a=10;
    $b=5;
    if($a==$b){
        echo "$a és $b egyenlő.<br/>";
    } else {
        if($a>$b){
            echo "$a a nagyobb<br/>";
        } else {
            echo "$b a nagyobb<br/>";
        }
    }
    echo "$a és $b összege=",$a+$b;
?>
```

21. ábra Egymásba ágyazott elágazások

Az itt látható programocska kiküszöböli az előző példa hibáját. Akkor is helyesen működne, ha az **$a** és a **$b** változó azonos értéket tartalmazna.

A sokszorosan egymásba ágyazott elágazások nehezen olvashatóvá teszik a programkódot. A feladat egyszerűbb megvalósítására használható az **if**…**elseif**…**else** szerkezet, ahol az igaz ágat követő **elseif** kulcsszóval új elágazást helyezhetünk el a programkódban.

```
if(tesztkifejezés){
   igaz_ág
} elseif (tesztkifejezés) {
   igaz_ág_2
} else {
   hamis_ág
}
```

Forrás: if_elseif.php

if_elseif.php

```php
<?php
    //if_elseif.php
    header('Content-Type: text/html; charset=utf-8');
    $a=10;
    $b=5;
    if($a==$b){
        echo "$a és $b egyenlő.<br/>";
    } elseif  ($a>$b) {
        echo "$a a nagyobb<br/>";
    } else {
        echo "$b a nagyobb<br/>";
    }
    echo "$a és $b összege=",$a+$b;
?>
```

22. ábra if…elseif…else

Előfordul, hogy egy feltételtől függően két érték közül vagy az egyiket, vagy a másikat kell használnunk. A PHP-ben használhatjuk az if…else… szerkezet egy egyszerű változatával könnyen kezelhetjük az ilyen feladatokat:

(*logikai_kifejezés***) ? *igaz_érték* : *hamis_érték*;**

```
$a=1;
$b=2;
echo ($a>$b) ? $a : $b;
//2;
```

5.3.2 Többágú elágazások

Az elágazások szervezésére alkalmas másik vezérlési szerkezet az **`switch`**, amelyet az **`if`**-fel szemben alapvetően többágú elágazások készítésére szántak.

```
switch (tesztkifejezés){
   case tesztérték1:
      ág_utasításai
      break;
   case tesztérték2:
      ág_utasításai
      break;
   default:
      különben_utasítások
}
```

A **switch** utáni zárójelek között tetszőleges **tesztkifejezés** állhat, de ennek értéke fogja meghatározni, hogy az interpreter melyik később leírt ágat fogja végrehajtani. Az ágak a tesztkifejezés után egyetlen kapcsos zárójelpár között helyezkednek el. Az egyes ágak **case** kulcsszóval kezdődnek, amelyeket egy **tesztérték** majd kettőspont követ. Minden ág tetszőleges utasításokat tartalmazhat, de az utolsó szinte mindig a **break!**

Amikor az interpreter végrehajtja a **switch** utasítást, kiszámolja a **tesztkifejezés** értékét, majd megkeresi az első olyan ágat, ahol a **case** kulcsszó utáni tesztérték megegyezik a tesztkifejezés kiszámolt értékével. Ennek az ágnak az utasításait fogja végrehajtani. A **break**-re azért van szükség, mert különben a kiválasztottat követő ág utasításait is végrehajtaná az értelmező.

A **break** hatására a vezérlés azonnal az elágazás (**switch** szerkezet) utáni első utasításra kerül.

A **default** különleges ág. Akkor kerül ide a vezérlés, ha az értelmező egyetlen, a tesztkifejezés értékének megfelelő ágat sem talál. Ha nincs **default** ág, akkor ilyen esetben az elágazás utáni részre kerül a vezérlés.

Forrás: switch_syntax.php

switch_syntax.php

```php
<?php
    header('Content-Type: text/html; charset=utf-8');
    $nap="kedd";
    switch ($nap){
        case "hétfő":
            echo "Munkanap<br/>";
            break;
        case "kedd":
            echo "Munkanap<br/>";
            break;
        case "szerda":
            echo "Munkanap<br/>";
            break;
        case "csütörtök":
            echo "Munkanap<br/>";
            break;
        case "péntek":
            echo "Munkanap<br/>";
            break;
        case "szombat":
            echo "Szabadnap<br/>";
            break;
        case "vasárnap":
            echo "Munkaszüneti nap<br/>";
            break;
        default:
            echo "Nincs ilyen nap<br/>";
    }
?>
```

23. ábra A switch használata

A fenti példaprogram a $**nap** változó értékét tekinti tesztkifejezésnek. Az egyes ágak a hét napjait tartalmazzák tesztértékként. A program kiírja, hogy $**nap** aktuális értéke munkanap, szabadnap, vagy munkaszüneti nap. Ha a **$nap** egyetlen ágnak sem felel meg, a **default** ág utasítása kerül végrehajtásra.

A **break** utasítás biztosítja, hogy egy ág végrehajtása után a vezérlés ne fusson át a következő ágba.

Akkor nem használjuk a **break**-et, ha a szándékunk az, hogy két különböző tesztérték esetén is ugyanazok az utasítások kerüljenek végrehajtásra. A fenti példában a hét első öt napja esetén azonos szövegnek kellene megjelennie. A **break** utasítások elhagyásával több különböző tesztértékre is azonos módon tudunk reagálni.

Forrás: switch_syntax_nobreak.php

switch_syntax_nob...

```php
<?php
    header('Content-Type: text/html; charset=utf-8');
    $nap="szombat";
    switch ($nap){
        case "hétfő":
        case "kedd":
        case "szerda":
        case "csütörtök":
        case "péntek":
            echo "Munkanap<br/>";
            break;
        case "szombat":
            echo "Szabadnap<br/>";
            break;
        case "vasárnap":
            echo "Munkaszüneti nap<br/>";
            break;
        default:
            echo "Nincs ilyen nap<br/>";
    }
?>
```

24. ábra Több ág együttes kezelése

5.4 CIKLUSOK

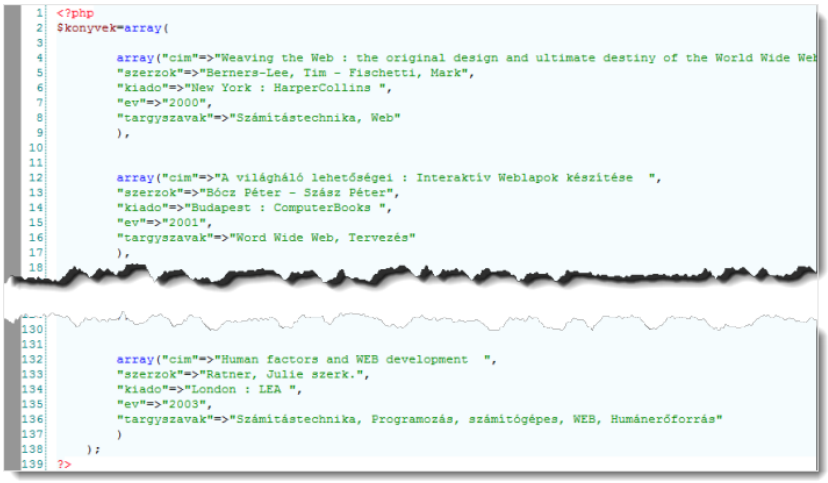
A ciklusok szintén a vezérlési szerkezetek közé tartoznak. Utasítások vagy utasításcsoportok többszöri végrehajtását, ismétlését teszik lehetővé, ezzel kódrészletek újrafelhasználhatóságát biztosítják. Minden ciklusszervező vezérlési szerkezetre igaz, hogy **ciklusfej**, **ciklusmag**, **ciklusvég** és **teszt** részekre tagolódik.

A ciklusfej és ciklusvég közé zárt ciklusmag tartalmazza azokat az utasításokat, amelyeket a ciklus többször is képes végrehajtani. Az interpreter teszt alapján dönteni el, hogy folytatódjon-e az ismétlés.

Az ismétlések között **elöltesztelő**, **hátultesztelő**, és **léptető ciklusokat** különböztethetünk meg. Az elöltesztelő és a léptető ciklusokban a **ciklusmag előtt történik a tesztelés**, a hátultesztelő ciklusok esetén a **ciklus vége után**. Ebből következik, hogy a **hátultesztelő** ciklusok ciklusmagja legalább **egy alkalommal biztosan** végrehajtódik, az **elöltesztelő ciklus** esetében azonban **lehetséges**, **hogy** a ciklusmagot **egyszer sem** hajtja végre az interpreter.

A következő szakaszok példaprogramjainak megértéséhez szükségünk van a példák PHP-szkriptjeivel azonos mappában található a **konyvtar.php** nevű fájlra, ami a **$konyvek** numerikus tömböt tartalmazza. A tömb minden eleme egy asszociatív tömb, amely egy-egy könyv címét, szerzőit, kiadási évét stb. tárolja, tehát hasonlít ahhoz, amit az előző lecke záró feladataként kellett létrehozni. Hogy a példáink áttekinthetők legyenek, a tömböt külön állományban tároltuk, és minden esetben a **require** utasítással csatoltuk. A tömböt tároló szkript felépítését a következő ábrán láthatjuk.

Forrás: konyvtar.php



```php
<?php
$konyvek=array(

        array("cim"=>"Weaving the Web : the original design and ultimate destiny of the World Wide Web
        "szerzok"=>"Berners-Lee, Tim - Fischetti, Mark",
        "kiado"=>"New York : HarperCollins ",
        "ev"=>"2000",
        "targyszavak"=>"Számítástechnika, Web"
        ),


        array("cim"=>"A világháló lehetőségei : Interaktív Weblapok készítése  ",
        "szerzok"=>"Bócz Péter - Szász Péter",
        "kiado"=>"Budapest : ComputerBooks ",
        "ev"=>"2001",
        "targyszavak"=>"Word Wide Web, Tervezés"
        ),

        array("cim"=>"Human factors and WEB development  ",
        "szerzok"=>"Ratner, Julie szerk.",
        "kiado"=>"London : LEA ",
        "ev"=>"2003",
        "targyszavak"=>"Számítástechnika, Programozás, számítógépes, WEB, Humánerőforrás"
        )
    );
?>
```

25. ábra konyvtar.php

5.4.1 Elöltesztelő ciklus

Az elöltesztelő ciklus szervezésére az alábbi struktúra használható:

```
while(tesztkifejezés){
   ciklusmag
}
```

A ciklus elejét a **while** kulcsszó jelzi, amit egy zárójelek közé írt **logikai kifejezés** követ. A logikai kifejezés értéke – mint tudjuk – igaz vagy hamis lehet. Ez szolgálja majd a teszt funkciót.

A **ciklusmag** ezután következik **kapcsos zárójelek között**. A ciklus végét a bezáró kapcsos zárójel mutatja.

Amikor az interpreter a ciklusfejhez ér, mindig megvizsgálja a tesztkifejezés pillanatnyi értékét. Ha ez **igaz**, akkor végrehajtja a ciklusmag utasításait. Ha **hamis**, akkor a ciklusvéget követő első utasításra ugrik. A ciklusmag végrehajtása után a vezérlés mindig visszakerül a ciklusfejhez, ahol újra megtörténik a tesztelés. Ennek eredménye dönt az ismétlés folytatásáról.

Forrás: while_syntax.php

while.sysntax.php

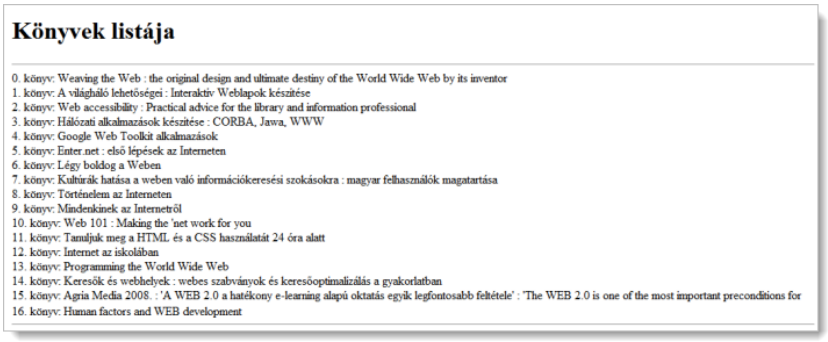
```
1  <?php
2      //while.sysntax.php
3      header('Content-Type: text/html; charset=utf-8');
4      require "konyvtar.php";
5      $konyvszam=count($konyvek);
6      $i=0;
7      echo "<h1>Könyvek listája</h1><hr/>";
8      while ($i<$konyvszam){
9          echo "$i. könyv: ",$konyvek[$i]["cim"],"<br/>";
10         $i++;
11     }
12     echo "<hr/>";
13 ?>
```

26. ábra while ciklus

A fenti példában használjuk a korábban említett **konyvtar.php**-ben található **$konyvek** tömböt.

A 5. sorban a **count()** függvénnyel meghatározzuk a tömb elemeinek számát, majd létrehozunk egy **$i** nevű o kezdőértékű változót. A ciklusfej előtt a "**<hr/>**" jelölő kiírásával biztosítjuk egy vízszintes vonal, megjelenését.

A ciklusfejben lévő teszt ellenőrzi, hogy **$i** kisebb-e mint a **$konyvszam** változó értéke. Ha igen, akkor végrehajtódik a ciklusmag, amelyben kiírjuk az **$i**. könyv címét, majd **$i** értékéhez hozzáadunk 1-et. Amikor elérjük a ciklusvéget, a vezérlés újra a fejre kerül, ahol már **$i** új értékét hasonlítjuk össze a tömb elemszámával. Így az ismétlés mindaddig tart, amíg az összes elemet ki nem írjuk.



27. ábra A példaprogram kimente a böngészőben

Ebben az esetben azért szerencsés az elöltesztelő ciklus használata, mert előfordulhat, hogy a tömb üres. Ebben az esetben a ciklusmag végrehajtását egyszer sem lenne szabad megkísérelni, különben nemlétező tömbelemre hivatkoznánk.

5.4.2 Hátultesztelő ciklus

A hátultesztelő ciklus megvalósítására a **`do..while`** szerkezetet használjuk.

```
do{
   ciklusmag
} while (tesztkifejezés)
```

A ciklus működésekor egyszer mindenképpen lefut a ciklusmag. A tesztkifejezés csak ciklus végén kerül kiértékelésre. Ha az értéke igaz, akkor a ciklus megismétlődik, különben a program következő utasítására kerül a vezérlés.

A n faktoriális az a szám, amely a pozitív egész n és az őt megelőző összes, 0-nál nagyobb egész szám szorzata. 0 faktoriálisa azonban 1. A negatív számokra nem értelmezett a faktoriális.

Mielőtt az olvasó kétségbeesetten távolabb tartaná magától könyvünket, sietünk megnyugtatni, hogy az előző matematikai fejtegetés csupán a következő példa megértését segíti. A példa ugyanis faktoriálist számol.

Forrás: dowhile_syntax.php

dowhile_syntax.php

```php
<?php
    //dowhile_syntax.php
    header('Content-Type: text/html; charset=utf-8');
    $n=5;
    $i=1;
    $fakt=1;
    do{
        $fakt*=$i;
        $i++;
    } while($i<=$n);
    echo "$n!=$fakt";
?>
```

28. ábra do..while szerkezet

Az **$n** értéke határozza meg, hogy melyik szám faktoriálisát kérjük. A számolást úgy végezzük, hogy 1-től a **$n**-ig összeszorozzuk az összes számot. Mivel 0 és 1 faktoriálisa is egy, a **$fakt** változó kezdőértékét 1-re állítjuk. Elindítunk egy ciklust, amelynek minden végrehatásakor megszorozzuk a **$fakt** értékét az 1 kezdőértékű, de végrehajtásonként egyesével növekvő **$i**-vel. Az eredményt mindig visszatesszük a **$fakt** változóba.

A ciklust addig folytatjuk, amíg **$i** nem nagyobb, mint **$n**.

Mivel **$n** 0 és 1 értéke esetén is 1 a faktoriális, ciklusnak 1-szer mindenképpen le kell futnia, ezért alkalmas a hátul tesztelő ciklus.

> Ez természetesen nem jelenti azt, hogy a feladat csak így oldható meg. A program átírásával akár elöltesztelő ciklussal is megoldhattuk volna a faktoriális kiszámítását.

5.4.3 Léptető ciklus

Ha egy pillanatra visszatérünk az elöltesztelő ciklusnál látott példához, a következő mozzanatokra lehetünk figyelmesek:

- A ciklus megkezdése előtt **$i** kezdőértékét 0-ra állítottuk.
- A ciklusmag minden futása során megnöveltük **$i** értékét 1-gyel.
- Visszatértünk a ciklus elejére,
- és **$i** pillanatnyi értékét vizsgálva döntöttünk a további ismétlésről.

Az **$i** változót **ciklusváltozónak** nevezhetjük, mert **értéke a ciklusmag minden futásakor módosul**, befolyásolja az ismétlést, sőt azt is jelzi, hányadik végrehajtásnál tartunk.

A ciklusváltozó egyszerűbb kezelésére, és a fenti műveletek megfogalmazására alkalmas az úgynevezett **for**, vagy **léptető ciklus**.

```
for(kezdőérték; teszt; változtatás){
   ciklusmag
}
```

A **for** utasítást követő kerek zárójelek közötti elemek egy-egy kifejezést jelentenek. A *kezdőérték* **értékadó** kifejezés. Ezzel adjuk meg a **ciklusváltozó kezdőértékét**. A *teszt* egy **logikai kifejezés**. Ez alapján dől el, hogy **kell-e még ismételni** a ciklusmagot. A *változtatás* egy a ciklusváltozó értékét változtató **értékadó kifejezést** tartalmaz, ami mindig **a ciklus végén kerül** végrehajtásra.

Amikor a ciklus kezdődik, a ciklusváltozó értéket kap. Ez az értékadás csak egyszer zajlik le. Az értelmező mindig megvizsgálja azonban, hogy igaz-e a tesztkifejezés. Ha nem, akkor átlépi a ciklusmagot és folytatja a programot. Ha igaz, akkor végrehajtja a ciklusmag utasításait, majd a ciklusmag végén elvégzi a ciklusváltozót változtató kifejezést. Ezt követően visszatér a ciklus elejére, és újra tesztel.

Forrás: for_syntax.php

for_sysntax.php

```
1  <?php
2      //for_sysntax.php
3      header('Content-Type: text/html; charset=utf-8');
4      require "konyvtar.php";
5      $hossz=count($konyvek);
6      echo "<h1>Könyvek listája</h1>";
7      for ($i=0;$i<$hossz;$i++){
8          $sorsz=$i+1;
9          echo "$sorsz. ",$konyvek[$i]["cim"],"<br/>";
10     }
11 ?>
```

29. ábra for ciklus

Az fenti példa a **$konyvek** tömböt dolgozza föl. A **$hossz** a **$konyvek** tömb elemeinek száma. A ciklusmagban mindig az **$i** ciklusváltozónak megfelelő könyv címét írjuk ki. Az **$i** kezdőértéke 0 (**$i=0**), de az érték minden futás után 1-el nő (**$i++**).

A ciklus ismétlése előtt mindig meg kell vizsgálni, hogy **$i** nem érte-e el a tömb elemeinek számát. Azért van szükség a **$hossz** változóra, mert a tesztki-

fejezésben ehhez az értékhez hasonlítjuk a ciklusváltozót **($i<$hossz**). Ha a kifejezés más hamis, akkor már nem szabad tovább ismételni, hiszen nemlétező tömbelemre hivatkoznánk.

5.4.4 A foreach ciklus

A **foreach** ciklus a léptető ciklus egy speciális változata, amelyet általában tömbök bejárására használnak. Ez a ciklus a ciklusmag minden végrehajtása előtt a ciklusváltozóba másolja az aktuális tömbelemet. Az egyes ismétlések alkalmával mindig a következő elemet veszi. Az ismétlés addig tart, amíg a tömbelemek el nem fogytak. A ciklusmagban mindig a ciklusváltozó tartalmazza az aktuális tömbelemet.

```
foraeach(tömbváltozó as ciklusváltozó){
   ciklusmag
}
```

A következő példa egyszerűbben valósítja meg a **for** ciklusnál látott feladatot:

foreach_syntax.php

```
1  <?php
2      //foreach_syntax.php
3      header('Content-Type: text/html; charset=utf-8');
4      require "konyvtar.php";
5      echo "<h1>Könyvek listája</h1><hr/>";
6      $sorsz=1;
7      foreach ($konyvek as $konyv){
8          echo "$sorsz. ",$konyv["cim"],"<br/>";
9          $sorsz++;
10     }
11 ?>
```

30. ábra foreach ciklus

Forrás: foreach_syntax.php

A fenti példa ciklusa a ciklusmag minden futásakor a **$konyvek** tömb soron következő elemét másolja a **$konyv** változóba.

Mivel minden tömbelem egy asszociatív tömb, a cikluson belül a **$konyv** változó asszociatív tömb lesz. Mindig a "**cim**" elemének értékét írhatjuk ki.

5.4.5 A foreach ciklus asszociatív tömbökkel

A **foreach** ciklus előző a formáját kifejezetten numerikus tömbök egyszerű bejárására használjuk. Létezik azonban egy speciális változat is, ami asszociatív tömbök kezelésekor használható sikerrel. Ez a változat sorra veszi az asszociatív tömb elemeit, de nemcsak értéküket, hanem kulcsaikat is ciklusváltozóba teszi. Ennek megfelelően két ciklusváltozóra van szükség. Az egyik a tömb aktuális eleme kulcsának nevét, a másik az értékét kapja az egyes ismétlések alkalmával. A ciklus most is annyiszor fut le, ahány eleme van a tömbnek. A ciklusmagban a kulcsra és az értékre egyaránt hivatkozhatunk.

```
foreach (tömbváltozó as kulcs=>érték){
   ciklusmag
}
```

Forrás: foreach_assoc_syntax.php

```
foreach_assoc_syn...
 1  <?php
 2      //foreach_assoc_syntax.php
 3      header('Content-Type: text/html; charset=utf-8');
 4      require "konyvtar.php";
 5      echo "<h1>Első könyv adatai</h1>";
 6      $konyv=$konyvek[0];
 7      echo "<table>";
 8      foreach ($konyv as $cimke=>$adat){
 9          echo "<tr><td> <b>$cimke:</b> </td><td>$adat</td></tr>";
10      }
11      echo "</table><hr/>";
12  ?>
```

31. ábra Asszociatív tömb bejárása foreach ciklussal

A fenti példa a 0. könyv asszociatív tömbjét a **$konyv** változóba teszi, majd a **foreach** ciklussal végighalad a tömb elemein. A kulcs neve mindig a **$cimke**, a hozzá tartotó érték pedig az **$adat** változóba kerül. A programocska HTML-táblázatba illesztve jeleníti meg az asszociatív tömb adatait.

32. ábra A szkript eredménye a böngészőben

5.4.6 Egymásba ágyazott ciklusok

Az elágazásokhoz hasonlóan a ciklusok is egymásba ágyazhatók. Ez azt jelenti, hogy egy ciklus magjában egy másik ciklust helyezünk el. Ilyenkor a befoglaló ciklust **külső,** a beágyazott ciklust **belső** ciklusnak nevezzük. A belső ciklus a külső ciklus minden egyes végrehajtásakor teljesen végigfut, így ha a belső ciklus önmagában N-szer ismétlődne, a külső pedig M-szer, akkor a belső ciklus végrehajtásszáma N*M lesz.

Az egymásba ágyazott ciklusok kiválóan alkalmasak több dimenziós tömbök kezelésére.

Egymásba ágyazott ciklusokat célszerű használni például akkor, ha a **`$konyvek`** tömb minden elemét be akarjuk járni, de az egyes elemek asszociatív tömbjeinek elemeit is ki akarjuk írni.

Forrás: foreach_embed_syntax.php

foreach_embed_syn...

```php
<?php
    //foreach_embed_syntax.php
    header('Content-Type: text/html; charset=utf-8');
    require "konyvtar.php";
    echo "<h1>Könyvek listája</h1><hr/>";
    foreach ($konyvek as $konyv){
        echo "<table>";
        foreach ($konyv as $cimke=>$adat){
            echo "<tr><td> <b>$cimke:</b> </td><td>$adat</td></tr>";
        }
        echo "</table><hr/>";
    }
?>
```

33. ábra Egymásba ágyazott ciklusok

5.4.7 Ciklus elhagyása

A ciklusok befejezése általában akkor történik, amikor az ismétlés feltétele már nem teljesül. Van azonban két rendhagyó eset. Az egyik a **break** a másik a **continue** utasítás használata. Ha az értelmező a ciklusmagban egy **break** utasításhoz ér, akkor azonnal megszakítja a ciklus végrehajtását, és a ciklusvég utáni utasításra lép.

A másik hasonló utasítás a **continue**, amely abban különbözik a **break**-től, hogy hatására nem a teljes ciklus, hanem csak a ciklusmag aktuális ismétlése fejeződik be. Az értelmező ugyanis nem a ciklusvég utáni utasításra, hanem vissza a ciklusfejhez lép, és a ciklusmag következő végrehajtásával folytatja a programot.

5.5 ÖSSZEFOGLALÁS, KÉRDÉSEK

5.5.1 Összefoglalás

A Böhm–Jacopini-tétel szerint bármilyen program elkészíthető utasítássszekvenciák, iterációk – azaz ismétlések – és szelekciók, más szóval elágazások sorozatával. Mai leckében a PHP-nak a szelekciók és iterációk megvalósítására alkalmas vezérlési szerkezeteivel ismerkedtünk meg.

Megtanultuk, hogy az elágazások olyan pontok a programban, ahol az utasítások kettő, vagy több ágra bomlanak, és az interpreter valamilyen tesztkifejezés vizsgálatával dönti el, hogy melyik ágat hajtsa végre. A PHP az **if**…**else** szerkezettel biztosítja a kétágú elágazások készítését. Az **if**…**elseif**…**else** szerkezet a kétágú elágazás továbbfejlesztése, valójában az egymásba ágyazott elágazások egyszerűbb leírására alkalmas. Ezzel azonban a többágú elágazás implementálásának egyik eszközévé válik. A klasszikus többágú elágazás kivitelezésre a **switch** struktúrát használhatjuk a PHP-ben. Az ismétlések kapcsán különbséget tettünk az elöl- és a hátultesztelő, valamint a léptető ciklusok között.

Megállapítottuk, hogy az elöl- és hátultesztelő ciklusok között a formai megvalósításon túl a minimális ismétlésszám jelenti legfontosabb különbséget. A hátultesztelő ciklusok egyszer biztosan lefutnak, az elöltesztelők esetleg egyszer sem hajtódnak végre.

A léptető ciklusok az elöltesztelő ciklusokhoz tartoznak, azonban a ciklusfejben elhelyezett kifejezések segítségével maguk kezelik a ciklusváltozót és annak tesztelését. A **foreach** a léptető ciklus speciális változata, amely tömbök bejárására biztosít kényelmesen használható nyelvi elemeket.

5.5.2 Önellenőrző kérdések

1. Hogyan készítene többágú elágazást, ha csak if…else szerkezetet használhatna?
   - Valódi többágú elágazást sehogy, de az egymásba ágyazott if…else utasításokkal meg lehet oldani a többszörös elágazást.
2. Miért kell a switch szerkezet minden ágát break utasítással zárni?
   - Mert különben a végrehajtás a következő ágban folytatódna.
3. Mi a legfontosabb különbség az elöl- és a hátultesztelő ciklusok között?
   - A hátultesztelő ciklus egyszer biztosan lefut.
4. Milyen értékeket ír ki az alábbi program?

   for($i=10;$i>0;$i-=2){
   echo „$i
”;
   }
   - A páros számokat 10-től 2-ig, csökkenő sorrendben.
5. Elemezze az alábbi programot! Mi lesz az alábbi program eredménye?

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <head>
        <meta Http-equiv="Content-type" Content=""=text/html; charset=utf-8">
        <style>.head{background-color:#cccccc;}</style>
    </head>
    <body>
        <?php
            echo "<table border='1'>";
            echo "<tr>";
            for ($j=0;$j<=10;$j++){
                if ($j<>0){
                    echo "<td class='head'>$j</td>";
                } else {
                    echo "<td class='head'> </td>";
                }
            }
            echo "</tr>";
            for ($i=1;$i<=10;$i++){
                echo "<tr>";
                for ($j=0;$j<=10;$j++){
                    if ($j<>0){
                        echo "<td>",$j*$i,"</td>";
                    } else {
                        echo "<td class='head'>$i</td>";
                    }
                }
                echo "</tr>";
            }
            echo "</table>";
        ?>
    </body>
</html>
```

34. ábra Forráskód (feladat)

Forrás: szorzotabla.php ☺

6. LECKE: KLIENSOLDALI ADATOK FELDOLGOZÁSA, ÁLLAPOTKEZELÉS

6.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

A webprogramozásban használt adatok egy része a programban elhelyezett literál, más részük a literálok értekeiből számított adat, vagy fájlokból, adatbázisokból betöltött érték, de az adatok igen jelentős része a felhasználótól származó, az alkalmazás kezelőfelületén, azaz weblapokon bevitt adat. A felhasználói adatbevitel kezelése minden számítógépes program elkészítésének kiemelt, sőt, talán legmunkaigényesebb feladata. A program ugyanis csak a helyes és feldolgozható adatok alapján képes hibamentesen működni. Amíg a program a programozó által megadott adatokon dolgozik, ez a hibamentesség garantálható. A felhasználói adatbevitel azonban időzített bomba. A véletlenül – vagy akár szándékosan – elrontott adatok kezeletlen állapotba juttathatják a programunkat, sőt, akár katasztrofális adatvesztéseket is okozhatnak.

A webalapú alkalmazások programozásakor nemcsak az adatok helyességének garantálása okoz gondot. Míg a hagyományos programok kezelőfelülete a programot futtató gépen jelenik meg, addig a webalkalmazások esetén a program az alkalmazásszerveren fut, míg a felület kezelése a kliens gépén működő webböngészővel történik. A távoli gépen futó alkalmazásnak olyan kezelőfelületeket kell előállítania, amely alkalmas a webböngészőben való megjelenítésre. A problémát tovább fokozza, hogy a HTTP protokoll alapvető jellemzője az állapotmentesség.

Mai leckénkben a felhasználói adatbevitel kezelésének alapjaival ismerkedünk meg. Arról fogunk beszélni, hogyan alakíthatjuk ki a kezelőfelületet, és hogyan biztosíthatjuk az adattok eljuttatását a szerver oldalra. A leckében arra is kitérünk, hogy milyen lehetőségek vannak a HTTP protokoll állapotmentessége okozta problémák orvoslására.

6.2 FELHASZNÁLÓI FELÜLETET ELŐÁLLÍTÁSA

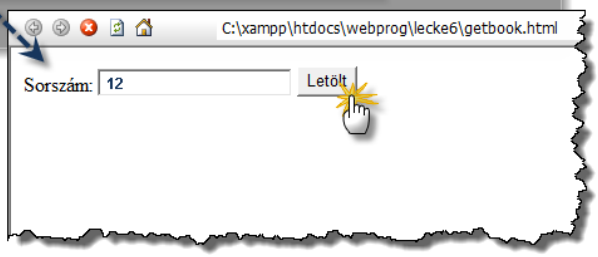
A felhasználói adatbevitel kezelésének egyik fontos eleme a kliensoldali felület előállítása. A szerveroldali programmal olyan HTML-oldalt kell készítenünk, amin a felhasználó be tudja gépelni az adatokat, majd el tudja küldeni azokat a szerveroldali alkalmazásnak.

Az adatbevitel biztosítására a HTML **`FORM`** elemével kialakított űrlapok használhatók. A **`FORM`** elembe számos olyan további jelölő **`(<input>`**

<TEXTAREA>, **<SELECT>**, **<FIELDSET>** **<LABEL>**…) ágyazható be, amelyek vezérlőelemek (beviteli mezők, listák, lenyíló listák, rádiógombok, kiválasztó négyzetek…) megjelenítését teszik lehetővé. A kitöltött űrlap elküldése egy nyomógombbal lehetséges, ami a **SUBMIT** típusú **INPUT** elem hatására jelenik meg a böngészőben.

Forrás: getbook.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <head>
        <title></title>
        <meta name="" content="">
        <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    </head>
    <body>
        <form action="getbook.php" method="GET">
            <label for="ssz">Sorszám:</label>
            <input type="text" name="sorszam" id="ssz" />
            <input type="submit" value="Letölt"/>
        </form>
    </body>
</html>
```



35. ábra HTML-űrlap a böngészőben

A **FORM** jelölőn belül elhelyezkedő adatbeviteli elemek mindegyike rendelkezik az elem nevének megadására használatos **NAME** attribútummal.

Ha az elembe beviszünk valamilyen adatot, akkor egy név-érték pár keletkezik. A név az elem neve az érték a begépelt adat. Ezeket a név-érték párokat kell eljuttatni a szerver oldalra.

Az adatok elküldésében **FORM** elem **ACTION** és a **METHOD** attribútumai játsszák a legfontosabb szerepet. Az **ACTION** attribútum egy szerveroldali programfájl URL-címe, a **METHOD** pedig a HTML-kérésekben használható POST vagy GET metódusnevek egyike. Amikor a felhasználó kitölt, majd elküld egy űrlapot, akkor a böngésző a metódusnak megfelelő kérést intéz a szerverhez. A kérésben az **ACTION** tulajdonságban megadott programfájlt kéri.

Ha az **ACTION** tulajdonságban nem adtunk meg a fájlnevet, vagy teljesen hiányzik az attribútum, akkor a kérés magára az űrlapot tar-

talmazó állományra vonatkozik. Ezt a technikát akkor alkalmazzuk, amikor a felhasználói felületet előállítását, és a felületen bevitt adatok feldolgozását ugyanaz a szerveroldali szkript végzi.

Az űrlapon megadott adatok *név=érték* formában kerülnek a kérésbe. GET metódus esetén az URL-címhez kapcsolódnak, POST esetén a kéréstestbe ágyazza őket a böngésző.

Tegyük fel, hogy a feladatunk, olyan „alkalmazás” készítése, amellyel a felhasználó a szerveroldalon tárolt könyvek közül le tudja tölteni az általa megadott sorszámhoz tartozó könyvcímet.

A szerveroldali programban rendelkezésünkre áll egy numerikus tömb, aminek minden eleme egy könyv adatait tároló asszociatív tömb. A programban elhelyezett ***$sorszam*** *változóban tárolt értékkel tömb bármelyik elemét ki tudjuk választani, majd ki tudjuk írni a kiválasztott elem "****cim****" kulcsához tartozó értéket. A programot azonban bármennyiszer is futtatjuk, mindig ugyanazt a címet jeleníti meg mindaddig, amíg át nem írjuk a* ***$sorszam*** *értékét.*

Forrás: onebook.php

onebook.php

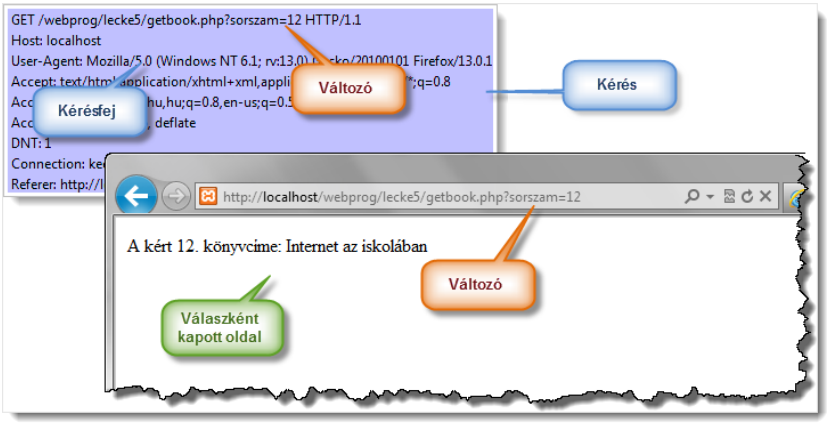
```
1  <?php
2      //onebook.php
3      header('Content-Type: text/html; charset=utf-8');
4      require "konyvtar.php";
5      $sorszam=10;
6      echo "A kért $sorszam. könyvcíme: ",$konyvek[$sorszam]["cim"],"<br/>";
7  ?>
```

36. ábra Egy könyv megjelenítése

Ha azt szeretnénk, hogy a felhasználó képes legyen változtatni a ***$sorszam*** *értékén, akkor weblapot kell készítenünk, amin begépelhető, és a programnak átadható a kívánt sorszám. Így mindig a kiválasztott könyv fog megjelenni.*

6.2.1 GET metódussal küldött kérés

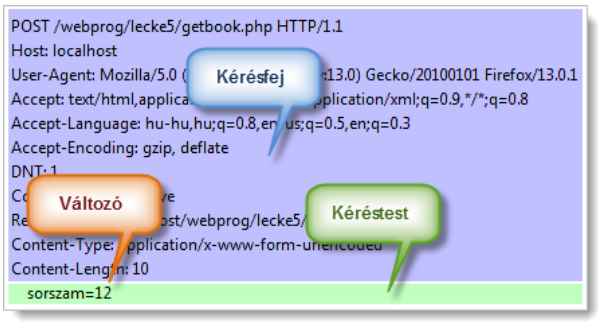
Ha a kérést GET metódussal küldtük, a válasz megérkezésekor a böngésző címsorában láthatók is az URL-hez kapcsolt változók. Az akkor lehet hasznos, ha még kissé járatlanok vagyunk az űrlapok készítésében, és szeretnénk látni a küldött változók értékét.



37. ábra Adatküldés GET metódussal

A GET metódust használva a felhasználók is ismerni fogják a szerver és az alkalmazás között küldött változók neveit és azok értékeit. Ez pedig biztonsági problémákat vethet fel. A végleges alkalmazásban a POST metódus biztonságosabb.

Ha a böngésző POST metódussal küldi kérést a címmezőben nem jelennek meg változók. Minden adat az üzenettestben kerül továbbításra.



38. ábra POST metódussal küldött kérés

A mi roppant egyszerű felhasználói felületünk így fest:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <head>
        <title>Könyv címének letöltése</title>
        <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    </head>
    <body>

        <form action="getbook.php" method="GET">
            <label for="ssz">Sorszám:</label>
            <input type="text" name="sorszam" id="ssz" />
            <input type="submit" value="Letölt"/>
        </form>

    </body>
</html>
```

39. ábra Felhasználói felület

Forrás: getbook.html

A **FORM** elem – mint tudjuk – megjeleníti a beágyazott vezérlőelemeket, a felhasználó pedig kitöltheti azokat. A **FORM** a **METHOD** attribútumban megadott metódusnak (**GET**) megfelelő kérésben továbbítja az adatokat, az **ACTION** attribútumban (**getname.php**) megadott szerver oldali programnak. A kérést akkor küldi el a böngésző, amikor az űrlapon a **SUBMIT** típusú **INPUT** elemre (**Letölt** gomb) kattint a felhasználó.

Mivel a kérésben megadott fájl PHP-szkript (**getname.php**), a webszerver átadja a PHP-értelmezőnek a kért állományt, az interpreter pedig elkezdi a feldolgozást. Igen ám, de hogyan jut be az adat a feldolgozott programba? A webszerver a kérésben megadott fájl mellett, az egész kérést átadja az értelmezőnek, ami különböző változókat hoz létre a kérés elemeiből.

Ilyen változók a **$_POST**, és a **$_GET** asszociatív tömbök. Ezek elemei az adott metódussal feltöltött űrlapelemek neveit és azok értékeit tartalmazzák.

Ha tehát a kérés **GET** metódussal jött, és a felületen található űrlapon volt egy "**sorszam**" nevű elem (**<input id="ssz" name="sorszam" type="text" />**) akkor a **$_GET** tömbnek lesz "**sorszam**" kulcsú eleme, amely a feltöltött értéket tartalmazza: **$_GET["sorszam"]**

A kérést kezelő szkriptet tehát így írhatjuk meg:

```
<?php
    header('Content-Type: text/html; charset=utf-8');
    require "konyvtar.php";
    $sorszam=$_GET["sorszam"];
    echo "A kért $sorszam. könyvcíme: ",$konyvek[$sorszam]["cim"],"<br/>";
?>
```

40. ábra getbook.php

Forrás: getbook.php; konyvtar.php

A 2-es sor a válasz egy fejlécmezőjének beállításával (**Content-type**), szabályozza kliens karakterkódolását.

A 3-as sor becsatolja és értelmezi a **konyvtar.php** tartalmát. Ebben helyeztük el a könyveket tároló tömböt. Ez a megoldás azért is szerencsés, mert a **getbook.php** így rövidebb és áttekinthetőbb, és azért is, mert a **konyvtar.php**-ben tárolt asszociatív tömböt így más szkriptek is tudják használni.

Az 4. sor kiolvassa a **$_GET** tömb "sorszam" elemét, és az értékét a **$sorszam** változóba teszi.

Az 5. sor egy üzenetbe ágyazva kiírja a sorszámot, majd a **$konyvek** tömb **$sorszam**-adik eleméből a "**cim**" kulcsú elemet.

A kimenet a webszerverhez, onnan pedig a böngészőhöz kerül, így a felhasználó megkapja kért könyvcímet.

41. ábra GET metódussal letöltött válasz

A $_GET és a $_POST úgynevezett előre definiált változók, amelyek értékeit bármikor kiolvashatjuk. A kéréssel érkező adatok egyébként a $_REQUEST előre definiált tömbváltozóba kerülnek, amiben a ”_GET”, ”_POST” elemeken keresztül is hozzáférünk a feltöltött adatokhoz. A $_REQUEST ezen kívül egy harmadik ”_COOKIE” nevű elemet is tartalmaz, amibe a böngészővel küldött cookie adatok kerülnek. A cookie-król még ebben a leckében olvashat.

6.2.2 Adatbevitel ellenőrzése

A fenti példa működik, de csak erős túlzással nevezhető webalkalmazásnak. Először is mert szolgáltatásai túlságosan szerények ehhez a kategóriához, másodszor mert egyszerűsége mellett ezer sebből vérzik, számtalan hibát tartalmaz.

- Miután a felhasználó megkapta a kért nevet, nem tud újat kérni, mert a kezelőfelület eltűnt. Visszaléphet ugyan a böngésző **Vissza** gombjával, és begépelhet új sorszámot, de ez nem túlságosan elegáns megoldás. Az **alkalmazásnak minden állapotban tartalmaznia kell** azokat a kezelőeszközöket, **navigációs elemeket**, amelyekkel az adott állapothoz logikusan kapcsolódó többi **állapot elérhető**!
- Sem kliensoldalon, sem szerveroldalon nem ellenőrizzük a felhasználó által megadott adatot, azt helyesnek és biztonságosnak ítélve azonnal dolgozni kezdünk vele. Ez hibát okozhat, ha a felhasználó nem ír be semmilyen értéket a szövegmezőbe, netán negatív, vagy a tömb méretéhez képest túlságosan nagy értéket adott meg. Ilyenkor programunk hibaüzenet küld, ami megjelenik a böngésző felületén. Ez bizony meglehetősen csúf malőr. A **programnak** úgymond **„bolondbiztosnak" kell lennie**, azaz a felhasználó bármilyen hibájára föl kell készülnie.

> **Nem fordulhat elő, hogy az alkalmazás nem várt hiba miatt leálljon, és ne biztosítsa a továbblépéshez szükséges felületet. A program nem kerülhet definiálatlan, kezeletlen állapotba.**

Az alábbi példa mindkét problémát megoldja:

book_title.php

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <head>
        <title>Könyv címének letöltése</title>
        <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    </head>
    <body>
        <?php
            if (count($_GET)>0){
                //book_title.php
                require "konyvtar.php";
                $konyvszam=count($konyvek);
                if (isset($_GET["sorszam"]) and $_GET["sorszam"]>0 and $_GET["sorszam"]<=$konyvszam){
                    $sorszam=$_GET["sorszam"];
                    $index=$sorszam-1;
                    echo "A kért $sorszam. könyv címe: ",$konyvek[$index]["cim"],"<br/>";
                } else {
                    echo "A sorszám helytelen! Csak 1 és $konyvszam közötti értéket adhat meg";
                }
                echo "<hr/>";
            }
        ?>
        <p><b>Adja meg a kért könyv sorszámát!</b></p>
        <form action="book_title.php" method="GET">
            <label for="ssz">Sorszám:</label>
            <input type="text" name="sorszam" id="ssz" size="3"/>
            <input type="submit" value="Letölt"/>
        </form>

    </body>
</html>
```

42. ábra book_title.php

Forrás: book_title.php

Először is figyeljük meg, hogy a **PHP-kódot HTML-szövegbe ágyaztuk**, de az állomány **php kiterjesztést** kapott (**book_title.php**)! Ha a böngészőnkkel a **book_title.php-t** töltjük le, a webszerver előbb a PHP-értelmezőnek adja a fájlt, és csak a kapott kimenetet küldi vissza a böngészőnek.

A fájl egyben tartalmazza a kezelőfelületet, és a szerveroldali feldolgozó programot is. A 8–22. sorok közötti PHP-kód a weblap egy részét, a kért könyv címét tartalmazó szöveget állítja elő. A weblap többi része a tájékoztató feliratot és a sorszám beolvasását szolgáló űrlapot jeleníti meg. **A PHP-kódnak csak akkor kell lefutnia, ha a felhasználó már beírt, és feltöltött egy sorszámot.**

Lássuk, hogyan működik a program! A fájl feldolgozását HTML-módban kezdi az értelmező. Csak a 8. sorban vált PHP-módba, addig mindent a kimenetre küld. A 9. sor ellenőrzést tartalmaz. Megvizsgálja, hogy van-e valamilyen elem a **$_GET** tömbben. Ha az oldalt most tölti először a felhasználó, akkor nyilván nem adott meg sorszámot, ilyenkor a **$_GET** tömb üres, elemszáma 0. Ha erről van szó azonnal a 21. sorra kerül a vezérlés, tehát az értelmező szinte a teljes PHP-kódot kihagyja. A 22-es sorban újra HTML-mód következik és az űrlapot leíró HTML-kód változatlanul a kimenetre megy.

Figyeljük meg, hogy az űrlap **ACTION** paramétere az oldalt előállító **book_title.php** fájlra hivatkozik. Amikor a felhasználó begépeli a sorszámot, majd a **Letölt** gombra kattint, a böngésző ugyanazt a fájlt kéri a szervertől, ami magát a weblapot is előállította. A begépelt adat feldolgozását tehát a **book_title.php** végzi. Most azonban a **$_GET** tömb már nem lesz üres, tartalmazni fogja a **"sorszam"** nevű elemet. Emiatt a 9. sor után nem a 22-re, hanem a 10-re kerül a vezérlés, azaz nem lépjük át a könyvet kiválasztó PHP-kódot. A 11. sorban a **require** utasítás hatására az interpreter beolvassa és értelmezi a **konyvtar.php** szkriptet, így a program tartalmazni fogja a **$konyvek** tömböt. A 12. sor lekérdezi, és a **$konyvszam** változóba teszi a **$konyvek** tömb elemszámát. Innen fogjuk tudni, hogy hány könyv van a tömbben. A 13. sor a nagyon fontos feladatot lát el. Itt ellenőrzi a program, hogy a felhasználó által feltöltött adat megfelelő-e.

A felhasználó által begépelt adat ellenőrzését sohasem szabad elmulasztani! Csak olyan értéket szabad elfogadni, amelynek feldolgozásá-

ra felkészítettük a programunkat! Ha a bevitt adat nem megfelelő, hibaüzenetet kell küldeni, és biztosítani kell a továbblépés lehetőségét.

Példánkban ellenőrizni kell, hogy van-e egyáltalán **sorszam** névvel feltöltött adat, és ha igen, akkor értéke 1 és **$konyvszam** között van-e. Ha bármelyik feltétel nem teljesül, akkor hibásnak kell tekinteni a bevitelt, és nem szabad megpróbálni a könyv megjelenítését. Ilyenkor a vezérlés a 18. sorra kerül, és a program hibaüzenetet ír a kimenetre.

Ha a bevitt adat megfelelő, akkor elkezdődhet a feldolgozás. A 14. sorban a **$sorszam** változóba tesszük a **$_GET[„sorszam”]** tömbelem értékét. Azonban nem feledkezünk meg arról, hogy a tömbünk első elemének 0, az utolsó elemnek pedig ennek megfelelően **$konyvszam-1** az indexe. A 15. sorban létrehozzuk az **$index** nevű változót, ami a **$sorszám**-nál egyet kisebb értéket kap. Ezzel hivatkozhatunk majd a megfelelő tömbelemre.

A 16. sor a kiírást végzi. A felhasználói felületen megjelenő üzenetbe ágyazza a sorszámot, és a **$konyvek** tömb **$index**-edik elemének [**cim**] kulcsú értékét.

A 22. sorban ismét HTML-módba vált az értelmező, és a kimentre írja az űrlapot is. Így a felhasználó újabb sorszámot adhat meg.

6.3 AZ ALKALMAZÁS ÁLLAPOTAINAK KEZELÉSE

Többször említettük már, hogy az alkalmazás működése során a program állapotai alakulnak át a következő állapotba. Az állapotok pedig adatokkal, a program által éppen tárolt változókkal és értékeikkel írhatók le.

A webprogramozás egyik nehézségeként neveztük meg a HTTP protokoll állapotmentességét. Azt a sajátosságot, hogy a webszerver nem tárolja a böngésző előző kapcsolódását leíró adatokat, ezért a következő kapcsolósáskor nem lehet reprodukálni a korábbi állapotot.

Lássunk most egy példát az állapotmentesség okozta problémára.

A feladat egyszerű. Olyan alkalmazást szeretnénk készíteni, amelynek felületén egy számot lehet begépelni és feltölteni. A feldolgozó program hozzáadja az aktuálisan feltöltött értéket a korábban beolvasott számok összegéhez, kiírja az eredményt és a következő szám bevitelére alkalmas űrlapot is.

Forrás: sum_rossz.php

sum_rossz.php

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
2  <html>
3      <head>
4          <title>sum_rossz.php</title>
5          <meta Http-equiv="Content-type" Content="=text/html; charset=utf-8">
6      </head>
7      <body>
8          <?php
9              if (!isset($sum)) $sum=0;
10             echo "<p><i>A régi egyenleg: $sum</i></p>";
11             if (isset($_GET["a"])){
12                 $a=$_GET["a"];
13                 $sum+=$a;
14                 echo "<b>Feltöltött összeg: $a</b><hr/>";
15                 echo "<b>Az új egyenleg: $sum</b>";
16             }
17         ?>
18         <hr/>
19         <p><b>Írjon be egy összeget, hozzáadom az egyenlegéhez!</b></p>
20         <form method="GET" action="sum_rossz.php">
21             <label for="a">Összeg:</label>
22             <input name="a" id="a" type="text" size="2" />
23             <input type="submit" value="Add hozzá!"/>
24         </form>
25
26     </body>
```

43. ábra Nem működő összegzés

A felület előállítást és a feldolgozó programot az előző példához hasonlóan ötvöztük. A **sum_rossz.php** olyan weblapot állít elő, amelyen PHP-kód generálja az addigi feltöltött értékek egyenlegét, és HTML-űrlap teszi lehetővé a következő összeg begépelését. Az egyenleget **$sum** változó tartalmazza. Az oldal első letöltésekor ez természetesen üres, sőt, és nem is létezik. Ilyenkor program a 9. sorban létrehozza és 0 kezdőértékűre állíja a változót. Ha a **$sum**-ban van érték, akkor a 9. sorban nem történik semmi. A 10. sor kiírja a **$sum** aktuális értékét. Most következik a felhasználói adatbevitel ellenőrzése. Az űrlapon „a” az adatbevitelt szolgáló input mező neve, ezért a 11. sorban azt ellenőrizzük, hogy van-e GET metódussal feltöltött, „a” nevű adat. Ha igen, akkor az érték az **$a** változóba kerül, amit rögtön hozzá is adunk **$sum**-hoz. A program kiírja a feltöltött értéket (**$a**), és az új egyenleget is (**$sum**). Az értelmező ezután HTML-módba vált, és a kimenetre írja a HTML-űrlapot. Így a weblapon beírható lesz a következő érték. Ha ez megtörténik, a HTML-űrlap ugyanannak a programnak küldi adatokat, ami magát az weblapot is előállította.

A program látszólag hibamentesen működik, de az egyenleg nem az összegzett, hanem mindig a legutoljára feltöltött érték lesz.

A hibát az állapotmentesség okozza. Miután a szerver elküldi a kész weblapot a böngészőnek, megszakítja a kapcsolatot és „elfelejti” a változók értékeit. A legközelebbi feltöltéskor gyakorlatilag minden újraindul. Az **”a”** érték ugyan megérkezik a kéréssel, de a **$sum** változó mindig üres lesz, ezért újra és újra 0

értéket kap. Az új érték feldolgozásának kezdetén tehát, már nincs meg az utolsó futás befejezéskor aktuális egyenleg, azaz a legutolsó állapot.

Mi lehet a megoldás? Csakis az, ha kijátsszuk a HTTP állapotmentességét! Valamilyen technikával meg kell őriznünk az alkalmazás utolsó állapotát leíró változókat, és értéküket. A következő kérés megérkezésekor, a változók ismételt létrehozásával és előző értékeik beállításával reprodukálni kell az utolsó állapotot, és csak ezután kezdhetünk hozzá az új adatok feldolgozásához.

6.3.1 Rejtett input mezők használata

Az állapotmegőrzés egyik elterjedt technikája a rejtett input elemek használata. Amikor a szerveren futó program létrehoz egy weblapot, az utolsó állapotot leíró változókat rejtett input mezők formájában beszúrja abba az űrlapba, amelyen a következő adatokat be lehet majd gépelni. A rejtett INPUT-elemek állandó rögzített értéket tárolnak, és nem jelennek meg a felhasználó előtt, de az űrlap legközelebbi feltöltésekor, a begépelt adatokkal együtt eljutnak a szerverre. Az újrainduló program így reprodukálni tudja saját korábbi állapotát, majd ebből az állapotból kiindulva el tudja végezni az új adatok feldolgozását.

Forrás: sum_hidden.php

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <head>
        <title>sum_hidden.php</title>
        <meta Http-equiv="Content-type" Content="=text/html; charset=utf-8">
    </head>
    <body>
        <?php
            //sum_hidden.php
            //Utolsó állapot előállítása, vagy $sum első értéke
            if (isset($_GET["sum"])){
                $sum=$_GET["sum"];
            } else {
                $sum=0;
            }
            echo "<p><i>A régi egyenleg: $sum</i></p>";
            //Új adatok feldolgozása
            if (isset($_GET["a"])){
                $a=$_GET["a"];
                $sum+=$a;
                echo "<b>Feltöltött összeg: $a</b><hr/>";
                echo "<b>Az új egyenleg: $sum</b>";
            }
        ?>
        <hr/>
        <p><b>Írjon be egy összeget, hozzáadom az egyenlegéhez!</b></p>
        <form method="GET" action="sum_hidden.php">
            <label for="a">Összeg:</label>
            <input type="hidden" name="sum" value="<?php echo $sum; ?>"/>
            <input name="a" id="a" type="text" size="2" />
            <input type="submit" value="Add hozzá!"/>
        </form>

    </body>
</html>
```

44. ábra Rejtett mező az űrlapon

Az itt látható program bár majdnem teljesen megegyezik az előző, működésképtelen változattal, mégis megkerüli az állapotmentességet. A 11. sorban a **$_GET** tömbből próbálja beolvasni a **"sum"** elemet. Ha van ilyen, akkor értékét a **$sum** változóba teszi, különben 0-ra állítja a **$sum**-ot. Ezt követően csak a 29. sor tartalmaz érdekességet. Itt ugyanis a következő szám beolvasására való űrlapban elhelyezzük a „**sum**" nevű rejtett input mezőt. A mező VALUE attribútumát PHP-módba váltva a **$sum** változó értékének beírásával adjuk meg. Így visszaküldött weblap kódjában mindig tárolódik az előző állapot.

A rejtetett mezőket szinte minden webes alkalmazás használja valamilyen szinten. A technika egyszerű, akár a felhasználói oldalon futó programkódok számára is szolgáltathat adatokat. Az állapotértékek a kliensoldalon is jól követhetők, ami a fejlesztés közben különösen akkor jöhet jól, ha nincs módunk az alkalmazás nyomkövetésére. Ez az előny egyszersmind hátrány is, ugyanis a weblapok forrásának elemzésével a rosszindulatú felhasználók akár érzékeny adatokra is szert tehetnek. A másik komoly korlát, a tárolható adatszerkezetekben van. INPUT-mezőkben csak egészen egyszerű adatok tárolhatók. Nincs lehetőségünk például tömbök, vagy objektumok elhelyezésére.

6.3.2 Az állapotmegőrzés a cookie-k használatával.

A cookie-k, vagy sütik a kliensgépen tárolódó, és a böngésző által kezelt szövegfájlok. Minden cookie egy-egy domainhez kötődik, és a domainen belül működő alkalmazásokról tárol *név=érték* formátumú állapotadatot. Amikor a böngésző kapcsolatba lép egy webszerverrel mindig megvizsgálja, hogy nem tárol-e a szerver domain nevének megfelelő cookie-t. Ha igen, akkor a kérés fejlécmezőiben elküldi a süti tartalmát, a szerver oldalon futó alkalmazás pedig megkísérelheti előállítani az utolsó állapotot.

Ezeket a szövegfájlokat a böngésző kezeli, de a szerver- és a kliensoldalon futó programoktól is kaphat utasítást egy cookie értékének megváltoztatására. A szerver oldalról, a válasz üzenetekben található Set-Cookie üzenetfejlécek utasítják a böngészőt egy süti létrehozására.

A cookie használattal úgy lehet megőrizni egy állapotot, hogy a futó program, befejeződése előtt Set-Cookie fejlécekben csatolja a válaszüzenethez az állapotleíró változókat. Amikor a böngésző ilyen válaszüzeneteket kap, akkor készségesen elhelyezi az adatokat a cookie-kban. Amikor újra ehhez a szerver-

hez fordul, a kéréssel automatikusan elküldi a cookie-kban tárolt név-érték párokat a szervernek.

A PHP-alkalmazásból az alábbi függvényhívással lehet cookie adatokat illeszteni a válasz fejlécmezőibe:

```
setcookie(név, érték, lejárat, útvonal, domain)
```

- A *név* a tárolni kívánt változó neve.
- Az *érték* természetesen a rögzítendő adat, a változó értéke.
- A *lejárat* Unix-timestamp formátumú időpont, ami a cookie élettartamát határozza meg. Az idő lejárta után a böngésző automatikusan törli a sütit. Ha az érték nincs megadva, vagy nulla (0), akkor a cookie a böngésző bezárásakor törlődik.
- A *domain* határozza meg, hogy milyen szervereknek kell elküldeni a cookie-t.
- Az *útvonal* a szerver könyvtárszerkezetében megadott relatív útvonal. Ha beállítjuk, akkor csak a megadott mappában tárolódó szkriptek számára lesz használható a cookie. Ha nincs megadva, akkor az aktuális könyvtár az alapértelmezett.

Amikor a webszerver cookie-t tartalmazó kérést ad át a PHP-értelmezőnek, akkor az interpreter a **$_COOKIE** nevű asszociatív tömbbe helyezi a kapott sütiadatokat. Ezeket programban pontosan úgy tudjuk kezelni, mintha POST/GET metódussal feltöltött adatok lennének.

Forrás: sum_cookie.php

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
2  <html>
3      <head>
4          <title>sum_cookie.php</title>
5          <meta Http-equiv="Content-type" Content="=text/html; charset=utf-8">
6      </head>
7      <body>
8          <?php
9              //sum_cookie.php
10             //Utolsó állapot helyreállítása
11             if(isset($_COOKIE["sum"])){
12                 $sum=$_COOKIE["sum"];
13             } else {
14                 $sum=0;
15             }
16             echo "<p><i>A régi egyenleg: $sum</i></p>";
17             //Új adatok feldolgozása
18             if (isset($_GET["a"])){
19                 $a=$_GET["a"];
20                 $sum+=$a;
21                 echo "<b>Feltöltött összeg: $a</b><hr/>";
22                 echo "<b>Az új egyenleg: $sum</b>";
23                 //Állapot tárolása
24                 setcookie("sum",$sum,0);
25             }
26         ?>
27         <hr/>
28         <p><b>Írjon be egy összeget, hozzáadom az egyenlegéhez!</b></p>
29         <form method="GET" action="sum_cookie.php">
30             <label for="a">Összeg:</label>
31             <input name="a" id="a" type="text" size="2" />
32             <input type="submit" value="Add hozzá!"/>
33         </form>
34     </body>
35 </html>
```

45. ábra Állapot megőrzése cookie segítségével

A fenti kód nagyon hasonlít a rejtett INPUT-elemeket használó megoldáshoz. A két különbség **$sum** változó értékének megőrzésében és visszanyerésében van. A 11–12. sorban nem a **$_GET** tömbből, hanem a feltöltött cookie adatokat tároló **$_COOKIE** tömbből próbáljuk kiolvasni a **"sum"** elemet. A 24. sorban pedig, a **setcookie("sum",$sum,0)** függvényhívással hozzuk létre a válaszüzenetben a **Set-Cookie** fejlécet. Ezek fogják utasítani a böngészőt a cookie-k tárolására. A példában **setcookie** első paramétere a változónév, a második a pillanatnyi érték. A harmadik paraméterként megadott 0 érték jelzi, hogy a sütinek a böngésző bezárásakor kell törlődnie.

A cookie-k használata az INPUT mezőknél jóval egyszerűbben használható lehetőséget ad a fejlesztők kezébe. A cookie-kban tárolt adatok nehezebben hozzáférhetők a felhasználók számára, így a sütik a biztonságot is kevésbé veszélyeztetik. Sajnos ebben az esetben is csak egyszerű értékeket tárolhatunk, ráadásul a domain-enként tárolható sütik száma is korlátozott.

6.3.3 Munkamenetek használata

A session-, vagy más néven munkamenet-használat az állapotkezelés szerveroldali megvalósítása. A technika abban hasonlít a cookie-kkal való munkához, hogy az alkalmazások állapot adatai szintén szövegfájlokba kerülnek. Alapvető különbség azonban, hogy ezek a fájlok a **szerveren tárolódnak**, így a felhasználó nem férhet hozzájuk, csak a szerver oldalon futó program kezelheti őket.
Az állapotadatok a szerveroldali tárolása – a felhasználó saját gépén tárolódó cookie-kal szemben – azért okozna nehézséget, mert ugyanazt a szerveroldali alkalmazást egyszerre több felhasználó is használhatja. Az alkalmazás így több példányban fut, és a különböző példányai más-más állapotokban lehetnek. Az állapotokat leíró változókat tehát felhasználónként, ha úgy tetszik: alkalmazáspéldányonként elkülönítve, és azonosítható módon kell tárolni a szerveren!

A problémát úgy oldották meg, hogy amikor egy adott IP-címről egy bizonyos böngészőprogram kapcsolódik a szerver oldali programhoz, akkor az alkalmazás egy új példánya indul el. A példány úgynevezett **munkamenetet**, **session-t indíthat** a saját állapotának tárolására. A **munkamenet** fizikailag egy egyedi azonosítóval, úgynevezett **`session_id`**-vel elnevezett **fájl**, amelyben a **példány eltárolhatja** saját **állapotleíró adatait**. Amikor a szerveroldali program lefut, a böngészőnek küldött válaszában közli a munkamenet azonosítóját is.

A HTTP protokoll jellemzőinek köszönhetően ezután megszakad a kapcsolat a szerver és a böngésző között, de amikor a kliens újra kapcsolódik, visszaküldi a szervernek a **`session_id`**-t.

Ha a szerveroldali alkalmazás ismét létre akarja hozni a munkamenetet, megvizsgálja, hogy kapott-e a klienstől munkamenet azonosítót. Ha igen, akkor ez alapján kikeresi a munkamenet adatait tároló fájlt, kiolvassa az abban tárolt változókat, és reprodukálja saját utolsó állapotát. Ha valamilyen oknál fogva nem érkezett **`session_id`**, akkor új munkamenet indul.

A munkamenet-azonosítókat általában cookie-ként küldik ki a szerverek, így a visszaküldés automatikus. A cookie-t ilyenkor úgy állítja be a szerver, hogy az csak a böngészőpéldány futásáig éljen. Ha a felhasználó bezárja a böngészőt, a cookie törlődik, így a legközelebbi kapcsolódáskor új munkamenet indul.

Ha a kliens nem kezeli a cookie-kat, akkor a web alkalmazást úgy is el lehet készíteni, hogy a böngésző minden egyes interakcióban az URL címhez kapcsolva küldje el a **`session_id`**-t.

Fölvetődhet a kérés, hogy meddig őrzi meg a szerver egy munkamenet adatait. Vajon mi történik a munkamenettel, ha a felhasználó hosszú ideig inak-

tív, azaz nincs semmiféle interakció a szerver és a kliens között? (Munkahelyünkön bekapcsolva hagyjuk a gépünket, és hazamegyünk…) A PHP konfigurációjában beállítható a munkamenetek élettartama (**session.gc_maxlifetime**). Ha a böngésző az utolsó kérés-választ követően olyan hosszú ideig nem kapcsolódik, hogy ez az időtartam lejár, akkor a szerver automatikusan törli a munkamenetet. Ilyenkor alkalmazásunk szerver oldalon tárolt állapotadatai elvesznek, és a legközelebbi kapcsolódáskor új munkamenet indul.

6.3.4 A munkamentek kezelése

A munkamenetek általában a fent leírt módon, fájlokban tárolódnak, és kezelésüket a PHP beépített függvényeivel végezhetjük el. Lehetőségünk van azonban az adatok adatbázisokban való tárolására, és saját munkamenet-kezelő függvények írására is. A munkamenet-kezelés teljes bemutatása akár több leckét is megtölthetne. Tananyagunk keretei erre nem adnak lehetőséget, ezért példáink csak az alapokat mutatják be, arra azonban alkalmasak, hogy megértsük és használni kezdjük a session-öket.

Munkamenet indítása

A munkamenet elindítását a **session_start()** függvénnyel végezhetjük. A függvény hatására a PHP létrehoz egy **$_SESSION** nevű asszociatív tömböt, majd megvizsgálja, hogy az aktuális kérésben kapott-e **session_id**-t, és ha igen, akkor megvan-e a munkamenethez tartozó állomány. Ha a munkamenetfájl létezik, akkor az abban tárolt változók név-érték párjait elemekként a **$_SESSION** tömbbe helyezi az értelmező. Ha nincs meg a fájlt, akkor új session indul, és a **$_SESSION** tömb üres marad.

Állapot helyreállítása

Az alkalmazás úgy tudja helyreállítani előző állapotát, hogy megvizsgálja a **$_SESSION** tömb tartalmát, és betölti az ott található adatokat. A programot úgy kell megírni, hogy az alkalmazás indulásakor, amikor a tömbben még nincsenek korábbi adatok, a megfelelő indulóértékek kerüljenek a változókba.

Utolsó állapot tárolása

Amikor az alkalmazás egy adott állapotát kezelő szkript lefut, gondoskodnunk kell a leíró adatok tárolásáról. Ehhez csak annyit kell tennünk, hogy beírjuk az adatokat a **$_SESSION** tömb megfelelő elemeibe. A tárolásról, a session-állomány kezeléséről a PHP maga gondoskodik.

Forrás: sum_session.php

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
2  <html>
3      <head>
4          <title>sum_session.php</title>
5          <meta Http-equiv="Content-type" Content="=text/html; charset=utf-8">
6      </head>
7      <body>
8          <?php
9              //sum_session.php
10             //Munkamenet indítása
11             session_start();
12             //Utolsó állapot helyreállítása
13             if(isset($_SESSION["sum"])){
14                 $sum=$_SESSION["sum"];
15             } else {
16                 $sum=0;
17             }
18             echo "<p><i>A régi egyenleg: $sum</i></p>";
19             //Új adatok feldolgozása
20             if (isset($_GET["a"])){
21                 $a=$_GET["a"];
22                 $sum+=$a;
23                 echo "<b>Feltöltött összeg: $a</b><hr/>";
24                 echo "<b>Az új egyenleg: $sum</b>";
25             }
26             //Állapot leíró adatok tárolása a munkamenetben
27             $_SESSION["sum"]=$sum;
28         ?>
29         <hr/>
30         <p><b>Írjon be egy összeget, hozzáadom az egyenlegéhez!</b></p>
31         <form method="GET" action="sum_session.php">
32             <label for="a">Összeg:</label>
33             <input name="a" id="a" type="text" size="2" />
34             <input type="submit" value="Add hozzá!"/>
35         </form>
36     </body>
37 </html>
```

46. ábra sum_session.php

A fenti ábrán látható **sum_session.php** szkript sessionkezeléssel biztosítja az állapotmegőrzést. A munkamenetet a 11. sorban lévő **session_start()** függvényhívás indítja el. Ha az alkalmazást a böngésző elindítása óta már használtuk, akkor a **$_SESSION** tömbben ott vannak a tárolt adatok. Ezt ellenőrizzük a 13. sorban. Ha már van tárolt egyenleg, akkor a **$sum** változóba kerül. Ha még nincs, akkor a **$sum** a nulla (0) kezdőértéket kapja. A példa már csak a 27. sorban tartalmaz érdekességet. A PHP-utasítások itt érnek véget, itt kell tárolnunk az állapotot leíró változót. A **$_SESSION** tömbben most helyezzük el a **$sum** változó aktuális értékét.

A session-kezelés az állapotok megőrzésének meglehetősen elterjedt technikája. Az állapotleíró változók biztonságosan tárolódnak a szerveren, a felhasználók egyáltalán nem férnek hozzájuk, a kezelés egyszerű, ráadásul a munkamenet-fájlokban egészen bonyolult adatstruktúrák, tömbök, objektumok is tárolódhatnak.

Úgy tűnik ez az állapotmegőrzés leghatékonyabb technikája. Ez így is van, de a webalkalmazások ennek ellenére mindhárom módszert használják. Az űrlap adatok mellé nagyon gyakran szúrunk be egyszerűbb értékeket hordozó rejtett INPUT-mezőket, és általában a cookie-k szolgáltatásait is igénybe vesszük. Utóbbival kapcsolatban elég arra gondolnunk, hogy a munkamenet kezeléskor a session_id-t általában sütik formájában kapják meg és küldik vissza a böngészők.

6.4 FÁJLOK FELTÖLTÉSE

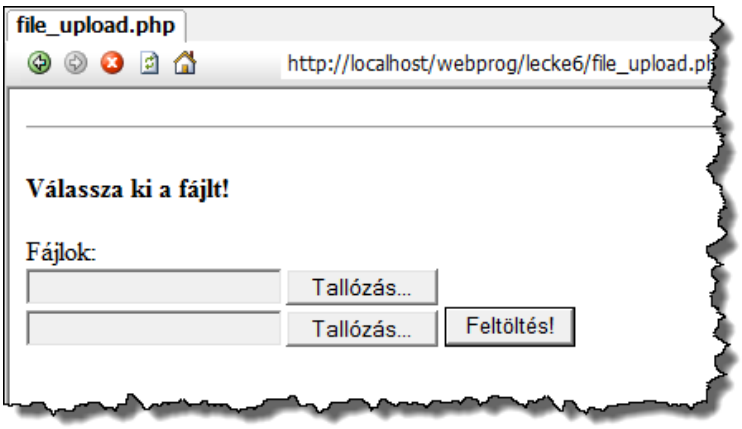
A fájlfeltöltés a felhasználói adatbevitel speciális formája. A HTTP protokoll nem csak a szerver számára teszi lehetővé a fájlok küldését. A **`POST`** metódussal küldött kérés üzenettestében gyakorlatilag bármilyen állományt feltölthetünk, egyedül arról kell gondoskodnunk, a szerver oldali szkript fel is dolgozza, lemezre, vagy adatbázisba mentse a dokumentumot.

6.4.1 Fájlfeltöltés kliens oldali feltételei:

Forrás: file_upload.php

A fájlok feltöltését a **`FORM`** elem, a felhasználói felületen fájlválasztóként megjelenő **`FILE`** típusú **`INPUT`** tag-je teszi lehetővé. A felhasználó az elem segítségével kiválaszthatja a saját gépe lemezein tárolt egyik fájlt, amely az űrlapadatokat feltöltő kérés üzenettestében kerül továbbításra.

Alábbi űrlap két ilyen elemet is tartalmaz, tehát egyszerre két fájl feltöltésére alkalmas.



47. ábra Fájlfeltöltésre használható űrlap

Ahhoz, hogy a kliensoldali felület megfelelően működjön, a feltöltést biztosító **FORM** elemnek ki kell egészülni az **ENCTYPE** attribútummal, amelynek értéke ebben az esetben kötött: "**multipart/form-data**". Ezen túl az űrlapon belül el kell helyezni egy rejtett, **MAX_FILE_SIZE** nevű **INPUT** elemet, amely arról tájékoztatja a böngészőt, hogy mekkora lehet a feltöltendő fájl maximális mérete. Értékét bájtban kell megadni.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <head>

        <meta Http-equiv="Content-type" Content="=text/html; charset=utf-8">
    </head>
    <body>
        <?php
                }
        ?>
        <hr/>
        <p><b>Válassza ki a fájlt!</b></p>
        <form action="file_upload.php" method="POST" enctype="multipart/form-data">
            <!-- 100 MB => 104857600 byte -->
            <input type="hidden" name="MAX_FILE_SIZE" value="104857600"/>
            <label for="f">Fájlok:</label><br/>
            <input name="f1" id="f1" type="file" /><br/>
            <input name="f2" id="f2" type="file" />
            <input type="submit" value="Küldd el!"/>
        </form>
    </body>
</html>
```

48. ábra Az űrlap HTML forrása

A PHP-vel feltölthető maximális fájlméretet **php.ini** **upload_max_filesize** direktívája állítja be. Ez alapértelmezésben 2 MB, de természetesen átállítható.

6.4.2 Feltöltött állományok kezelése

A kérésekkel érkező fájlokat ideiglenesen, véletlenszerűen generált egyedi névvel a **php.ini**-ben beállítható mappába (**upload_tmp_dir**) teszi a PHP. A fájlok leíró adatai a **$_FILES** asszociatív tömbbe kerülnek, aminek elemei az űrlap **FILE** típusú **INPUT** tagjeinek nevei lesznek. Az elemek szintén asszociatív tömbök, a fájl eredeti neve a **["name"]**, a mérete a **["size"]**, típusának MIME-kódja a **["type"]**, az ideiglenes állomány jelenlegi útvonala pedig a **[tmp_name]** tömbelemben található. Az **["error"]** elem a feltöltés hibakódját tartalmazza. A hibás feltöltést 0-tól eltérő érték jelezi.

Ha például az állományok mérete meghaladja a beállított limitet, akkor a hibakód 2.

$_FILES	
[f1]	
[error]	0
[name]	SZ030.jpg
[size]	339871
[tmp_name]	C:\xampp\tmp\php6376.tmp
[type]	image/pjpeg
[f2]	
[error]	0
[name]	SZ032.jpg
[size]	984468
[tmp_name]	C:\xampp\tmp\php6387.tmp
[type]	image/pjpeg

49. ábra Két feltöltött fájl adatai a $_FILES tömbben

A feltöltött fájlt a **`[tmp_name]`** tömbelemben található útvonalról kell a megfelelő helyre áthelyeznie a feldolgozó programnak.

```
8   <?php
9       //file_upload.php
10      if (count($_FILES)>0){
11          $destdir="upload/";
12          foreach($_FILES as $file){
13              if ($file["error"]==0){
14                  $src=$file["tmp_name"];
15                  $dest=$destdir.$file["name"];
16                  move_uploaded_file($src,$dest);
17              } else {
18                  echo "A ",$file["name"]," feltöltése nem sikerült!";
19              }
20          }
21      }
22  ?>
```

50. ábra Feltöltést kezelő szkript

- A művelethez a PHP **`move_uploaded_file(`***forrás*`,`*cél*`)** függvényét használhatjuk. Az első paraméter az ideiglenes fájl teljes útvonala, a második a végleges állomány helye és neve. Utóbbi megadható abszolút és – a szkript mappájából induló – relatív útvonallal is.

A **`file_upload.php`** 10. sorában megvizsgáljuk, van-e feltöltött fájl, majd a **`$destdir`** változóba tesszük a feltöltött fájlok végleges tárolására kiválasztott mappa nevét.

A 12. sorban kezdődő ciklus sorra veszi a feltöltött fájlokat. A **`$files`** ciklusváltozóba minden egyes iterációban a soron következő fájl adatai kerülnek.

Ha az adott állomány feltöltésekor nem történt hiba (13. sor) akkor az **$src** változóba az ideiglenes útvonalat (14. sor), a **$dest** változóba pedig a célútvonalat tesszük (15. sor). A 16. sorban, **move_upload_file()** függvény segítségével kerül a helyére a fájl.

6.5 ÖSSZEFOGLALÁS, KÉRDÉSEK

6.5.1 Összefoglalás

Tananyagunk 6. leckéje a felhasználótól érkező adatok fogadásáról, feldolgozásáról, és többállapotú alkalmazások állapotmegőrzéséről szólt.

A leckében megtanultuk, hogy a felhasználói oldalon a HTML **FORM** tagjével és az abba ágyazott elemekkel létrehozott űrlapokon van lehetőség. Az űrlap adatai név=érték formában kerülnek a szerverre. GET metódus esetén az URL-címhez kapcsolva, POST esetén pedig a kéréstestbe ágyazva.

A feltöltött adatokat **FORM** tag **ACTION** attribútumában jelzett szkript kapja meg. Az értékek a **$_GET**, illetve a **$_POST** asszociatív tömbökből olvashatók ki, és használhatók fel a szerver oldali alkalmazásban. Nagyon fontos, hogy a feltöltött adatokat mindig ellenőrizzük, és csak akkor használjuk fel, ha azok megfelelnek a programunk követelményeinek.

Ha olyan alkalmazást készítünk, ami több állapoton keresztül jut el a feladat megoldásához, akkor valamilyen technikával gondoskodnunk kell az utolsó állapot adatainak megőrzéséről. Erre a kliensnek küldött weblapok forrásában tárolódó rejtett INPUT-mezők, a cookie-k illetve a munkamenet kezelés ad lehetőséget.

A leghatékonyabb a sessionök kezelése, mert az állapotadatok biztonságosan tárolódnak a szerveren, szerver oldali hozzáférés egyszerű, és bonyolult adatszerkezetek tárolására is van lehetőség. Ennek ellenére a három tanult technikát vegyesen szokták alkalmazni a webprogramozók.

6.5.2 Önellenőrző kérdések

1. **Mi a különbség az űrlapadatok POST és a GET metódussal történő feltöltése között?**
   - A GET metódus esetén az URL-cím végéhez illesztve, a POST esetén pedig a kéréstestbe ágyazva jutnak a szerverre az adatok. A POST metódussal feltöltött értékek rejtve maradnak a felhasználó előtt.

2. Miért fontos a feltöltött adatok ellenőrzése?
   - Azért mert a felhasználó által megadott hibás adatok kezeletlen állapotot idézhetnek elő a szerver oldali alkalmazásban. Ez jó esetben csak hibaüzenetet, rossz esetben akár súlyos adatvesztést is okozhatnak.
3. Hogyan fér hozzá a szerver oldalon futó program a felhasználói felületről feltöltött adatokhoz?
   - A POST metódussal feltöltött adatok a $_POST, a GET metódussal küldött változók pedig a $_GET asszociatív tömbökön keresztül válnak elérhetővé.
4. Hogyan küldhet cookie-ként tárolódó adatokat a böngészőnek?
   - A setcookie(név, érték, lejárat, útvonal, domain) függvénnyel.
5. Mi a sessionök és a cookie-k használata közötti alapvető különbség?
   - Az, hogy a cookie adatok a kliensoldalon, a felhasználó számítógépén lévő szövegfájlokban vannak, és a böngésző kezeli őket, a munkamenetek pedig a szerveren tárolódnak. A munkamenetet leíró adatok általában munkamenetfájlba kerülnek, de akár adatbázisban is tárolódhatnak.

7. LECKE: ALPROGRAMOK ÉS PARAMÉTERÁTADÁS, BEÉPÍTETT FÜGGVÉNYEK

7.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

Az előző fejezetben láthattuk, hogy még egy egyszerű feladat megoldása is jelentős méretű programkódot eredményez. Elő kell állítani a kliensoldali felületet, be kell olvasni az adatokat, el kell juttatni azokat a szerveroldalra, ellenőrizni kell a bevitelt, föl kell dolgozni a változókban tárolt értékeket, elő kell állítani a kimenetet, azaz az új állapotnak megfelelő felhasználói felületet, és gondoskodni kell az utolsó állapot rögzítéséről is.

A bonyolultabb feladatok megoldásakor a kód jóval nagyobb lesz. A méret növekedésével csökken az áttekinthetőség és egyre nehezebbé válik a program utólagos megváltoztatása, az esetleges hibák javítása.

A bonyolult és áttekinthetetlen programkód kialakulását alapos tervezőmunkával lehet megelőzni. Nagyobb programok esetén a teljes, összetett feladatot részfeladatokra bontjuk, és azok megoldására külön-külön programkódokat, úgynevezett alprogramokat készítünk. A teljes feladatot a főprogrammal oldjuk meg. Amikor a főprogram kódjában egy részfeladat elvégzésére van szükség, akkor egyszerűen meghívjuk, elindítjuk a megfelelő alprogramot. A főprogram ilyenkor csak az alprogram befejeződése után, de az attól visszakapott adatok felhasználásával folytatódik.

Mai leckénkben a PHP alprogramjaival, az úgynevezett függvényekkel foglalkozunk.

- A lecke végére tudni fogja, hogyan hozhat létre saját függvényeket.
- Megismeri a formális és aktuális paraméterlista fogalmát.
- Megérti a globális és a lokális változók jelentőségét.
- Tudni fogja, mit jelent és hogyan hozható létre az érték és a cím paraméter.
- Képes lesz alapértelmezett paraméterek kialakítására.
- Tudni fogja, hogyan hívhat meg függvényeket, és hogyan adhat vissza értékeket a főprogramnak.

7.2 FÜGGVÉNYEK MŰKÖDÉSE

A bonyolultabb programokat főprogramra és alprogramokra bontva készítjük el. A főprogram kódja vezérli a feladat megoldásának egészét, az alprogramok pedig a kisebb részfeladatokat oldják meg.

> **Az alprogramok a főprogramtól és a többi alprogramtól elkülönülő programkódok, amelyek csak a saját részfeladatukhoz kapcsolódó tevékenységet végeznek, és csak az ehhez kapcsolódó adatokkal dolgoznak.**

A PHP alprogramjait **függvényeknek** nevezzük. A függvények csak saját részfeladatukon és az ehhez kapcsolódó adatokon dolgoznak. A részfeladat kezdőállapotából befejező állapotot, a részfeladat megoldását állítják elő.

Amikor a főprogramban olyan részfeladat végrehajtására van szükség, amelyre már elkészítettük a megfelelő alprogramot, a főprogram elindítja, **meghívja a függvényt** és átadja a részfeladat **kezdőállapotát leíró adatokat,** az úgynevezett **paramétereket.** A függvénynek csak a paraméterekre van szüksége feladata megoldásához. A feladat eredménye lehet **műveletek** elvégzése (kimenet előállítása, adatok fájlba írása, adatbázis művelet…) vagy olyan érték előállítása, amit az alprogram úgynevezett **visszatérési értékként** ad vissza a főprogramnak.

> **A függvények a PHP-program részfeladatait megoldó alprogramok. Hívásukkor feladatuk kezdőállapotát leíró paramétereket vehetnek át a hívás helyéről, a feladat megoldása nyomán keletkező eredményt pedig visszatérési értékként adhatják vissza a hívónak.**

A függvények természetesen nemcsak a főprogramból, hanem másik alprogramból is hívhatók. Ezért beszélünk hívás helyéről és hívóról.

Valójában eddig is többször használtunk függvényeket, de ezek a nyelv beépített elemei voltak. A PHP-t gyakran emlegetik függvényalapú nyelvként, ami arra utal, hogy a programozásban gyakran szükséges részfeladatokra a nyelv számos beépített függvényt tartalmaz.

Ilyen beépített függvény például a **`count()`**, amely egy tömböt vár paraméterként és visszatérési értéke a tömb elemeinek száma.

A **`$konyvszam=count($konyvek);`** programsorban meghívjuk a **`count()`** függvényt, paraméterként átadjuk a **`$konyvek`** tömböt, a visszaté-

résí értékként kapott elemszámot pedig későbbi felhasználásra a **`$konyvszam`** változóba tesszük.

A függvény zárt egység. Saját részfeladatát a főprogramtól és a program más részeitől függetlenül végzi. A főprogramnak sem kell ismernie a függvény utasításait. A főprogram és az alprogram csak a híváskor átadott paramétereken és a visszatérési értéken keresztül kommunikál egymással.

Amikor a teljes feladat bonyolultságát, a részfeladatokat megvalósító alprogramokkal csökkentjük, a **strukturált programtervezés** elveit követjük. A bonyolultság csökkentésének másik technikája a **moduláris tervezés**, amelynek során a teljes feladatot kisebb egységekre, úgynevezett modulokra bontjuk, és a modulokra írunk programokat.

A PHP és számos más programozási nyelv használatakor is van lehetőségünk a két technika ötvözésére. A feladatot modulokra bonthatjuk, és azokra külön-külön programokat írhatunk. A modulok azonban még így is bonyolultak lehetnek, így azok kódját főprogramra és alprogramokra bontva készítjük el.

7.3 A PHP BEÉPÍTETT FÜGGVÉNYEI

A PHP számos beépített, belső függvénnyel rendelkezik. A belső függvények száma olyan nagy, hogy a PHP dokumentációjában csupán a függvényeket csoportosító kategóriákból 85 található. Különálló függvénycsoport áll rendelkezésre az Apache webszerverrel kapcsolatos műveletekre, a fájlrendszer kezelésére, a hibakezelésre, a grafikák manipulálására, a HTTP protokollal való munkára, a tömbök kezelésére, a különböző adatbázisok elérésére… A teljes függvényreferencia az alábbi címen érhető el:

http://www.php.net/manual/en/funcref.php

10. link PHP függvényreferenciája

Az alábbiakban összefoglaljuk a programozásban legszükségesebb függvényeket.

7.3.1 A PHP konfigurációs beállításai

A PHP konfigurációs beállításait a **php.ini** direktíváiban találjuk, és természetesen itt is változtatjuk meg. A módosítások az összes PHP-alkalmazásra vonatkoznak, azonban csak az értelmező újraindítása után jutnak érvényre.

A PHP biztosítja a **php.ini** beállításainak **dinamikus** (futó programból történő) lekérdezését, és (korlátozások mellett, de) azok szabályozását is.

A direktívák lekérdezése az **ini_get()** függvénnyel végezhető el.

```
$max_upload=ini_get("upload_max_filesize");
```

A direktívákat felsoroló dokumentáció **Changeable** oszlopának **PHP_INI_ALL** és a **PHP_INI_USER** értéke az **ini_set()** függvénnyel, dinamikusan is megváltoztatható opciókat jelzi. Ezek a változtatások azonnal kifejtik hatásukat, de csak az adott program futása alatt érvényesek.

```
ini_set("direktíva", érték)
```

7.3.2 Változók kezelése

Programunkban gyakran van szükség egy **változó létezésének ellenőrzésére**. Az **isset()** függvény logikai igaz eredményt a vissza, ha egy változó létezik, és értéke nem NULL.

```
isset(változó [,változó...])
```

Ha a függvény több paraméterrel is rendelkezik, akkor csak akkor ad **true** eredményt, ha a fentiek minden paraméterre igazak.

Az **unset()** függvény **törli** a paraméterként megadott **változókat**.

```
unset(változó [,változó...])
```

A **változók típusa** a **get_type()** függvénnyel kérdezhető le, és a **set_type()** függvénnyel változtatható meg.

```
get_type(változó)
set_type(változó, típus)
```

7.3.3 Szövegkezelés

A PHP számos függvénnyel támogatja a szövegkezelő műveleteket. Talán leggyakrabban használt függvény az **strlen()**, amely a szöveg karakterszámát adja vissza.

```
strlen(szöveg)
```

Fontos tudni, hogy eredménye a **szöveg bájtokban megadott** hossza, ami nem feltétlenül azonos a karakterszámmal. A UNICODE kódolás esetén ugyanis egy karakter akár 4 bájt hosszúságú is lehet.

Az **mb_strlen()** függvény az adott kódolással tárolt szöveg karakterszámát adja meg.

```
mb_strlen(szöveg,kódolás)
```

```
$szoveg="árvíztűrőtükörfúrógép";
```

```
echo "Szöveghossz:",strlen($szoveg),"<br/>";  //30
```

```
echo "Karakter
szám:",mb_strlen($szoveg,"utf8"),"<br/>";//21
```

Szövegkeresés szintén gyakori feladat. Az **strpos()** függvény azt a karakterpozíciót adja vissza, ahol a szövegben egy szövegrész kezdődik.

```
strpos(szöveg,rész[,kezdet])
```

Ha megadjuk a 3. paramétert, akkor a keresés a kezdet sorszámú karaktertől kezdődik. A függvényt használhatjuk arra, hogy egy részszöveg előfordulási helyét megkeressük, de akár arra is, hogy ellenőrizzük a létezését a vizsgált karaktersorozatban.

```
if(strpos("Hello World!","W")>0){
```


```
echo "Van benne 'W'!";
```


```
}else{
```


```
echo "Nincs benne 'W'!";
```


```
}
```

Ha a szöveg egy részét akarjuk visszakapni, akkor **substr()** függvényt kell használnunk.

```
substr(szöveg, kezdet[,hossz])
```

A függvény által visszaadott részlet a szöveg kezdet karakterpozíciójában kezdődik, és *hossz* hosszúságú. Ha az utolsó paraméter nincs megadva, akkor a *kezdet* paraméter értékétől a szöveg végéig terjedő részt kapjuk vissza. A kezdet megadásakor az 1. karakter a 0 sorszámú.


```
echo substr ("Hello World!",0,5); //"Hello"
```

Gyakran van szükség arra, hogy egy elválasztó karakterekkel tagolt szöveg egyes részeit külön-külön kezeljük. Az **explode()** függvény a határolóval tagolt szöveget földarabolja és eredményként olyan tömböt ad vissza, aminek elemei a szöveg részei.

```
explode(határoló, szöveg )
```

7.3.4 Tömbkezelő függvények

A szövegekhez hasonlóan a tömbök esetében is a hossz meghatározása a leggyakoribb feladat.

A **count()** függvény a paraméterként megadott tömb elemeinek számát határozza meg. Ha a paraméter nem tömbváltozó, akkor az eredmény 1.

```
count(tömb)
```

Az **is_array()** függvény logikai eredménnyel jelzi, hogy a paraméterként kapott változó tömb típusú-e.

```
is_array(változó)
```

Ha azt kell megvizsgálnunk, hogy egy tömb elemei között szerepel-e egy megadott érték, akkor a logikai eredményt adó **in_str()** függvényt használhatjuk.

```
in_str(ellenőrzött_érték, tömb,[típusellenőrzés])
```

A függvény megvizsgáltja az összes tömbelemet, és igaz értékkel tér vissza, ha bármelyik értéke megegyezik az ellenőrzött értékkel. Ha a harmadik paraméter **true**, akkor az ellenőrzéskor a típust is megvizsgálja a PHP.

7.4 FÜGGVÉNYEK LÉTREHOZÁSA A PHP-BEN

A PHP programok megírásakor felhasználhatjuk a rendelkezésre álló beépített függvényeket, de természetesen saját alprogramokat is létrehozhatunk. Saját függvényt természetesen csak meghatározott szintaxis szerint, az alábbi formában hozhatunk létre:

```
function függvénynév ([formális_paraméterlista]){
  utasítások
  [return visszatérési_érték]
}
```

A függvényre a nevével lehet majd hivatkozni. A formális **paraméterlista nem kötelező**, csak akkor használjuk, ha a függvénynek adatokra van szüksége a működéshez. Valójában nem más, mint **változók felsorolása**. Az itt megadott változókat nevezzük **paraméterváltozóknak**, mert ezekbe kerülnek a függvény hívásakor átadott adatok, a paraméterértékek. A formális jelző arra utal, hogy a paraméterváltozók felsorolása meghatározza a hívás formátumát. Azt írja le, hogy a híváskor megadott **aktuális paraméterlistában hány paramétert kell átadni** a függvénynek.

Az utasítások a függvény feladatát megoldó programsorok.

A **return** utasítás – mint látjuk – opcionális. Akkor használjuk, ha visszatérési értéket akarunk átadni a hívás helyének.

Példaként lássunk egy egyszerű függvényt, amelynek feladata egy adott méretű kép pixelszámának meghatározása. A függvénynek a kép vízszintes és függőleges méretét meghatározó paraméterekre van szüksége, tehát formális paraméterlistájában két változót kell megadni.

Forrás: get_pixel.php

```
function get_pixel($pSor, $pOszlop){
  $pixel=$pSor*$pOszlop;
  echo $pixel;
}
```

A függvény két paramétert vár. Az eredményt nem adja vissza, hanem a kimenetre írja. Hívása így történhet a főprogramból:

```
//…
get_pixel(1024,768);
//…
```

Ha azt akarjuk, hogy az eredmény ne a kimenetre kerüljön, hanem a további feldolgozás érdekében a főprogram kapja vissza a pixelszámot, akkor **return** utasítást kell elhelyezni a függvényben, és változó, kifejezés, vagy literál formájában meg kell adni, hogy mi legyen visszaadott érték.

```
function get_pixel($pSor, $pOszlop){
  $pixel=$pSor*$pOszlop;
  return $pixel;
}
```

A **return** utasítás helyén **a függvény befejeződik**, és a vezérlés visszakerül a hívás helyére. Az visszatérési értéket felhasználhatjuk, például változóba tehetjük a hívás helyén.

Ha a **return** utasítás után nem szerepel semmilyen érték, a függvény futása akkor is megszakad, de az alprogram nem ad visszatérési értéket.

```
//…
$kep_thumb=get_pixel(102,75);
$kep_large=get_pixel(1024,768);
$rate=$kep_thumb/$kep_large;
//…
```

Itt a függvényt itt kétszer is meghívtuk, egyszer egy kisebb, egyszer pedig egy nagyobb képmérettel. A visszatérési értékeket egy-egy változóba tettük majd felhasználtuk. Meghatározzuk a két kép méretének arányát.

A visszatérési értéket visszaadó függvényt a legtöbb programozási nyelvben úgy kell meghívni, hogy a hívás helyén föl is használjuk a visszatérési értéket. Ez azt jelenti, hogy a függvényhívás lehet értékadás jobb oldalán (mint a példánkban), lehet kifejezés operandusa, vagy más alprogram paramétere.

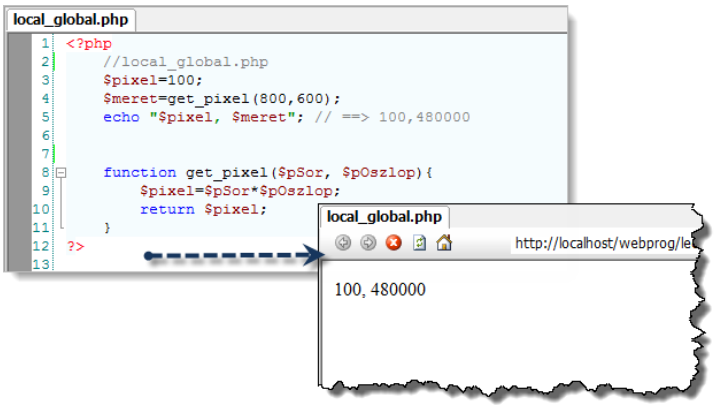
A PHP nem ilyen szigorú. Ha akarjuk, felhasználjuk a visszatérési értéket, de ha nem akarjuk, akkor nem kötelező.

7.5 VÁLTOZÓK ÉRVÉNYESSÉGI KÖRE

Az alprogramok (függvények) a főprogramhoz hasonlóan változókat használnak adataik tárolásához. Előfordulhat, hogy a főprogramban és egy, vagy több alprogramban azonos nevű változókat használunk, mégsem fordul elő, hogy egy alprogram átírja egy másik alprogram azonos nevű változójának értékét. A **főprogramban** létrehozott változókat **globális** az **alprogramokban** létrehozott változókat pedig **lokális változóknak** nevezzük. A függvények formális paraméterlistájában megadott **paraméterváltozók is lokális változók.**

A főprogram és az egyes alprogramok **saját memóriaterülettel** rendelkeznek, és változóikat ezeken a memóriaterületeken tárolják. A főprogram és a függvények is **csak a saját memóriaterületükön** tárolódó változókhoz férnek hozzá. Így semmiféle hibát nem okoz, ha egy globális és egy lokális, vagy két különböző függvény lokális változóinak neve megegyezik. Ezek különböző változók, csak a nevük azonos.

Forrás: local_global.php



51. ábra Lokális és globális változók

7.5.1 Előre definiált és szuperglobális változók

A PHP számos automatikusan létrehozott változót bocsát a programozó rendelkezésére. Ezek általában asszociatív tömbök, amelyek az értelmezőre, a webszerverre, a böngésző aktuális kérésére, munkamenetekre, cookie-kra stb. vonatkozó adatokat tartalmaznak. Ezeket a PHP-értelmező által előállított változókat nevezzük **előre definiált változóknak**.

Az előre definiált változók némelyike a programunk bármely pontján elérhető. Az ilyen előre definiált változókat nevezzük **szuperglobális** változóknak. Tananyagunkban már számos esetben használtunk ilyen változókat, hiszen **`$_GET`**, a **`$_POST`**, a **`$_SESSION`**, és a **`$_COOKIE`** változók mind előre definiáltak, és szuper globálisak. A PHP rendelkezésre bocsátja a **`$GLOBALS`** nevű szuper globális asszociatív tömböt is, ami a program összes pillanatnyilag elérhető globális változója számára tartalmaz egy elemet.

Az előre definiált változók teljes dokumentációját az alábbi címeken találja meg.

http://www.php.net/manual/en/reserved.variables.php

11. link Előre definiált változók dokumentációja

http://www.php.net/manual/en/language.variables.superglobals.php

12. link Szuper globális változók

7.5.2 A global kulcsszó

Mint említettük a főprogram és a függvények csak a paramétereken és a visszatérési értéken keresztül kommunikálnak. Előfordulhat, hogy egy függvénynek mégis szüksége van a főprogram valamilyen változóinak értékére.

Az ilyen problémák megoldására használható a **`global`** kulcsszó, ami a főprogramban deklarált globális változók lokális elérést teszi lehetővé.

Az alábbi példaprogram az előző feladat olyan megoldását mutatja be, ahol a függvény nem paramétereket vár, hanem globális változók értékét használja.

Forrás: global.php

global.php

```php
<?php
//global.php
    $sor=800;
    $oszlop=600;
    $meret=get_pixel();
    echo "$meret"; // ==> 480000

    function get_pixel(){
        global $sor,$oszlop;
        $pixel=$sor*$oszlop;
        return $pixel;
    }
?>
```

52. ábra Globális változók használata

A **`global`** kulcsszóval elért globális **változók nemcsak olvashatók, hanem írhatók, azaz meg is változtathatók a függvényekben**. Ez meglehetősen veszélyes, ezért csak akkor használjuk, ha mindenképpen szükség van rá.

Ha csak tehetjük, úgy írjuk meg a programunkat, hogy a függvények minden szükséges adatot paramétereken keresztül kapjanak.

A **`global`** kulcsszót általában a programunk konfigurációs adatainak átvételére használjuk a függvényekben.

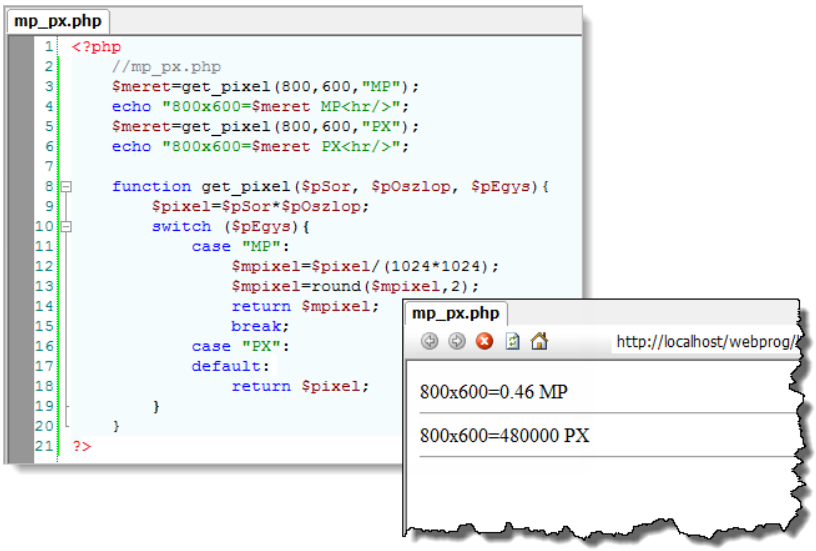
7.6 ALAPÉRTELMEZETT PARAMÉTERÉRTÉKEK

Minden függvény valamilyen részfeladatot old meg, és az ehhez szükséges adatokat paramétereken keresztül kapja. Kódját úgy célszerű megírni, hogy adott részfeladatot, a lehetőleg minél több körülmény figyelembevételével tudja elvégezni.

A képméretet kiszámoló függvényünk például meghatározza a kép pontjainak a számát, de mindig pixelben adja meg a visszatérési értéket. Kis előrelátással gondolhatunk arra, hogy lehet, hogy a főprogram egyes helyein képpontban lesz szükségünk egy kép méretére, míg máshol megapixelben megadott kerekített értéket akarunk használni.

*Az alábbi példa korrigálja a hibát. A **`get_pixel`** függvény, itt harmadik paraméterként egy rövidítést vár. Ha ennek értéke "MP", akkor megapixelben, különben pixelben adja vissza az eredményt.*

Forrás: mp_px.php

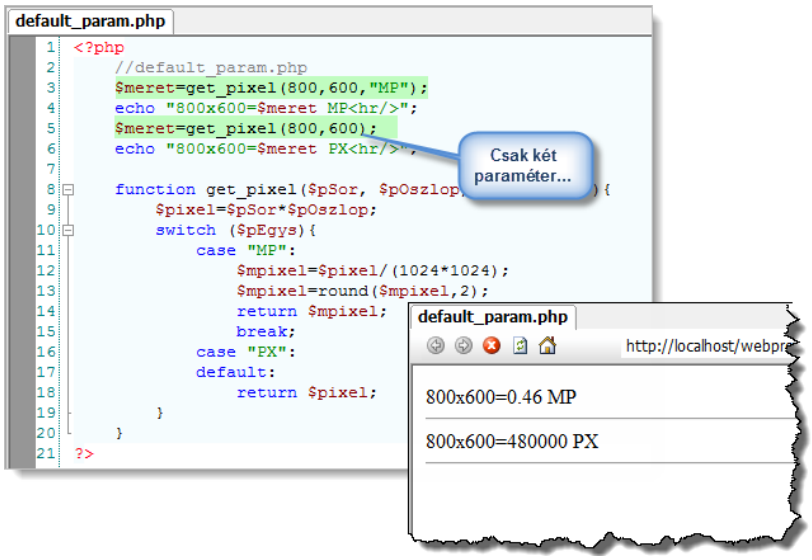


53. ábra Mértékegység figyelembevétele

Világos, hogy minél több részletet akarunk figyelembe venni a kiinduló állapot megadásakor, annál több paraméterre van szükség. Ez nem is baj, ha a paraméterek értékei szinte minden híváskor különbözőek. Ha azonban az esetek zömében ugyanazzal a paraméterrel dolgozunk, és csak néha van szükség más érték megadására, akkor egyszerűbb lenne, ha híváskor csak az általánostól eltérő esetben kellene megadnunk a paramétert. Ezt teszik lehetővé az úgynevezett **alapértelmezett paraméterértékek**. Ha a formális paraméterlistában egyszerű értékadással megadjuk egy paraméter alapértelmezett értékét, akkor híváskor az adott paraméter elhagyható, opcionális lesz. Ha értéke nem szerepel az aktuális paraméterlistában, akkor a paraméterváltozó automatikusan az alapértelmezett értéket kapja. Az alapértelmezett értékkel megadott paramétereket **opcionális paramétereknek** nevezzük.

*Az fenti példa az alábbi formában módosulhat. A függvény harmadik, **`$pEgys`** paraméterénél alapértelmezett értéket adtunk meg. Ha híváskor csak két aktuális paraméter van, akkor automatikusan "PX" lesz a **`$pEgys`** változó értéke.*

Forrás: default_param.php



54. ábra

> **Mivel a formális paraméterlista meghatározza a várt paraméterek számár és sorrendjét is, az opcionális paramétereket mindig a paraméterlista végére kell tenni. Az értelmező csak így tudja eldönteni, hogy melyik paramétert hagytuk el.**

7.7 ÖSSZETETT VISSZATÉRÉSI ÉRTÉK

Többször megállapítottuk, hogy a függvény kódja akkor jó, ha csak a paramétereken és a visszatérési értéken keresztül kommunikál a hívás helyével. A paraméterek kezelésével kapcsolatban megismerkedtünk néhány alternatív lehetőséggel, de az eredményt szolgáltató **`return`** parancs csak egyetlen értéket tud visszaadni. Vajon mi a teendő, ha a függvény nem egy, hanem több értéket adna vissza?

A probléma egyik megoldása az, hogy a visszatérési érték nemcsak egyszerű adat, hanem tömb, vagy objektum is lehet. A **PHP-ben** semmi sem tiltja, hogy egy függvény eredménye például numerikus, vagy akár asszociatív tömb legyen. Egy **függvény visszaadott értéke bármilyen típusú lehet**.

*Az alábbi programocska **`get_pixel()`** függvénye ismét módosult. Eredménye most nem egyetlen érték. hanem asszociatív tömb.*

A kép sor és oszlopszámát valamint a pixelben és megapixelben számolt méretét is visszaadja az eredménytömbben. A hívás helyén természetesen ennek megfelelően kell feldolgozni az eredményt.

Forrás: return_array.php

return_array.php

```php
<?php
    //return_array.php

    $kep=get_pixel(800,600);
    echo "800x600 =", $kep["px"],"PX<hr/>";
    echo "800x600 =", $kep["mp"],"MP<hr/>";

    function get_pixel($pSor, $pOszlop){
        $pixel=$pSor*$pOszlop;
        $mpixel=$pixel/(1024*1024);
        $mpixel=round($mpixel,2);
        $ret=array(
            "s"=>$pSor,
            "o"=>$pOszlop,
            "px"=>$pixel,
            "mp"=>$mpixel
        );
        return $ret;
    }
?>
```

return_array.php

http://localhost/webprog/

800x600 =480000PX

800x600 =0.46MP

55. ábra A visszatérési érték tömb

7.8 ÉRTÉK- ÉS CÍMPARAMÉTEREK

A fenti probléma másik megoldása az úgynevezett **címparaméter használata**. A függvények **paraméterei lokális változók**, amelyekbe a híváskor megadott **aktuális paraméterek értékeinek másolata** kerül. Ezért az ilyen paramétereket **érték paramétereknek** nevezzük.

A címparaméterek másképpen viselkednek. A formális paraméterlistában elfoglalt **helyükön kizárólag változó állhat a függvény hívásakor**! Az ilyen paraméterbe azonban nem a híváskor megadott változó értéke, hanem **memóriabeli címe** kerül. Így a függvényben csak a paraméterváltozó neve lesz lokális, a néven keresztül a hívás helyén lévő változó értékéhez lehet hozzáférni. A **hívás helyén** lévő **változónévhez** és a függvény **címparaméteréhez** ilyenkor **azonos**

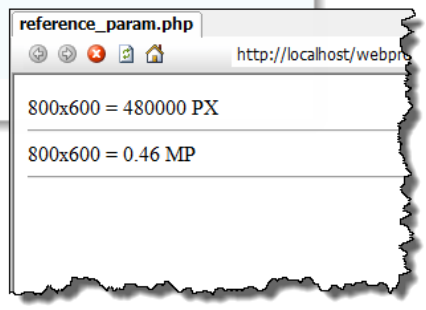
memóriaterület kapcsolódik. Így a függvény át tudja írni a híváskor megadott változó értékét. Ezzel a technikával tulajdonképpen értéket adhat vissza az alprogram. Mivel bármelyik paraméter lehet címparaméter, akár több eredményt is visszaadhatunk.

A címparaméterek létrehozása egyszerű, csupán a formális paraméterlistában nevük elé írt **&** karakterrel jelezzük eltérő működésüket.

Az alábbi példa a képméret számoló függvény egy alternatív változatát mutatja be. A függvény a hagyományos módon nem ad visszatérési értéket. Ezzel szemben négy paramétert is vár. Ezek közül a két utolsó, a ***$pix*** *és az* ***$mpix*** *címparaméterek. A függvény megváltoztatja a* ***$pix****, és az* ***$mpix*** *változók értékeit, így a főprogramban is új értéket kapnak a címparaméterek helyén álló* ***$px*** *és* ***$mp*** *változók.*

Forrás: reference_param.php

```
reference_param.php
1  <?php
2      //reference_param.php
3      $kep=get_pixel(800,600, $px,$mp);
4      echo "800x600 = $px PX<hr/>";
5      echo "800x600 = $mp MP<hr/>";
6
7      function get_pixel($pSor, $pOszlop, &$pix, &$mpix){
8          $pix=$pSor*$pOszlop;
9          $mpix=$pix/(1024*1024);
10         $mpix=round($mpix,2);
11     }
12 ?>
13
```



56. ábra Címparaméterek

7.9 VÁLTOZÓ SZÁMÚ PARAMÉTEREK

Programunk megírásakor szükségünk lehet olyan függvényre, aminek nem ismerjük előre a paraméterszámát. Az egyik alakalommal több, a másik híváskor kevesebb paramétert szándékozunk átadni neki.

Ilyen például az a feladat, amikor különböző számú, átadott számértékeket összegző függvényt szeretnénk készíteni.

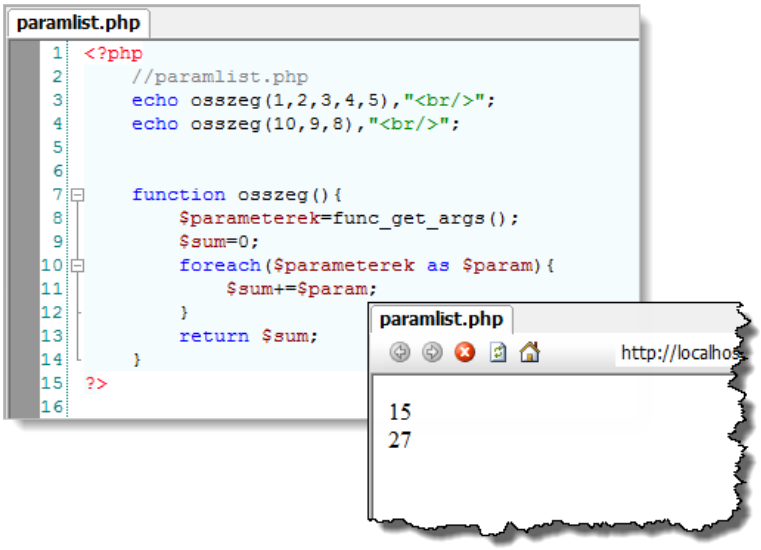
Ha deklaráláskor egyetlen változót sem adunk meg a formális paraméterlistában, akkor az elvben azt jelentené, hogy a függvény egyetlen paramétert sem fogad. A PHP formális paraméter nélküli függvényei ezzel szemben bármennyi (nulla, vagy több) paraméter fogadására alkalmasak.

Az ilyen függvényeken belül a PHP három beépített függvényével juthatunk hozzá az híváskor átadott paraméterekhez.

- A **`func_num_args()`** függvény a híváskor átadott paraméterek számát adja vissza.
- **`func_get_arg(index)`** a megadott indexű paraméter értékét kérdezetjük le.
- **`func_get_args()`** az összes paramétert egyetlen tömbbe teszi

A feladat megoldását az alábbi ábra mutatja.

Forrás: paramlist.php



57. ábra Változó számú paraméterek

7.10 STATIKUS VÁLTOZÓK

A függvények lokális változóit tároló memóriaterületet mindig a függvény hívásakor foglalja le az interpreter. Miután a függvény befejezte működését, ez a memóriaterület felszabadul, az abban tárolt változók értékei elvesznek. A következő híváskor új, üres lokális memóriaterületet kap a függvény. Ez persze nem is baj, hiszen a hívás helyén csak a függvény eredményére van szükség, ismételt híváskor pedig a paraméterekkel adjuk meg a függvény kiinduló állapotát.

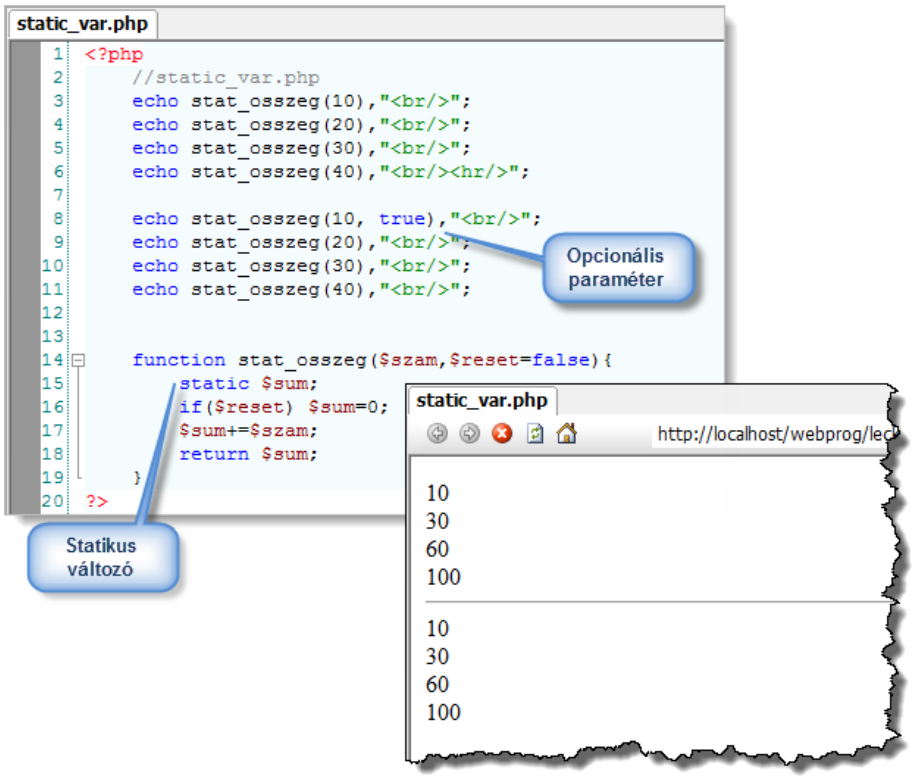
Vannak azonban speciális esetek, amikor szeretnénk, ha egy függvény megőrizné az előző állapot egy vagy több változóját.

> **Az értéküket a hívások között is megőrző változókat statikus változóknak nevezzük.**

Ilyen statikus változóra lehet szükség, ha olyan függvényt akarunk készíteni, ami összegzi az egyes hívások alkalmával átvett paraméter értékét.

Az alábbi példa **`stat_osszeg()`** *függvényben a* **`$sum`** *változó statikus, értéke megmarad a hívások között. A függvény az első paraméter (***`$sum`***) értékét minden híváskor hozzáadja a* **`$sum`** *változóhoz, majd kiírja annak aktuális értékét. A második* **`$reset`** *paraméter opcionális, alapértelmezett értéke* **`false`***. Ha híváskor* **`true`** *értékűre állítjuk, akkor a* **`$sum`** *változót nullázza a függvény.*

Forrás: static_var.php



58. ábra Statikus változó használata

7.11 ÖSSZEFOGLALÁS, KÉRDÉSEK

7.11.1 Összefoglalás

Tananyagunk 7. leckéje a PHP nyelv alprogramokkal kapcsolatos lehetőségeiről szólt. Elsőként az alprogramok szerepéről beszéltünk. Megállapítottuk, hogy az alprogramok a program bonyolultságának csökkentésére, strukturált program kialakítására használhatók.

A PHP-ben függvényeknek nevezzük és a **`function`** kulcsszóval deklaráljuk az alprogramokat. A függvények a program teljes feladatrendszerének egy-egy részfeladatát oldják meg, és csak az ahhoz kapcsolódó adatokon dolgoznak. A függvényt ezért zárt egységként célszerű megírni, ami csak paraméterein és visszatérési értékén keresztül kommunikál a környezetével.

A függvények deklarálásakor megadott formális paraméterlista tartalmazza az úgynevezett paraméterváltozókat. Ha a formális paraméterlista nem üres és nincsenek opcionális paraméterek sem, akkor a híváskor megadott aktuális paraméterek száma meg kell, hogy egyezzen a paraméterváltozók számával. Az opcionális paraméterek a híváskor elhagyhatók, ilyenkor alapértelmezett értéküket kapják. Üres formális paraméterlista esetén tetszőleges számú paramétert adhatunk át a függvénynek.

A függvényekben létrehozott változók, így a paraméterváltozók is lokálisak, más függvények számára nem elérhetők. A függvények alaphelyzetben nem férnek hozzá a főprogram globális változóihoz sem. A **`global`** kulcsszó használata esetén azonban elérhetővé válnak a kulcsszó után felsorolt főprogramban deklarált változók.

A PHP a környezeti adatok elérésére úgynevezett előre definiált változókat biztosít. Az előre definiált változók egy része a program bármely részén így a függvényekben is elérhető. Az ilyen előre definiált változókat szuperglobális változóknak nevezzük.

7.11.2 Önellenőrző kérdések

1. Hogy adhat vissza értéket egy függvény a hívás helyének?
   - A return paranccsal. Alternatív megoldás lehet címparaméterek használata. Végső soron a global kulcsszóval elérhetővé tett globális változók is írhatók, de ez a megoldás erősen ellenjavallt.
2. Hogyan kell meghívni az alábbi módon deklarált függvényt?
   function plusz($a,$b){ return $a+$b;}
   - A plusz függvénynek két paramétert kell átadni.
3. Hány paramétere lehet az alábbi függvénynek?
   function plusz(){//…}
   - Mivel a formális paraméterlista üres, a függvény tetszőleges számú aktuális paramétert képes fogadni.
4. Mik azok a szuper globális változók?
   - Olyan előre definiált változók, amelyek a program bármely pontján elérhetőek. Ilyen például a $_GET, a $_POST, vagy a $_SESSION is.
5. Mi a különbség cím- és értékparaméter között?

- Az értékparaméter a híváskor átadott érték másolatát, a cím paraméter a híváskor megadott változó memória-beli címét tartalmazza. Ha a címparaméter változó értékét átírjuk a függvényben, akkor a híváskor megadott változó értéke is változik. Az értékparaméter változó írása nem hat vissza a hívás helyére.

8. LECKE: MYSQL-ADATBÁZISOK KEZELÉSE PHP-BEN

8.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

Tananyagunk korábbi fejezeteiben tapasztaltuk, hogy az adatok kezelése és tárolása központi helyet tölt be a számítógép-programozásban, hiszen program, futása során adatokkal leírható állapotokon keresztül jut el a teljes feladat megoldásához. Az adatok a program működése alatt folyamatos feldolgozás alatt állnak, ezért tárolásukra és kezelésükre változókat használunk. Az egyszerű számokat, szövegeket, vagy logikai értékeket tároló változók mellett, az összetett adatszerkezetek kezelésére is van mód. A numerikus és asszociatív tömbök együttes használatával egészen bonyolult struktúrák kialakítására és feldolgozására nyílik lehetőség.

A PHP különböző típusú változói kiváló lehetőségeket biztosítanak az adatok futásidő alatti tárolására és kezelésére. Az, hogy értékeik a memóriában tárolódnak, biztosítja a gyors hozzáférést és feldolgozást, de egyben azt is jelenti, hogy a változókban tárolt adatok csak a program futásának befejeződéséig maradnak meg. Amikor a szkript véget ér, a program által lefoglalt memóriaterület fölszabadul, és a változók értéke elvész. A tanult technikákkal ugyan át tudjuk menteni adatainkat az állapotok között, de sem a munkamentek, se a cookie-k, sem pedig a rejtett INPUT-mezők nem biztosítják az adatok perzisztens tárolását.

A legtöbb webalkalmazás működése ugyanakkor éppen a tartósan, és biztonságosan tárolt, bármikor, akár több felhasználó által párhuzamosan is hozzáférhető és manipulálható adatokon alapszik. Ezeket a funkciókat csak speciális, kifejezetten az adatkezelésre kifejlesztett szoftverek, az úgynevezett adatbáziskezelő rendszerek tudják biztosítani.

A mai tananyagban arra a problémára keresünk megoldást, hogy hogyan kapcsolódhatunk a négyrétegű kliens-szerver architektúrában működő webalkalmazásokkal a negyedik réteg feladatait ellátó adatbázis-kezelő rendszerekhez. A most következő leckében a PHP MySQL-adatbázisok kezelésével kapcsolatos lehetőségeibe nyerhet betekintést.

Megtanulhatja a MySQL szerver és a PHP alkalmazás közötti kapcsolat fölépítésének, az alkalmazás adatbázisa kiválasztásának módját, elsajátíthatja a MySQL-szerver SQL-mondatok segítségével történő vezérlését, a szerver által

küldött adatok feldolgozását, és megismerheti a folyamat során bekövetkező hibák kezelésének alapvető technikáját.

A lecke tananyagának elsajátítása után le tudja kérdezni és meg tudja jeleníteni az adatbázisban tárolt adatokat, de képes lesz új rekordok rögzítésére, és a tárolt értékek megváltoztatására is.

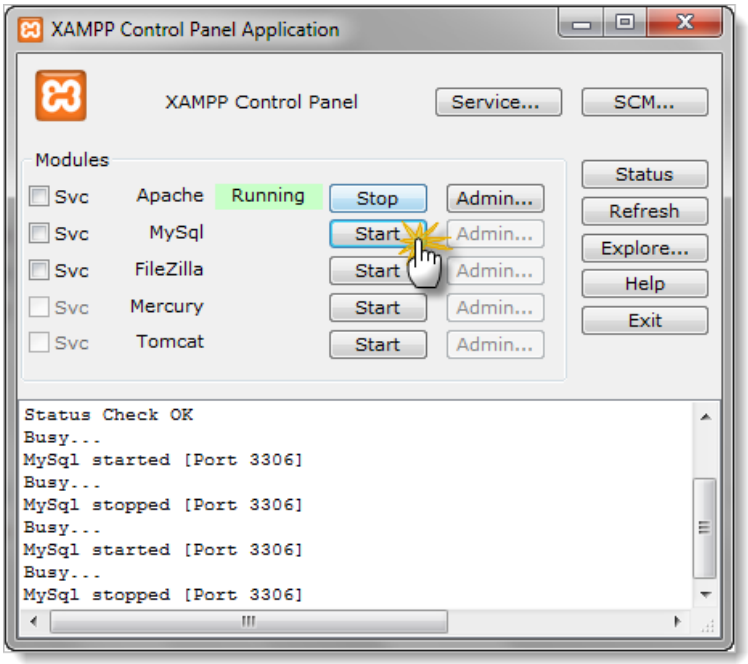
8.2 SZOFTVERKÖRNYEZET

Az adatbázisok webes kezeléséhez számos járulékos feladat kapcsolódik. Az összes kapcsolódó részfeladat bemutatása és megoldása elterelné a figyelmet az eredeti tananyagról, ezért a leckében kifejezetten a PHP-MySQL kapcsolatra koncentrálunk, a többi tevékenységre a csak utalásokat teszünk. Nem foglalkozunk például az adatbázis-tervezés lépéseivel, hanem kész adatbázist veszünk használatba. Egészen egyszerű felhasználói felülettel dolgozunk, csak utalunk a felhasználótól származó adatok ellenőrzésének szükségességére. A bemutatott szkripteket nem szervezzük működő webes alkalmazássá, a példák külön-külön mutatják be az adatbázis-kezelés egy-egy résztevékenységét

A lecke az eddigiekhez hasonlóan számos példaprogramot mutat be, amelyek mind megtalálhatók a **lecke8** mappájában. Kipróbálásukhoz azonban már nem lesz elegendő a webszerver és a PHP-értelmező. Szükségünk lesz működő MySQL-szerverre, fizikailag kialakított adatbázisra, és megfelelő jogosultságokkal rendelkező MySQL-felhasználóra is.

8.2.1 MySQL-szerver

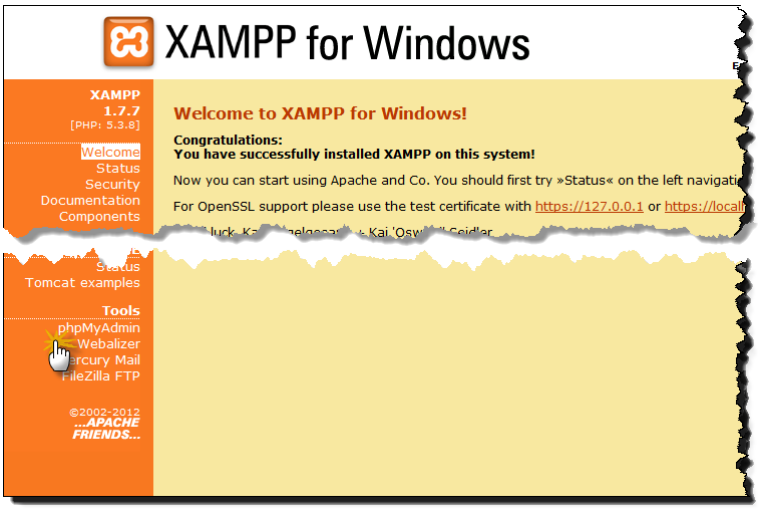
A MySQL-szerver üzembe helyezése nem jelent semmiféle nehézséget, az hiszen a tananyagunk elején telepített XAMPP alkalmazáscsomag tartalmazza az adatbázis-kezelő rendszert is. A szerver elindítását az XAMPP **Control Panelj**én egyetlen kattintással elvégezhetjük.



59. ábra MySQL-szerver indítása

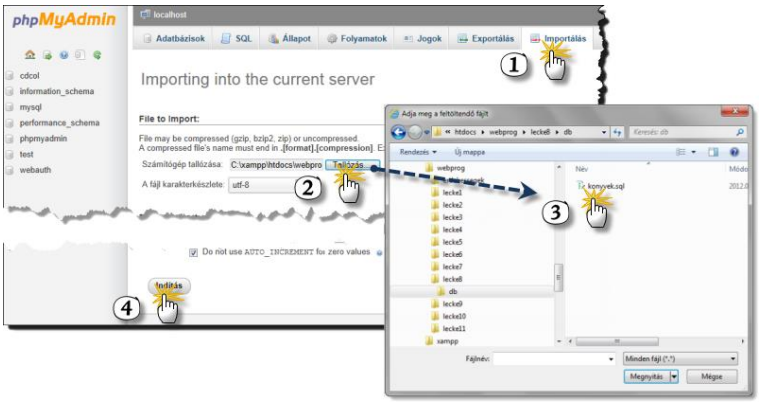
Miután a szerver elindult, létre kell hoznunk az adatbázist és a hozzáféréshez szükséges felhasználói fiókot. A telepítés után saját gépünkről jelszó megadása nélkül, **`root`** felhasználóként férhetünk hozzá az XAMPP csomag MySQL-szerveréhez, így csupán a megfelelő kliensprogramra van szükség az adatbázis kialakításához, és a felhasználó létrehozásához. Az XAMPP szerencsére ebben is segít. A csomag részét képezi a web felületen elérhető **phpMyAdmin** nevű alkalmazás, amely lehetővé teszi a szerver teljeskörű felügyeletét.

A **`lecke8/db`** alkönyvtárában találkató **`konyvek.sql`** állomány importálásával a MySQL szerveren létre tudjuk hozni a lecke példáiban használt adatbázist, és az eléréshez szükséges felhasználói fiókot is.



60. ábra http://localhost

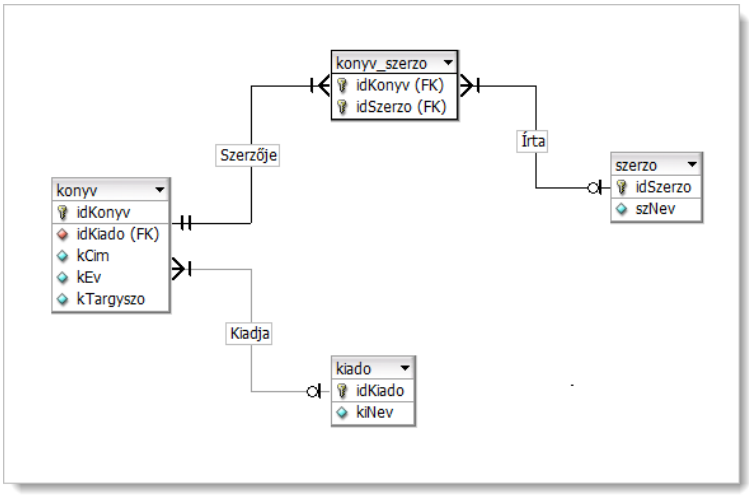
- Böngészőnkkel indítsuk el az XAMPP webes kezelőfelületét, és a kattintsunk a **Tools** menü **phpMyAdmin** parancsára.



61. ábra konyvek.sql importálása

- A **phpMyAdmin** felületén válasszuk az **Importálás** gombot, importáljuk, majd indítsuk el a **lecke8/db/konyvek.sql** állományt. Az importálás után a rendelkezésünkre áll a **konyvek** nevű adatbázis, amelyet a localhost-ról a **webprog/guttenberg** accounttal érünk el.

Az adatbázis a korábban tömbökben elhelyezett könyvek tárolását és kezelését teszi lehetővé.



62. ábra Az adatbázis szerkezete

A könyvek címe, kiadási éve, és tárgyszavai a **konyv** táblában találhatók, amelyet idegen kulcs kapcsol össze a **kiado** táblával. A **konyv** és **szerzo** táblák közötti N:M kapcsolatot a **konyv_szerzo** tábla írja le. Minden tábla tranzakcióbiztos, InnoDB típusú. A táblák üresek, egyedül a **kiado** tábla tartalmaz pár rekordot.

A redundancia csökkentése érdekében az adatokat több táblában helyeztük el, így egy kiadót csak egyszer tárolunk, akkor is, ha sok könyvet jelentettek meg, és a szerzőket sem kell többször rögzítenünk csak azért, mert több könyvet írtak. A redundancia csökkentésének természetesen megvan az ára. Amikor meg akarjuk nézni egy könyv adatait akkor a négy tábla kapcsolatát kell kezelnünk.

Ha új könyvet akarunk rögzíteni, akkor előfordulhat, hogy a **szerzo** és **kiado** táblába is új rekordot kell tennünk, de a **kony_szerzo** relációba mindenképpen új sor kerül.

8.3 MYSQL API

A MySQL adatbázisokkal való kapcsolat kiépítésére számos csatolófelületet, úgynevezett connector-t és API-t (Application Programming Interface) fejlesztettek ki. PHP-környezetben két ilyen programozói interfész áll rendelkezésre. A PHP a korábban kifejlesztett **mysql** kiterjesztése (MySQL Extension) függvénykönyvtár formájában áll rendelkezésre. Ha ezt az API-t használjuk,

PHP-függvényhívások segítségével vezérelhetjük MySQL adatbázis-kezelő rendszert.

A MySQL 4.1 és későbbi szerverek funkciónak használatához az eredeti API-t továbbfejlesztették. A PHP **mysqli** kiterjesztése (MySQL Improved Extension) a továbbfejlesztett funkciókon túl objektumorientált programozási környezetet is biztosít. Tananyagunkban függvényhívásokon keresztül bonyolítjuk le az adatbázis elérést, tehát az első API-változatot használjuk.

8.4 ADATBÁZIS-MŰVELETEK LÉPÉSEI

A PHP API segítségével gyakorlatilag bármilyen műveletet el tudunk végezni a MySQL adatbázisokkal, de akár DDL-, DML-, DCL- vagy DQL-mondatokat küldünk a szervernek lépések több-kevéssé azonosak:

- Föl kell építeni a PHP-szkript és a MySQL-szerver kapcsolatát.
- Ki kell választani a szükséges adatbázist.
- Össze kell állítani a MySQL-nek küldendő SQL-mondatot.
- Az utasítást el kell küldeni a MySQL-nek.
- Fogadni kell a szerver válaszát.
- Föl kell dolgozni az eredményt.
- Ha szükséges, meg kell jeleníteni a felhasználó felületén az adatokat.
- Bontani kell a PHP-MySQL kapcsolatot.

Elsőként DQL-mondatokat hajtunk végre, tehát választó lekérdezéseket futtatunk. Példánkban a **`kiado`** tábla rekordjainak lekérdezésén és a kiadók megjelenítésén keresztül tekintjük át a fenti lépéseket.

8.4.1 Kapcsolat fölépítése

A PHP-MySQL kapcsolat fölépítését a **`mysql_connect()`** függvénnyel végezhetjük el.

```
erőforrás mysql_connect(szerver,user,jelszó);
```

A függvény három paramétert, az adatbázis szerver címét, a felhasználó nevét, és jelszavát várja. Ha a kapcsolódás sikeres, akkor a visszatérési érték, az úgynevezett kapcsolatazonosító. A **kapcsolatazonosító** a PHP speciális **erőforrás típusának** megfelelő adat. A PHP-szkript befejeződéséig, vagy a **`mysql_close()`** függvény meghívásáig a többi MySQL függvényben ezzel a

kapcsolatazonosítóval hivatkozhatunk a kapcsolatra. Ha a kapcsolat sikertelen, a **mysql_connect()** függvény **false** (logikai hamis) értéket ad vissza.

Az alábbi kód, a localhoston futó MySQL szerverrel épít föl kapcsolatot, amelyben a **webprog** felhasználói nevet, és a **guttenberg** jelszót használja.

```
$host="localhost";
$user="webprog";
$passwd="guttenberg";
$conn=mysql_connect($host,$user,$passwd)
```

Egy PHP szkritpből elvileg több MySQL-kapcsolat is megnyitható, ezért minden további függvényben megadható, hogy a művelet melyik kapcsolatra vonatkozik. Az esetek többségében egyetlen kapcsolattal dolgozunk, ezért a kapcsolatazonosító általában elhagyható.

Látható, hogy a PHP-szkriptben tárolódik a MySQL-kapcsolathoz szükséges felhasználói név, és jelszó. Ez bizony potenciális veszélyforrást jelent. Feltétlenül el kell kerülni, hogy a szkript forrása illetéktelenek kezébe kerüljön!

8.4.2 Hibakezelés

Mivel egyáltalán nem biztos, hogy a kapcsolódás sikeres lesz, a hibára is föl kell készülnünk. A hibás kapcsolódási kísérlet esetén figyelmezető üzenet kerül a kimenetre, de a szkript futása nem szakad meg.

A hibakezelés legegyszerűbb változata, a saját hibaüzenet küldése és a szkript futásának felfüggesztése. Mindkét művelet egyszerre a **die()** függvénnyel végezhető el. A **die()** a kimenetre írja a paraméterként megadott szöveget, majd leállítja a szkripet.

A kapcsolat hibáját úgy figyelhetjük, hogy **mysql_connect()** függvény hívását egy elágazás logikai kifejezéseként adjuk meg, és a szkriptet leállító **die()** függvényt az igaz ágba tesszük. Mivel hiba esetén a visszatérési érték **false**, a kifejezés hiba esetén akkor válik igazzá, ha a kapcsolatazonosító elé a tagadás jelét tesszük.

```
if (!$conn=mysql_connect($host,$user,$passwd)) die("A
kapcsolódás sikertelen");
```

A MySQL API függvényhívásaikor bekövetkező hibákat mindig ebben a formában figyeljük. A felhasználó tájékoztatása, és a hiba utáni továbblépés

lehetőségének biztosítására többféle lehetőség kínálkozik. A további mysql függvényhívások esetén az egyszerűség érdekében a **die()** függvénnyel megvalósított hibakezelést használjuk.

A mysql_connect() függvény fenti három paraméterének alapértelmezett értékei rögzíthetők a php.ini konfigurációs állományban. Ha ezeket akarjuk használni, a paraméterek akár el is hagyhatók. A függvény még két további opcionális paramétert is kaphat, amelyek a többszörös kapcsolatot illetve a kapcsolat egyéb jellemzőit szabályozzák.

8.4.3 Kapcsolat lezárása

A kapcsolat lezárása a **mysql_close(***kapcsolat_id***)** függvénnyel történik. Hívásakor a **mysql_connect()** függvénytől kapott kapcsolatazonosítóval jelezhetjük, melyik kapcsolatot akarjuk bezárni. Mivel általában csak egy kapcsolat van nyitva a **mysql_close()** paraméter nélkül is használható.

A szkript befejeződésekor mindenképpen megszakad a PHP-MySQL kapcsolat, de ettől függetlenül illik explicit módon meghívni a **mysql_close()** függvényt.

8.4.4 Adatbázis kiválasztása:

Miután sikeresen kapcsolódtunk a MySQL szerverhez, ki kell választanunk a webalkalmazás adatait tároló adatbázist.

Ezt a feladatot a **mysql_select_db()** függvény végzi el.

```
logikai mysql_select_db(adatbázis,[kapcsolat_id])
```

A függvény első paramétere az adatbázis neve, a második, elhagyható paraméter pedig a kapcsolat azonosítója. A visszatérési érték igaz, ha művelet sikeres, és hamis, ha sikertelen.

Forrás: mysql_connect.php

mysql_connect.php

```php
<?php
    //mysql_connect.php
    header("Content-type: text/html; charset=utf-8");
    $host="localhost";
    $user="webprog";
    $passwd="guttenberg";
    $database="konyvek";
    //Kapcsolat fölépítése
    if (!$conn=mysql_connect($host,$user,$passwd)) die("A kapcsolódás sikertelen");
    //Ide már csak akkor jutunk, ha sikeres a művelet
    echo "A kapcsolat létrejött a $host szerverrel!<br/>";
    //Adatbázis kiválasztása
    if (!$ok=mysql_select_db($database)) die ("A $database adatbázis nem elérhető, vagy nem létezik!");
    echo "A $database adatbázis kiválasztása sikeres.<br/>";
    mysql_close();
    echo "A $conn kapcsolat lezárult.<br/>";
?>
```

63. ábra mysql_select_db()

8.4.5 SQL-mondat előkészítése

A következő feladat a SQL-mondat előállítása. Ez gyakorlatilag egy szöveg összeállítását jelenti, ami nem túlságosan nehéz feladat. Akkor válik érdekessé, ha az SQL-mondat valamelyik elemét egy változóból, esetleg a kliens oldalról érkezett adat alapján készítjük el.

> **A felhasználótól érkezett adatokat feltétlenül ellenőrizni kell, mielőtt SQL-mondatba illesztenénk őket.**

A feltöltött adatok közvetlen SQL-mondatba illesztése ad lehetőséget az **SQL injection** típusú támadásra. Használatával a támadó jogosulatlanul is hozzáférhet bizonyos adatokhoz.

Tegyük föl, hogy csak a 3-as azonosítójú kiadó adatait szeretnénk megmutatni a felhasználónak. Az SQL mondat így fog szólni:

```
$id=3;
$sql="SELECT * FROM kiado WHERE idKiado=$id";
```

Ha a fenti módon készült SQL-mondatot küldjük el az SQL-szervernek (rögtön látni fogjuk, hogy hogyan), akkor a **kiado** tábla egyetlen rekordját kapjuk vissza. Tegyük fel, hogy az azonosító értékét a felhasználói felületről GET metódusú kérés URL-címében kapjuk:

http://localhost/webprog/lecke8/injection.php?id=3

Ebben az esetben a fenti kódrészlet így módosul:

```
$id=$_GET["id"];
$sql="SELECT * FROM kiado WHERE idKiado=$id";
```

Az így létrehozott **$sql** változó értéke az alábbi SQL-mondat lesz:

```
SELECT * FROM kiado WHERE idKiado=3
```

Ha felhasználó a 3-as értéket töltötte föl, akkor minden rendben. A támadó azonban észreveszi, a böngésző címmezőjében az **?id=3** paramétert, és a címmező értékét egyszerűen átírja:

http://localhost/webprog/lecke8/injection.php?id=3+or+1%3d1

A fenti kódrészlet eredmény **$sql** változójának értéke most ez lesz:

```
SELECT * FROM kiado WHERE idKiado=3 or 1=1
```

Mivel a WHERE utáni logikai kifejezés így mindenképpen igaz, a MySQL az összes rekordot visszaadja a **kiado** táblából, a felhasználó a tábla teljes tartalmához hozzájuthat. Elég azonban az alábbi kódban látható apró változtatás, és a fenti trükköt kivédtük.

```
$id=(integer) $_GET["id"];
$sql="SELECT * FROM kiado WHERE idKiado=$id";
```

Forrás: injection.php; injection_safe.php

injection_safe.php

```
1  <?php
2      //injection_safe.php
3      if (isset($_GET["id"])){
4          //Az id-t egésszé konvertáljuk
5          $id= (integer) $_GET["id"];
6          echo "<h3>SELECT * FROM kiado WHERE idKiado=$id</h3>";
7      }
8      echo "<a href='injection_safe.php?id=3'>id=3</a><br/>";
9      echo "<a href='injection_safe.php?id=3+or+1%3d1'>id=3+or+1%3d1</a><br/>";
10 ?>
```

64. ábra SQL injection kivédése

A fenti példa csak az SQL injection egy elleni védekezés egy egyszerű változatát mutatja be. Nem is ellenőrzést, hanem direkt átalakítást végezve megakadályozza, hogy az **$id** változóba szöveg kerüljön. A valóban biztonságos védelemhez alaposabb ellenőrzés szükséges, ami nem csak a típusra, de az értékekre is kiterjed.

8.4.6 SQL mondat elküldése

Az előkészített SQL mondatot el kell juttatni a MySQL-szerverhez, ami majd értelmezi az utasítást, és a kliens által megadatott karakterkódolású választ küld a kérésre. Az alapértelmezett karakterkódolást a **php.ini** szabályozza. Ha ettől eltérő értéket akarunk használni, akkor a **mysql_set_charset()** függvénnyel állíthatjuk be, hogy milyen kódolást alkalmazzon a MySQL szerver.

```
  logikai mysql_set_charset(karakterkódolás);
if(!$ok=mysql_set_charset("utf8")) die („Nem létező
kódolás”);
```

SQL mondatot a **mysql_query()** függvénnyel küldhetjük el a DBMS-nek.

```
erőforrás mysql_query (sql_mondat [,kapcsolat_id] );
```

A függvény első paramétere az SQL mondat, a második az elhagyható kapcsolatazonosító. Hibamentes végrehajtás esetén erőforrás azonosítót, hiba esetén pedig **false** értéket kapunk visszatérési értékként.

```
if (!$res=mysql_query($sql)) die("A lekérdezés nem
sikerült.");
```

8.4.7 Eredmény feldolgozása

A lekérdezés eredményhalmaza nem áll közvetlenül rendelkezésre az SQL-mondat elküldése után. A rekordokhoz a sikeres lekérdezéssel visszakapott erőforrás azonosító birtokában erre a célra alkalmas függvényekkel segítségével juthatunk hozzá.

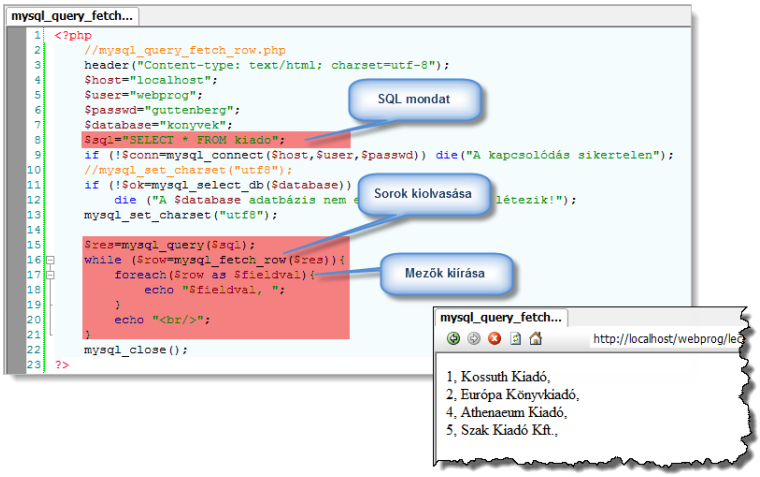
A **mysql_fetch_row()** függvény **egy rekordot** olvas ki az eredményhalmazból, és numerikus tömb formájában adja vissza. A függvény paramétere a **mysql_query()**-vel kapott erőforrás azonosító. Az eredménytömb elemei a kiolvasott rekord mezőértékei lesznek.

```
$res=mysql_query($sql);
$row=mysql_fetch_row($res);
foreach($row as $fieldval){
   echo "$fieldval, ";
}
```

A fenti szkript egyetlen rekord mezőértékeit teszi a ***$row*** *tömbbe, majd* ***foreach*** *ciklussal kiírja a tömb elemeit.*

A **mysql_fetch_row()** csak egy sort ad vissza, pedig egy eredményhalmaz általában nem egyetlen rekordot tartalmaz. PHP egy belső mutató segítségével tárolja, hogy melyik volt az utolsó kiolvasott rekord, és ha a **mysql_fetch_row()** függvényt ismét meghívjuk, akkor már a következő sort kapjuk majd vissza az eredményhalmazból. Ha az utolsó rekordot is kiolvastuk, azaz nincs több rekord, akkor a függvény **false** értékkel tér vissza. Ha a függvényhívást ciklusba tesszük, akkor az összes rekord összes mezőjét kiolvashatjuk.

Forrás: mysql_query_fetch_row.php



65. ábra Rekordok kiolvasása

A rekordok kinyerésének másik technikája a **mysql_fetch_assoc()** függvény használata. Ez hasonlóan működik, mint **a mysql_fetch_row()** de a kiolvasott rekordot asszociatív tömbbe teszi. Az elemek kulcsai a mezőneveknek, értékeik a mezőértékeknek felelnek meg.

Forrás: mysql_query_fetch_assoc.php

66. ábra mysql_fetch_assoc() használata

8.4.8 Információk az eredményhalmazról:

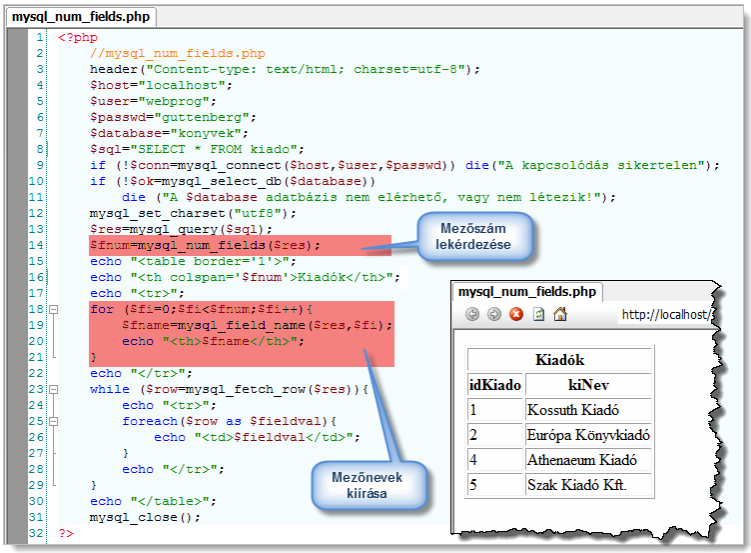
Amikor egy SQL mondatot elküldünk a MySQL szervernek, az nem csak az eredményhalmazt, hanem annak metaadatait is vissza tudja küldeni.

A **`mysql_num_rows(`***erőforrás_id***`)`** függvény a **`mysql_query()`**-vel végrehajtott lekérdezés rekordszámát adja vissza.

A **`mysql_num_fields(`***erőforrás_id***`)`** visszatérési értéke az eredményhalmaz mezőszáma. Ez akkor használható jól, ha meg szeretnénk tudni az egyes mezőneveket.

A **`mysql_field_name(`***erőforrás_id*`,`*mező_sorszám***`)`** a megadott eredményhalmaz meghatározott sorszámú mezőnevével tér vissza.

Forrás: mysql_num_fields.php



67. ábra Mezőnevek kiírása

8.4.9 Erőforrás fölszabadítása

A szkript befejezésekor a PHP fölszabadítja az összes erőforrást, azonban hatékony memóriahasználat érdekében helyes, ha a programunk futása közben magunk végezzük el a feladatot. Erre a **`mysql_free_result()`** függvényt használhatjuk.

```
mysql_free_result(erőforrás)
```

8.5 DML UPDATE, DELETE

Miután képesek vagyunk DQL-mondatok végrehatására, itt az ideje UPDATE, DELETE vagy INSERT parancsokat tartalmazó DML-mondatokat küldjünk a MySQL-szervernek. A DML (Data Manipulation Language) mondatok futtatása pontosan úgy történik, mint a DQL-lekérdezés végrehajtása. A különbség mindössze az, hogy mivel DML-mondatoknak nincs eredményrelációjuk, a **`mysql_query()`** függvény sikeres futás esetén nem erőforrás azonosítót, hanem **`true`** értéket kapunk visszatérési értékként. A hibát változatlanul **`false`** eredmény jelzi.

A DML-mondatok futtatásakor nem lehet sorokat kiolvasni az erőforrásból! A mysql_fetch_row(), mysql_fetch_field(), mysql_fetch_assoc() függvények használata hibát eredményez!

A következő példákban a DML-mondatok végrehajtása előtt és után is kiírjuk az érintett tábla tartalmát. Hogy ne kelljen minden esetben leírni a tábla rekordjait lekérdező és a formázott kimenetet előállító kódot leírni, ezeket a programsorokat alprogramba helyeztük (write_table), és a főprogram megfelelő pontjain meghívjuk a függvényt.

A lényeg kiemelése érdekében a kapcsolatot kialakító sorok is függvénybe kerültek (create_conn)

```php
<?php
    //mysql_query_update.php
    header("Content-type: text/html; charset=utf-8");
    $sql="SELECT * FROM kiado";

    function create_conn(){
        $host="localhost";
        $user="webprog";
        $passwd="guttenberg";
        $database="konyvek";

        if (!$conn=mysql_connect($host,$user,$passwd)) die("A kapcsolódás sikertelen");
        if (!$ok=mysql_select_db($database))
            die ("A $database adatbázis nem elérhető, vagy nem létezik!");
        mysql_set_charset("utf8");
    }

    function write_table($res, $tbl="Tábla"){
        echo "<table border='1'>";
        echo "<th colspan='2'>$tbl</th>";
        $fnum=mysql_num_fields($res);
        echo "<tr>";
        for ($fi=0;$fi<$fnum;$fi++){
            $fname=mysql_field_name($res,$fi);
            echo "<th>$fname</th>";
        }
        echo "</tr>";
        while ($row=mysql_fetch_row($res)){
            echo "<tr>";
            foreach($row as $fieldval){
                echo "<td>$fieldval</td>";
            }
            echo "</tr>";
        }
        echo "</table><hr/>";
    }
?>
```

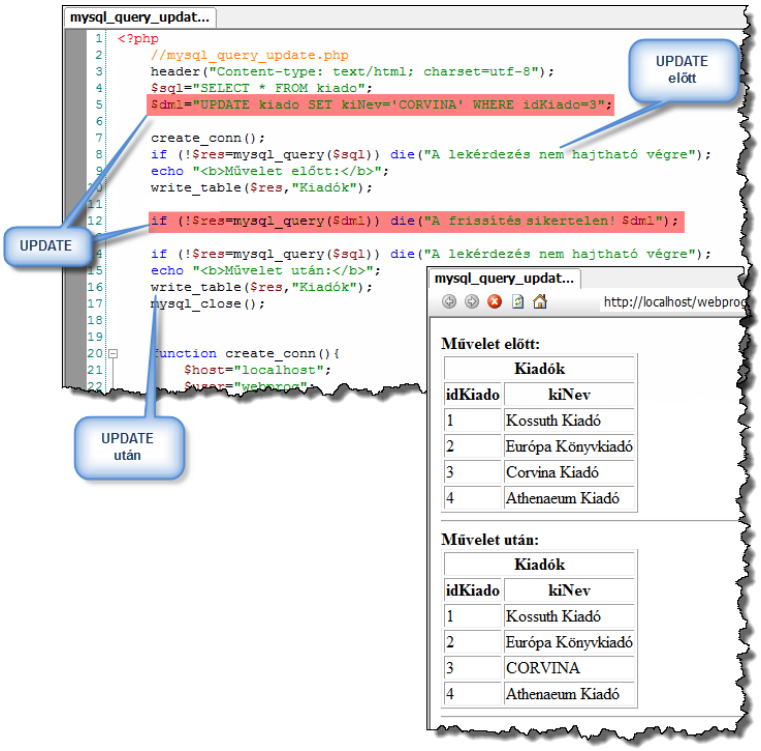
Kapcsolat létrehozása

Adott erőforráshoz tartozó eredményhalmaz lekérdezése, és megjelenítése

68. ábra Függvények

A következő ábrán a fenti függvényeket is használó kódot látjuk.

Forrás: mysql_query_update.php



69. ábra DML-mondat futtatása

8.6 DML INSERT

Új rekord beszúrása egy táblába legalább olyan egyszerű, mint a rekord módosítása. A végrehajtásban sincs nagy különbség, csupán az SQL-mondat lesz más. Mégis azt kell mondanunk, hogy a jól megtervezett adatbázisban az új rekordok fölvétele okozza a legtöbb gondot. Lássuk csak miért!

A redundancia kiküszöbölése érdekében az adatbázis tervezésekor olyan adatszerkezetet alakítunk ki, amely részleges és a tranzitív függésektől mentes. Mivel a káros függések megszűntetésekor dekompozíciót alkalmazunk, még a kis adatbázisok is több táblákra bomlanak. A táblák közötti kapcsolatokat idegen kulcsok írják le.

A felhasználó ebből persze semmit sem lát. Amikor megadja egy könyv adatait, nem érdekli őt, hogy a kiadók, és a szerzők külön táblában tárolódnak és arra is fittyet hány, hogy a **`konyv`** és **`szerzo`** tábla N:M kapcsolat miatt kapcsolótáblára is szükség volt. Egy könyv összes adatát egyetlen űrlapon akarja begépelni. A programozónak természetesen olyan kezelőfelületet kell előállíta-

nia, amelyen a felhasználó kényelmesen dolgozhat, de a bejövő adatokat (megfelelő ellenőrzés után) a különböző táblákban kell tárolnia, és az idegen kulcsok helyességéről is gondoskodnia kell.

Következő példánkban egy könyv adatait rögzítjük. Az egyszerűség kedvéért induljunk ki abból, hogy a bevitel megtörtént, az adatok eljutottak a szerverre, és a validáláson is túlvagyunk. A rögzítendő adatokat a jobb áttekinthetőség kedvéért egy asszociatív tömbbe rendeztük:

```
  $insert=array(
    "kiado"=>array("kiNev"=>"Szak Kiadó Kft."),
    "szerzo"=>array("szNev"=>"Young, Michael J."),
    "konyv"=>array(
              "kCim"=>"XML lépésről lépésre",
              "kEv"=>2002,
              "kTargyszavak"=>"informatika,XML, SGML,
XPATH"
            )
  );
```

A "kiado", "szerzo" és "konyv" elemekben vannak az egyes táblák új rekordjainak mezőnevei és értékeik. Az azonosítók mindenhonnan hiányoznak, ugyanis a táblában automatikus sorszámozású (AUTO_INCREMENT) mezők lesznek az azonosítók.

A fenti tömb csak az adatok átláthatóságát szolgálja, a példaprogramokban nem szerepel.

A feladatot a **kiado** táblával kell kezdenünk. Az új kiadó rekordjának beszúrása nem fog nehézséget okozni, azonban nem szabad megfeledkezni arról, hogy szükségünk lesz az újonnan beszúrt rekord azonosítójára, hiszen ez idegen kulcsként fog szerepelni a **konyv** táblában.

A kiadó után fölvehetjük az új könyvet, majd a szerzőt, de ismét meg kell jegyeznünk azonosítókat, hiszen a **kony_szerzo** táblában idegen kulcsokkal kell jelezni az új könyv és az új szerző kapcsolatát.

Összesen tehát négy táblába kell rekordot elhelyeznünk. A nehézségeket tovább fokozza ez egyes műveletek esetén fennálló **hibalehetőség**.

Ha fel tudjuk venni az új könyvet és a szerzőt, de a **konyv_szerzo** tábla rekordjának beszúrásakor hiba keletkezik, akkor minden lépést vissza kellene vonnunk…

Töröljük az érintett táblából az utolsó rekordokat? Jó megoldás! ...Akkor, ha csak mi használjuk az adatbázist. A web alkalmazásnak azonban éppen az az

óriási előnye, hogy egyszerre többen is dolgozhatnak vele. Mi történik, ha a mi munkánk közben más is rögzített könyveket? Bizony ilyenkor az ő rekordjaikat törölnénk egy estleges visszalépéskor.

Szerencsére a MySQL InnoDB táblái tranzakcióbiztosak (a mi adatbázisunk táblái ilyenek), így nyugodtan használhatjuk a tranzakció kezelést!

Ehhez semmi egyebet nem kell tennünk, mint a **mysql_query()** függvénnyel, a megfelelő helyeken el kell küldenünk a "**BEGIN**", "**COMMIT**", illetve a "**ROLLBACK**" parancsokat.

Az első **INSERT** előtt a **mysql_query("BEGIN");** függvényhívással elindítjuk a tranzakciót. Elvégezzük a rekordok beszúrását, de minden egyes **INSERT**-eknél külön változóban tároljuk az erőforrás azonosítót. A rekordfölvéltelek után megvizsgáljuk őket. Ha bármelyik erőforrás azonosító **false**, akkor visszagörgetjük a tranzakciót. Azaz minden **INSERT**-et semmissé teszünk. Ha minden rendben ment, akkor egy **mysql_query("COMMIT")**-tal zárjuk a műveleteket.

Valójában egyetlen problémánk maradt. Hogyan tudjuk meg az elsődleges táblákba beszúrt rekordok azonosítóit? A beszúrás utáni utolsó rekord azonosítójának lekérdezése az ismert okok miatt nem járható út. A MySQL API-ban azonban rendelkezésre áll a **mysql_insert_id()** függvény, amely a megadott kapcsolat legutolsó AUTO_INCREMENT értékét adja vissza.

```
egész mysql_insert_id ([erőforrás_id] )
```

Ezzel a függvénnyel már megtudhatjuk az egyes INSERT parancsokkal beszúrt utolsó rekordok azonosítóit, hiszen a tábláink mindegyike AUTO_INCREMENT azonosítókat tartalmaz.

Forrás: mysql_query_insert.php

mysql_query_inser...

```php
<?php

    header("Content-type: text/html; charset=utf-8");
//Kapcsolat fölépítése
    create_conn();
//Tranzakció kezdete
    mysql_query("BEGIN");
//Kiadó beszúrása
    $dml_kiado="INSERT kiado (kINev) VALUES ('Szak Kiadó Kft.')";
    $res_kiado=mysql_query($dml_kiado);
    $id_kiado=mysql_insert_id();
//Könyv beszúrása
    $dml_konyv="INSERT konyv (kCim,kEv,kTargySzavak,idKiado)
                VALUES ('XML lépésről lépésre',2002,'informatika,XML, SGML, XPATH',$id_kiado)";
    $res_konyv=mysql_query($dml_konyv);
    $id_konyv=mysql_insert_id();
//Új szerző
    $dml_szerzo="INSERT szerzo (szNev) VALUES ('Young, Michael J.')";
    $res_szerzo=mysql_query($dml_szerzo);
    $id_szerzo=mysql_insert_id();
//Új kapcsolat
    $dml_szk="INSERT konyv_szerzo (idKonyv,idSzerzo)
                    VALUES($id_konyv,$id_szerzo)";
    $res_szk=mysql_query($dml_szk);
//Visszagörgetés hiba esetén
    if (!$res_kiado or !$res_konyv or !$res_szerzo or !$res_szk){
        mysql_query("ROLLBACK");
        die("A rekordok beszúrása <b>nem sikerült</b>!");
    }
//COMMIT, ha minden sikerült
    mysql_query("COMMIT");
    echo "A rekordok beszúrása <b>sikeres</b>!";
    mysql_close();

```

70. ábra Rekordok tranzakció biztos beszúrása

8.7 ÖSSZEFOGLALÁS, KÉRDÉSEK

8.7.1 Összefoglalás

Mai leckénkben a PHP MySQL API-jának legfontosabb függvényeit ismertük meg. Megtanultuk, hogy a MySQL-szerverek elérése a kapcsolat fölépítésével kezdődik, amit a **`mysql_connect()`** függvény hívásával végezhetünk el. Ezután következik a **`mysql_select_db()`** indítása, azaz az adatbázis kiválasztása. A létrehozott SQL-mondatot, a **`myslq_query()`** függvénnyel küldhetjük el az adatbázis-kezelő rendszernek, az eredményt pedig általában változóba tesszük. A DQL-lekérdezések erőforrás azonosítóval, vagy **`false`** értékkel a DML-mondatok **`true`**, vagy **`false`** eredménnyel térnek vissza. A relációkat visszaadó lekérdezések rekordjait a **`mysql_fetch_row()`**, illetve a **`mysql_fetch_assoc()`** függvények ciklikus hívásával nyerhetjük ki.

A DML-mondatok végrehatása után csak arra kell figyelni, hogy a művelet sikeres volt-e. Ha adatbázisunk táblái tranzakció biztosak, a **`BEGIN`**, **`COMMIT`**, és **`ROLLBACK`** parancsok elküldésével valósíthatjuk meg tranzakció kezelést.

8.7.2 Önellenőrző kérdések

1. Hány MySQL kapcsolatot nyithatunk meg ugyanahhoz a szerverhez egy PHP-szkriptben?

- Általában egyet nyitunk, de több kapcsolatot is fölépíthetünk. Tudni kell azonban, hogy a mysql_connect() többszöri meghívása esetén alapértelmezésben ugyanazt a kapcsolat azonosítót adja vissza. Ha azonban a függvény negyedik paraméterként true értéket adunk meg, akkor minden hívás új kapcsolatot alakít ki.

2. Megmaradhat-e a kapcsolat, ha a mysql_connect() függvénytől kapott azonosítót munkamenet változóként tároljuk?
   - Nem. A kapcsolat mindenképpen megszakad, amikor a PHP-szkript befejeződik.
3. Mire használható a mysql_set_charset() függvény?
   - A MySQL-szervertől érkező válaszok karakterkódolásának szabályozására.
4. Hogyan törölhet rekordot egy táblából?
   - A DELETE parancsot tartalmazó SQL-mondatot el kell küldeni a MySQL-szervernek, és megvizsgálni, hogy a mysql_query() függvény visszatérési értéke true, vagy false.
5. Hogyan tudja elérni, hogy az egymás után végrehajtott DML-lekérdezésnek csak akkor legyen hatása, ha mindegyik végrehajtása sikeres volt?
   - A lekérdezések elküldését tranzakción belül kell elvégezni. Ha bármelyik mysql_query() false eredményt ad vissza, ki kell adni a mysql_query("ROLLBACK"); parancsot. Ha minden DML-mondat sikeresen lefutott, akkor a mysql_query("COMMIT") hívással lehet jóváhagyni a tranzakciót.

9. LECKE: OBJEKTUMORIENTÁLT PROGRAMOZÁS A PHP-BEN

9.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

A világ számtalan másoktól elkülöníthető dologból, objektumból épül föl. Objektum lehet bármi: tárgy, fogalom, élőlény, ami a többi dologtól különbözik. Az objektumokat adatokkal leírható tulajdonságaik jellemzik.

A való életben fölmerülő problémák, feladatok megoldásakor a problémához kapcsolódó objektumokat manipuláljuk, úgy, hogy tulajdonságaikkal különböző műveleteket végezve, megváltoztatjuk azokat.

Ha az objektum például lakásunk egy szobája, akkor annak jellemző tulajdonsága többek között az alapterülete, vagy a falak festésének állapota. Ha a festés már kopott, akkor úgy oldhatjuk meg a problémát, hogy egy művelettel megváltoztatatjuk a helyiség megfelelő tulajdonságát ... azaz kifestjük a szobát.

Amikor a valós problémák megoldását számítógépes eszközökkel akarjuk támogatni, akkor modelleznünk kell az érintett objektumok adatait, és a velük végezhető műveleteket is. Az úgynevezett strukturált modellezés és programozás során a tulajdonságokat változókban tároljuk, az adatokkal végezhető műveleteket pedig alprogramokba, elhelyezett utasításokkal írjuk le.

Mivel függvények zárt egységet alkotnak, csak az adatok egy szűk körét dolgozzák föl. Egy-egy objektum azonban sok tulajdonsággal rendelkezik és számos tevékenység, művelet jellemzi. A napjainkra hagyományosnak tekinthető strukturált programozásban nem tudjuk megfelelően modellezni a teljes objektumot, a rá jellemző tulajdonságokat és tevékenységeket.

Az objektumorientált programozás erre a problémára adja meg a választ azzal, hogy lehetőséget teremt az objektumra jellemző adatok, és az azokkal végezhető tevékenységek egységbe zárt modellezésére és programozási eszközökkel történő leképezésére.

A PHP-t eredetileg strukturált programozásra szánták és sokan még ma is elsősorban ezeket a lehetőségeit használják. Az OOP kezdetleges támogatása a nyelv megjelenéséhez képest csak később, mai formájában pedig csak az PHP 5.0 verziótól kezdve jelent meg. Mai leckénkben ezzel a témakörrel foglalkozunk.

Fölelevenítjük az osztályokkal és objektumokkal kapcsolatos fontosabb tudnivalókat, megtanuljuk, hogyan lehet létrehozni a PHP-ben egy létező osz-

tály objektumait, hogyan tudunk hivatkozni az objektumok tulajdonságaira, illetve metódusaira. Áttekintjük a saját osztályok létrehozásának módját. Megtanuljuk, hogyan lehet létrehozni konstruktort, adattagokat, metódusokat, és hogyan lehet szabályozni a kód láthatóságát.

9.2 AZ OOP JELENTŐSÉGE

A procedurális nyelvekben írt programkód egymást követő utasításokból álló főprogramra és eljárásokra tagolódik.

Az objektum-orientált programozási (OOP) nyelvek a valóságot próbálják modellezni. A világban minden, a többitől elkülöníthető dolog, egy-egy objektum. Objektum bármi lehet: élőlény, tárgy, fogalom, jelenség...

Az objektumok lehetséges tevékenységeik, jellemzőik alapján csoportokba, úgynevezett osztályokba sorolhatók:

Bogáncs, Frakk és Lassie például mindhárman a kutyák osztályába sorolható objektumok. Azért sorolhatók közös osztályba, mert lehetséges tevékenységeik és tulajdonságaik azonosak.

Bogáncs, Frakk és Lassie mind tudnak ugatni, futni, ugrani (tevékenységek), van nevük, színük, fajtájuk, életkoruk (tulajdonságok).

Ha kutyarobotokat készítenénk, programkóddal kellene irányítanunk Lassie-t, Bogáncsot és Frakkot. Megtehetnénk, hogy különkülön programot írunk a három kutyarobotra, ez azonban nagy munka, a program utólagos változtatása pedig meglehetősen bonyolult lenne.

Olyan programkódra van szükség, amely kihasználja, hogy mindhárman kutyák, ezért ugyanazokat a dolgokat tudják csinálni, és ugyanolyan jellemzőik vannak. Valójában csak tulajdonságaik értékeiben különböznek.

9.3 OBJEKTUM-ORIENTÁLT PROGRAM

9.3.1 Egységbezártság

Az objektumorientált nyelvekben nem az egyes objektumok, hanem az objektumokat tartalmazó osztály programját írjuk meg. Az osztály kódja tartalmazza azokat a változókat, más néven **adattagokat**, amelyek a tulajdonságok értékeinek a tárolására valók, és magába foglalja azokat az függvényeket, azaz **metódusokat**, amelyek a lehetséges működést írják le, illetve kezelik az adattagokat.

> **Az osztály mintegy egységbe zárja, az objektumaira jellemző adattagokat, és metódusokat. Ezt sajátosságot nevezzük egységbezártságnak.**

Például:

Osztály ***Kutya***

Adattag ***Fajta***

Adattag ***Név***

Adattag ***Szín***

Adattag ***Életkor***

Metódus ***Ugrik()***

Metódus ***Ugat()***

Metódus ***Fut()***

Metódus ***Alszik()***

Metódus ***Eszik()***

Metódus ***Házat_őriz()***

9.3.2 Kód újrahasznosítása

Hogyan lesznek az osztályból példányok, azaz objektumok? Az objektumorientált nyelvekben – miután elkészítettük az osztály kódját – könnyedén létrehozhatjuk az osztály példányait, azaz az objektumokat. Ehhez a nyelv szabályai szerint azt kell jeleznünk, hogy a példány az osztály új objektuma.

Például:

```
Kutya1 = új Kutya
```

```
Kutya2 = új Kutya
```

```
Kutya3 = új Kutya
```

> **Az osztály alapján létrehozott objektumok mind rendelkeznek az osztály összes metódusával és adattagjával. Az egyes objektumok számára nem kell külön kódot írni. Ezt lehetőséget nevezzük a kódújrahasznosításának.**

Egy osztály alapján létrehozott összes objektum ugyanazokkal a metódusokkal és jellemzőkkel rendelkezik. Létrehozáskor minden objektum egyforma, de tulajdonságaik értékeinek szabályozásával különbözővé tehetjük őket. Egy objektum valamelyik adattagjára vagy metódusára való hivatkozáskor a legtöbb OOP-nyelv az úgynevezett pont (.) szintaxist használja:

```
   objektumnév.adattag, vagy objektumnév.metódus()
formában.
```

A PHP ezzel szemben a **nyíl operátorral** teszi elérhetővé egy objektum valamely tulajdonságát vagy metódusát.

```
   objektumnév->adattag, vagy objektumnév->metódus()
```

Kezdetben minden kutyánk egyforma, de adattagjaik értékének beállításával megkülönböztethetjük, metódusaik segítségével pedig vezérelhetjük őket:

```
Kutya1->Név="Lassie"
```

```
Kutya2->Név="Bogáncs"
```

```
Kutya1->Fajta="Skót juhász"
```

```
Kutya2->Fajta="Puli"
```

```
Kutya1->Ugat()
```

```
Kutya2->Eszik()
```

9.3.3 Öröklődés

Az osztály kódjának megírásával valójában egyfajta sablont készítettünk az osztály objektumai számára. Nagy előny, hogy ezek után akármennyi objektumot is létrehozhatunk az osztály megváltoztatása nélkül.

Egy objektumorientált nyelven írt programban általában több osztályra van szükség. Ha például a kutyarobot programunkat kibővítve macskarobotok irá-

nyítására is alkalmassá akarjuk tenni, akkor egy **Macska** osztályt is kell készítenünk megfelelő metódusokkal és adattagokkal.

 *Osztály* ***Macska***

 *Adattag* ***Fajta***

 *Adattag* ***Név***

 *Adattag* ***Szín***

 *Adattag* ***Életkor***

 *Metódus* ***Ugrik()***

 *Metódus* ***Nyávog()***

 *Metódus* ***Fut()***

 *Metódus* ***Alszik()***

 *Metódus* ***Eszik()***

 *Metódus* ***Dorombol()***

 *Metódus* ***Egerészik()***

 *Metódus* ***Karmol()***

Ezután létrehozhatjuk a macskapéldányokat, objektumokat

```
Macska1 = új Macska
```

```
Macska1->Név= "Tom"
```

```
Macska1->Alszik()
```

Könnyen észrevehetjük, hogy a két osztály számos adattagban és metódusban megegyezik. Hogy lehetne tovább takarékoskodni a programkóddal? Az objektumorientált programozás lehetővé teszi, hogy egy osztály létrehozáskor örökölje egy már meglévő osztály kódját.

*A **Kutya** és **Macska** osztály kódjának megírása egyszerűbb lesz, ha előbb elkészítjük a **Háziállat** osztályt, amelyben a házi kedvencek összes közös adattagja és metódusa megtalálható*

Osztály **Háziállat**

Adattag **Faj**

Adattag **Fajta**

Adattag **Név**

Adattag **Szín**

Adattag **Életkor**

Metódus **Ugrik()**

Metódus ***Fut()***

Metódus ***Alszik()***

Metódus ***Eszik()***

Ez után a **Kutya** és **Macska** osztályok elkészítése egyszerűbb, mert nem kell megírni a közös metódusokat, csak jelezni kell, hogy ezek az osztályok a **Háziállatok** osztály alapján, annak kibővítéseként készületek, öröklik annak minden metódusát, és adattagját:

Osztály Kutya Háziállat kibővítésével

Osztály Macska Háziállat kibővítésével

Az objektumorientált programozásban öröklődésnek nevezik azt a lehetőséget, amikor egy osztály, létrehozáskor átveszi egy másik, már létező osztály kódját. Szülő osztálynak nevezzük az örökítőt származtatottnak az öröklő osztályt. Az öröklődés a kód újrahasznosításának másik lehetősége.

Kutya, Macska Háziállat példánkban a Háziállat a szülő osztály a Kutya, és a Macska a származtatott osztályok.

9.3.4 Polimorfizmus

Az öröklődés önmagában kétélű fegyver lenne, hiszen így a származtatott osztályok azonos metódusokkal és adattagokkal rendelkeznének. A Kutya és Macska osztály egyforma lenne, hiszen mindkettő a Háziállat osztályból származik.

Az objektumorientált programozás azonban lehetőséget biztosít a származtatott osztályok kiegészítésére, módosítására. Amikor létrehozunk egy származtatott osztályt, abban új metódusokat, vagy adattagokat is helyezhe-

tünk, sőt, arra is van lehetőségünk, hogy átírjuk a szülőtől örökölt kódot. Ezzel a technikával az azonos szülőtől származó osztályok különbözővé tehetők, specializálhatók.

Azt a lehetőséget, hogy az azonos szülőtől származó osztályok az átírás következtében különbözőek lehetnek, polimorfizmusnak nevezik.

Osztály ***Kutya Háziállat kibővítésével***

Metódus ***Ugat()***

Metódus ***Házat_őriz()***

Osztály ***Macska Háziállat öröklődésével***

Metódus ***Nyávog()***

Metódus ***Dorombol()***

Metódus ***Egerészik()***

Metódus ***Karmol()***

Így a ***Kutya*** *és a* ***Macska*** *osztály is örökli a* ***Háziállat*** *osztály minden metódusát és adattagját, de a Kutya osztály kiegészül az* ***Ugat****, és a* ***Házat*** *őriz, a* ***Macska*** *osztály pedig a* ***Nyávog****,* ***Dorombol****,* ***Egerészik*** *és* ***Karmol*** *metódusokkal.*

Az objektumorientált programozás fontos előnyei tehát az egységbezártság, a kód újrahasznosítása, az öröklődés és a polimorfizmus.

9.4 OBJEKTUMORIENTÁLT NYELVEK

Számos különböző objektumorientált programozási nyelv létezik. Akármelyikkel is dolgozunk, tudnunk kell:

- hogyan hozhatunk létre önálló,
- és származtatott osztályokat,
- hogyan határozhatjuk meg egy osztály adattagjait, és metódusait,
- hogyan lehet olyan metódust írni, ami egy példány (objektum) létrehozásakor beállítja az egyedi tulajdonságértékeket,
- hogyan lehet létrehozni az osztályok objektumait, példányait,
- hogyan lehet beállítani adattagjaik értékét,
- hogyan lehet működésre bírni metódusaikat.

Az, hogy egy nyelv objektumorientált, önmagában még nem jelenti, hogy a programkódot egyszerű lenne megírni. Ha a programot a „semmiből" kellene elkészíteni, akkor még egy viszonylag kis feladat megoldásához is elég sok osztály megírására lenne szükség.

A legtöbb objektumorientált nyelvet úgy készítik el, hogy abba beépítenek számos, előre megírt osztályt. Ezeket programozás közben már nem kell megírni, azonnal létrehozhatjuk, és felhasználhatjuk objektumaikat, vagy elkészíthetjük származatott osztályaikat. A PHP-ben ilyen előre definiált osztály például a **mysqli API** által biztosított **mysqli** osztály, amelynek metódusaival hasonló feladatokat valósíthatunk meg, mint az általunk használt procedurális API segítségével.

Az osztályok egyes objektumai speciális objektumváltozók, amelyek létrehozásakor a *változó***=new** *osztálynév***([***paraméterek***]);** szintaxist használjuk. Ilyenkor valójában az osztály egy speciális metódusát, az úgynevezett **konstruktort** indítjuk el, ami létrehozza a példányt és beállítja fontosabb tulajdonságainak kezdőértékét.

A **mysqli** osztály egy objektumát például az alábbi módon hozhatjuk létre:

```
$db = new mysqli("localhost", "webprog", "guttenberg",
"konyvek");
```

A létrehozott **$db** változó a **mysqli** osztály egy objektuma. A példányt az osztály a konstruktor metódusa a fenti értékek átvételével hozta létre, és azonnal föl is építi az adatbázis kapcsolatot.

A további műveletek már az objektumváltozón keresztül hívhatók.

```
$res=$db->mysql_query("SELECT * FROM konyv");
```

9.5 SAJÁT OSZTÁLY LÉTREHOZÁSA

A PHP-ben nincsenek korlátozások az osztály kódjának elhelyezésére. Az osztályt létrehozhatjuk abban a fájlban, amelyben az alkalmazásunk főprogramja van, de tehetjük különálló állományba is, majd az **include** vagy **require** utasításokkal csatolhatjuk. Az sincs korlátozva, hogy egy fájlban hány osztály kódját helyezhetjük el. Tehetünk minden osztályt külön állományba, de egyetlen dokumentumba is integrálhatjuk osztályainkat. Arra azonban nagyon kell figyelni, hogy a származtatott osztályok értelmezése a szülő osztály után következzen.

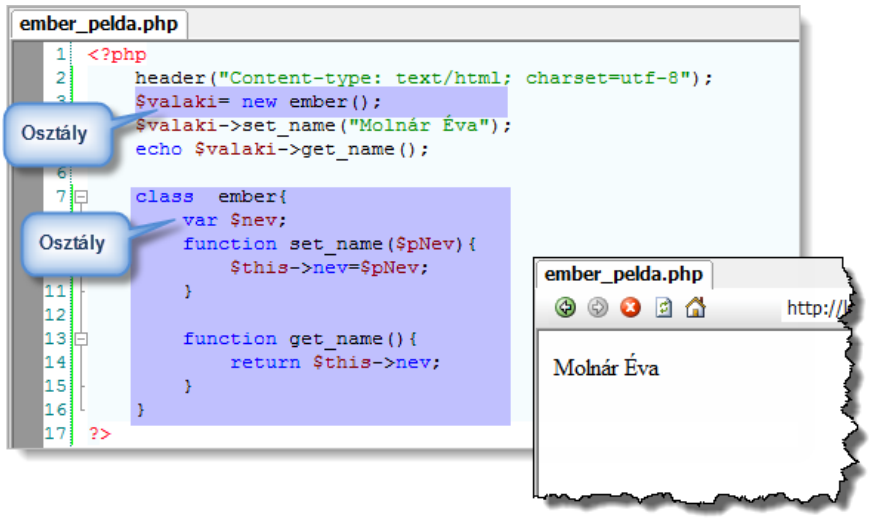
9.5.1 Osztály létrehozásának szintaxisa

Az osztály létrehozása abból áll, hogy megfelelő formában leírjuk az osztály adattagjait és metódusait.

```
class osztálynév {
   adattagok
   metódusok
}
```

Az *osztálynév* lesz az új osztály hivatkozási neve. Az *adattagok* változó deklarációk, amelyeket a var kulcsszóval (vagy a később részletezett láthatóság szabályozók valamelyikével) kell kezdeni. A metódusok a már ismert módon létrehozott függvények.

Forrás: ember_pelda.php; ember_tobb_objektum.php



71. ábra Ember osztály

A fenti osztály egy adattaggal (**$nev**) és két metódussal (**get_name**, **set_name**) rendelkezik. A **set_name** a **$nev** adattagban tárolja az átvett paraméter értékét, a **get_name** pedig visszatérési értékként adja vissza.

Ha az osztály kódját megírtuk, létrehozhatjuk az osztály változóit.

```
$valaki=new ember();
$valaki->set_name("Molnár Éva");
```

A példányon keresztül a nyíl (**->**) operátorral hivatkozunk a példány adattagjaira és a metódusaira. Az osztály kódján belül szintén a **->** operátort használjuk, amikor a leendő példány adattagjaira és metódusaira hivatkozunk. Ilyenkor azonban objektumváltozó még nem létezik, ezért helyén a **$this** változó áll.

9.6 ADATTAGOK ÉS METÓDUSOK LÁTHATÓSÁGA

A **->** operátorral tehát hivatkozhatunk az objektum adattagjaira (tulajdonságaira) és metódusaira is. Az ember osztály egy példányának nevét ezek szerint így is beállíthatjuk:

ember_novalid.php

```php
<?php
    header("Content-type: text/html; charset=utf-8");
    $valaki= new ember();
    $valaki->nev="Molnár Éva";
    echo $valaki->get_name();

    class  ember{
        var $nev;
        function set_name($pNev){
            $this->nev=$pNev;
        }

        function get_name(){
            return $this->nev;
        }
    }
?>
```

72. ábra Adattag közvetlen elérése

Forrás: ember_novalid.php

Az adattagok metódusokon keresztül történő elérése azért szerencsésebb, mert a programozó így validálni is képes az adatokat.

Forrás: ember_validal.php

ember_validal.php

```php
<?php
    header("Content-type: text/html; charset=utf-8");
    $valaki= new ember();
    $valaki->set_born(2030);

    class  ember{
        var $nev;
        var $szulev;

        function set_born($pYare){
            if ($pYare>date("Y")) return $this->born_error();
            $this->szuletett=$pYare;
        }

        function born_error(){
            echo "Még nem született meg.<br/>";
            return false;
        }

        function set_name($pNev){
            $this->nev=$pNev;
        }

        function get_name(){
            return $this->nev;
        }
    }
?>
```

Adattag elérése metóduson keresztül

ember_validal.php

http://lo

Még nem született meg.

73. ábra Adattag elérése metóduson keresztül

A dolog persze megkerülhető, hiszen az adattagot közvetlenül is elérjük. Az ilyen hibákat a változó és metódusnevek elé írt **public**, **private**, vagy **protected** kulcsszavakkal szabályozható „láthatóság” beállításával korrigálhatjuk.

A **private** láthatóságú metódusok, illetve adattagok csak az adott osztály kódján belül látszanak. A **protected** láthatóság szintén nem engedi a „külső” hozzáférést, de öröklődéssel létrehozott osztályok kódjából elérhetők lesznek a szülő osztály **protected** elemei.

A **public** elérés az alapértelmezett. Ha nem tűntetünk föl más láthatóságot, akkor ez lesz érvényben és az adattag vagy metódus korlátozás nélkül elérhető lesz.

Ha tiltani akarjuk a születési idő közvetlen beállítását, akkor az ***$szulev*** *adattagot tegyük* ***private*** *elérésűvé.*

A változó deklaráláskor használt **var** kulcsszó a **public** szinonimája. Ha szabályozni akarjuk a láthatóságot, **private**, vagy **protected** kulcsszóval kell helyettesítenünk.

Forrás: ember_valid.php

ember_valid.php

```php
<?php
    header("Content-type: text/html; charset=utf-8");
    $valaki= new ember();
    $valaki->set_born(2030);

    class  ember{
        private $nev;
        private $szulev;

        function set_born($pYare){
            if ($pYare>date("Y")) return $this->born_error();
            $this->szuletett=$pYare;
        }

        private function born_error(){
            echo "Még nem született meg.<br/>";
            return false;
        }

        function set_name($pNev){
            $this->nev=$pNev;
        }

        function get_name(){
            return $this->nev;
        }
    }
?>
```

74. ábra Láthatóság szabályozása

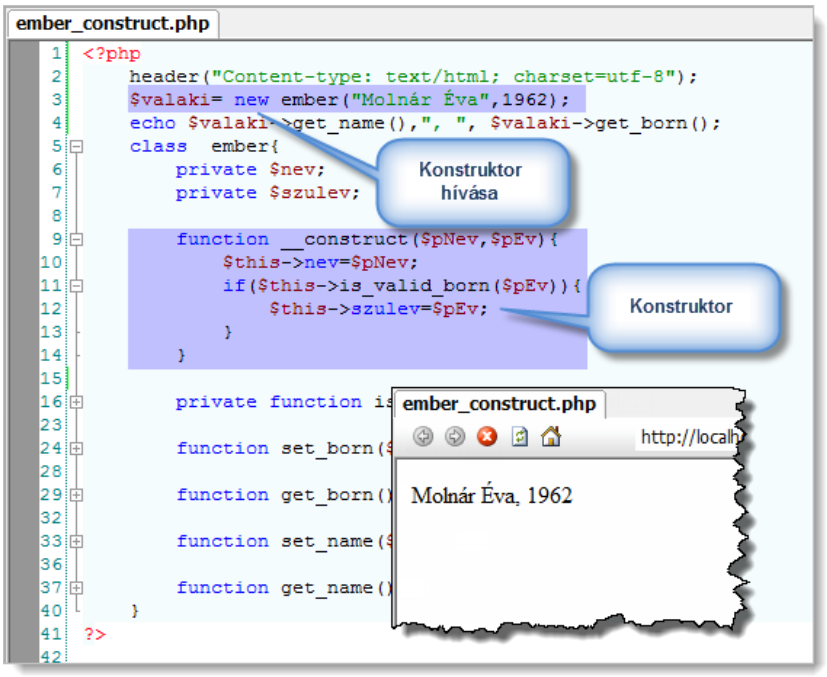
9.7 KONSTRUKTOR, DESTRUKTOR

Előfordul, hogy amikor létrehozunk egy objektumot, azonnal be akarjuk állítani néhány tulajdonságát. Erre a feladatra használható az úgynevezett **konstruktor** metódus. A konstruktor speciális metódus, ami automatikusan meghívásra kerül az objektum létrehozásakor (**`new`** *osztály*).

A konstruktornak a **`__construct`** nevet, és mindenképpen **`public`** láthatóságot kell kapnia.

A **`new`** kulcsszó ezt a metódust fogja hívni, ezért ha deklarálásakor paraméterváltozókat adunk meg, akkor az objektum létrehozásakor át kell adni az ezeknek megfelelő értékeket.

Forrás: ember_construct.php



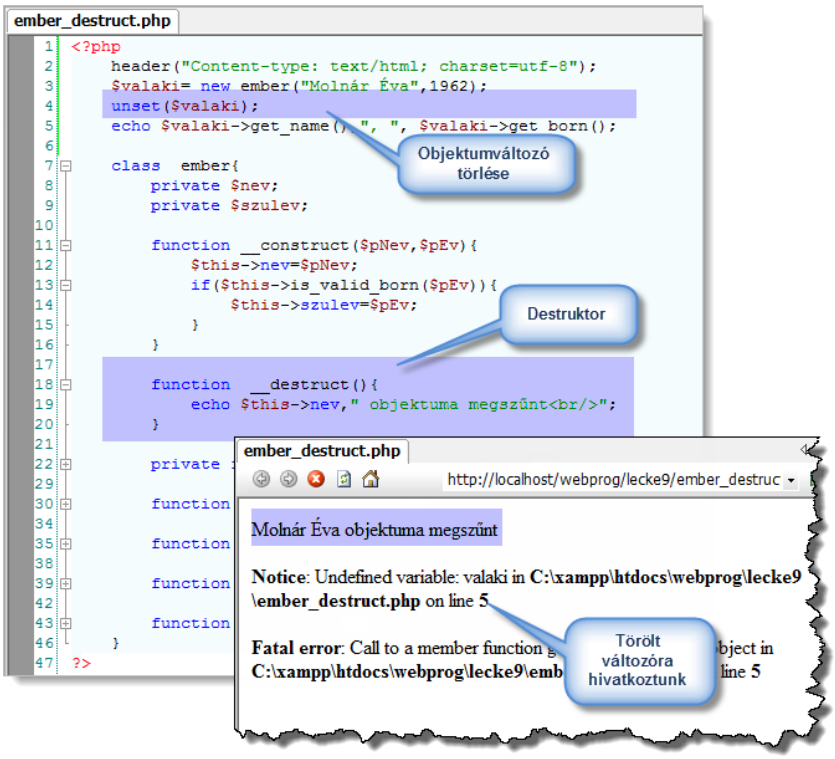
75. ábra Konstruktor és hívása

A konstruktorhoz hasonlóan létrehozható az objektum megszűnéskor automatikusan meghívott eljárás is. A **`__destruct`** metódusban fölszámolhatjuk az objektum esetleges adatbázis kapcsolatait, naplózási feladatokat láthatunk el, üzenetet jeleníthetünk meg.

A **`__destruct`** eljárást akkor hívja meg a PHP, amikor az objektumváltozó megszűnik.

Vigyázzunk, ha megszűntettünk egy változót, akkor már ne hivatkozzunk rá többet!

Forrás: ember_destruct.php



76. ábra Destruktor hatása

9.8 ÖRÖKLŐDÉS

Új osztályok kódját nem mindig kell teljesen újraírni. Az alábbi formában létrehozhatjuk őket meglévő osztályok alapján, örökléssel is.

```
class származtatott extends szülő {
   osztály_kódja
}
```

A származtatott osztály minden metódussal és tulajdonsággal rendelkezni fog, amellyel a szülő is bír. A származatott osztály objektumaival csak a szülő publikus metódusai, az osztályból pedig a szülő publikus és protected elemei is elérhetők.

A származtatott osztályban új metódusokat és adattagokat helyezhetünk el, átírhatjuk a szülőtől örökölt elemeket, vagy változatlanuk hagyhatjuk őket.

Ha a leszármazott osztály kódjából a szülő valamelyik elemére akarunk hivatkozni, akkor az alábbi formulát kell használni.

```
parent::$adattag
parent::metódus()
```

Az alábbi példában a **dolgozo** osztály az **ember** osztály leszármazottja. Felülírja a konstruktort, két új adattaggal és metódussal bővíti a szülő elemeit. Az ember osztály konstruktora két paramétert vár, a dolgozó összesen négyet. Kettőt maga kezel, de a másik kettővel meghívja a szülő konstruktorát.

Forrás: ember_inherit.php

```php
<?php
    //ember_inherit.php
    header("Content-type: text/html; charset=utf-8");
    $valaki= new ember("Molnár Éva",1962);
    $mas=new dolgozo("Gál Lajos",1970,"manager",1997);
    echo $mas->get_name()," ",$mas->get_assignment()," ",$mas->get_since()," óta dolgozik a cégnél.";

    class  ember{
        private $nev;
        private $szulev;

        function __construct($pNev,$pEv){
            $this->nev=$pNev;
            if($this->is_valid_born($pEv)){
                $this->szulev=$pEv;
            }
        }


        static public function get_age($pBorn=null){
            if(is_null($pBorn)) $pBorn =$this->szulev;
            return date("Y")-$pBorn;
        }
        ...
            return $this->nev;
        }
    }

    class dolgozo extends ember{
        private $munkakor;
        private $belepes;
        function __construct($pNev,$pEv,$pMunka,$pBelep){
            $this->munkakor=$pMunka;
            parent::__construct($pNev,$pEv);
        }

        function get_since(){
            return date('Y')-$this->belepes;
        }


        function get_assignment(){
            return $this->munkakor;
        }

    }
?>
```

Objektumok

Szülő

Származatott

ember_inherit.php

http://localhost/webprog/lecke

Gál Lajos manager 2012 óta dolgozik a cégnél.

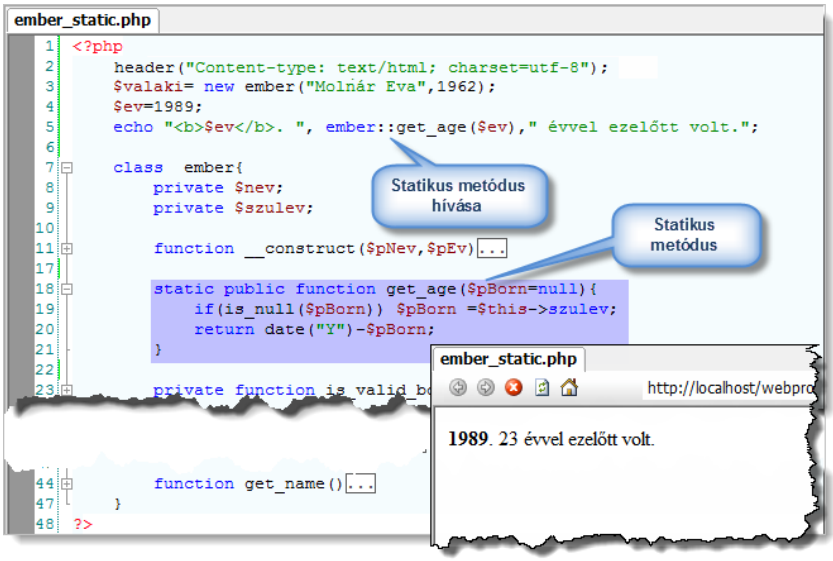
77. ábra Származtatott osztály

9.9 STATIKUS METÓDUSOK

Az osztályok metódusai és adattagjai akkor érhetők el, ha létrehozzuk az osztály objektumait. Előfordul, hogy olyan elemeket szeretnénk létrehozni egy osztály kódjában, amelyek objektumok nélkül, közvetlenül az osztály kódjára hivatkozva is elérhetők. Az ilyen elemeket statikus adattagoknak, illetve statikus metódusoknak nevezzük.

A statikus elemeket a deklarációjuk elején elhelyezett **`static`** kulcsszóval jelezzük, hivatkozáskor pedig az *osztálynév*`::`*statikus_elem* formulával érjük el őket.

Forrás: ember_static.php



78. ábra Statikus metódus elérése

9.10 OBJEKTUMOK ÁLLAPOTÁNAK MEGŐRZÉSE

Korábban már tanultunk a PHP állapotkezelési lehetőségeiről. Az objektumok állapotának kérések közötti megőrzésére a rejtett INPUT-mezők és a cookie-k csak erős korlátozásokkal, néhány adattag erejéig használhatók.

Az igazán alkalmas megoldást a munkamenet kezelés biztosítja, azonban ha tanult technikát alkalmazzuk, programunk nem fog működni. Ennek az az oka, hogy a PHP szöveggé alakítva tárolja az adatokat munkamenet fájlban. Ha objektumot akarunk tenni a $_SESSION változóba, akkor azt előbb megfelelő formátumú szöveggé kell alakítani, amikor pedig ismét betöltjük a változót, akkor a szövegből vissza kell állítani az objektumot.

```
<?php
   session_start();
   if(!isset($_SESSION["myobject"])){
      $ myobject =new myclass();
      $_SESSION["myobject"]=serialize($myobject);

   } else {
      $c=unserialize($_SESSION["myobject "]);
   }
   //Program kódja
   $_SESSION["myobject "]=serialize($myobject);
?>
```

9.11 ÖSSZEFOGLALÁS, KÉRDÉSEK

9.11.1 Összefoglalás

Az objektumorientált programozás a strukturált programozásnál jóval hatékonyabb lehetőséget biztosít a valóság modellezésében, a program elkészítésében, és karbantartásában is. Jellemzői az egységbezártság, az öröklődés, a kód-újrahasznosítás és a polimorfizmus.

A PHP kevés formai megkötést fogalmaz meg az osztályokkal kapcsolatban. Az osztály kódja a főprogrammal azonos, vagy különböző fájlban is lehet. Egy állományban több osztály is elhelyezkedhet. Fontos azonban, hogy a származtatott osztályok kódja mindig a szülő osztály után kerüljön a forráskódba.

Az osztályok létrehozása a **`class`** kulcsszóval történik, amit az osztály neve, majd kapcsos zárójelek közötti kódja követ. Az osztályban a **`var`**, vagy a láthatóságot szabályozó valamelyik kulcsszóval deklarálhatjuk a változókat. A metódusok hozzáférését szintén a **`public`**, **`protected`**, és **`private`** kulcsszavakkal szabályozhatjuk.

A **`__construct`** és a **`__destruct`** metódusok speciális függvények. A **`__contruct`** az új objektum meghívásakor automatikusan kerül hívásra, és megkapja a létrehozáskor átadott peremétereket. Az objektum alaphelyzetének beállítására használjuk.

A **`__destruct`** az objektum megszűntetésekor kap vezérlést.

A létrehozott objektumon keresztül a nyíl operátorral **`->`** férhetünk hozzá az osztály metódusaihoz, és adattagjaihoz. Az osztályon belül szintén a nyíl operátort használjuk, de az objektumot a **`$this`** változó helyettesíti. A **`$this`**-szel valójában az objektum saját elemeire hivatkozik.

Az osztályokat származtatással is létrehozhatjuk. Ilyenkor az új osztály a szülő minden hozzáférhető metódusával és adattagjával rendelkezik. A származtatott osztályok felüldefiniálhatják a szülő metódusait és adattagjait, sőt új elemekkel is bővíthetik saját kódjukat. A származtatott osztályból a **`parent::`** kulcsszóval érhető el a szülőosztály kódja.

9.11.2 Önellenőrző kérdések

1. Sorolja fel az OOP legfontosabb előnyeit!
   - Egységbezártság, öröklődés, kód-újrahasznosítás, polimorfizmus.
2. Hogyan hozhatja létre egy osztály új objektumát?
   - Új osztályként vagy származtatással. Mindkét esetben a class *osztálynév* formulát kell használni, de származtatáskor ezt az extend kulcsszó és a szülő osztály neve követi.
3. Lehet-e objektumot munkamenetben tárolni?
   - Igen, de a serialize függvénnyel szöveggé alakítva tehető a $_SESSION tömbbe. Kiolvasáskor az unserialize függvényt kell hívni.
4. Mit jelent a PRIVATE, a PROTECTED és a PUBLIC kulcsszó?
   - Egy osztály PROTECTED tagjait csak a származtatott osztály kódjából lehet elérni. A PRIVATE elemek „kívülről" nem használhatók, a PUBLIC kulcsszó használatakor azonban szabadon hozzáférhetővé válik az elem.
5. Hogyan hívhatja meg a származtatott osztályból a szülőosztály konstruktorát?
   - A parent::__constuct() formulával. A hívással annyi paramétert kell átadni, amennyit a szülő konstruktora vár.

10. LECKE: HIBAKERESÉS, HIBA- ÉS KIVÉTELKEZELÉS

10.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

A 10. leckéig eljutó olvasó már minden bizonnyal találkozott azzal a kellemetlen helyzettel, amikor hosszú perceket kellett töltenie saját programja, általában elgépelésből adódó szintaktikai hibáinak megtalálásával. Ilyenkor a program futása megáll, és jó esetben hibaüzenet tudatja, hogy az melyik sorban akadt el az interpreter a végrehajtásban. Tipikus szintaktikai hiba az elhagyott sorvégi pontosvessző, a felsorolásból hiányzó vessző, a változónév elől hiányzó dollár jel, a páratlan zárójel, vagy idézőjel.

Nos, bármilyen bosszantóak is ezek a hibák, hamar észrevesszük, és egyszerűen kijavíthatjuk őket. Sokkal több kellemetlenséget okoznak az úgynevezett futásidejű hibák, amelyeket az egyébként szintaktikailag helyes programban a váratlan külső körülmények idéznek elő. Ilyen körülmény lehet például a felhasználótól származó feldolgozhatatlan tartalmú adat, a hálózati kommunikációban bekövetkezett hiba, vagy a program rendelkezésére álló memória elfogyása. A futás idejű hibák miatt programunk könnyen kerülhet kezeletlen helyzetbe, olyan definiálatlan állapotba, amiből képtelen bármilyen következő állapotba lépni. A váratlan leállások elkerülése érdekében programunkat úgy kell elkészíteni, hogy az fölkészült legyen a szokásostól eltérő körülmények kezelésére is. Természetesen nem lehet minden létező esetet külön-külön kezelő programkódot írni. Az azonban meghatározható, hogy mi történjen, ha előre nem tervezett helyzet, más néven hiba állna elő. Mai leckénkben arról lesz szó, hogyan kezelhetjük a programunk működése során bekövetkező váratlan helyzeteket, hogyan írjunk olyan programot, amely képes reagálni a futás közben bekövetkező hibákra, és az ilyen esetben is mindig előre tervezett állapotba kerül.

A leckében megismerheti a PHP hibaszintjeit, és a hibák kezelésére alkalmas technikákat. Megtanulhatja, hogyan helyettesítheti saját függvénnyel a die() utasítást, és elsajátíthatja a PHP-ben újnak számító többszintű hibakezelés kivitelezését.

10.2 FUTÁSIDEJŰ HIBÁK

A PHP úgynevezett interpretált nyelv. Az értelmező nem előre lefordított kódot futtat, mint például a Java, C, vagy C# esetében, hanem futásidőben, közvetlenül végrehajtás előtt értelmezi a kódot. Emiatt a hibák is csak futásidő-

ben a PHP-állomány feldolgozásakor, a program használatkor derülnek ki. Ebből persze az is következik, hogy az esetleges hibaüzenetek a felhasználó előtt, a böngészőben jelennek meg.

A PHP interpretált volta miatt furcsa futásidejű, és fordítási idejű hibáról beszélni, hiszen a jellemzően fordítási idejű szintaktikai hibák is futás közben derülnek ki. A fordítás és a futtatás valójában mégis élesen elkülönül egymástól. Amikor a felhasználó megpróbál letölteni egy php szkriptet, a webszerver átadja a programot tartalmazó fájlt a php értelmezőnek, a futtató motornak, az úgynevezett Zend Engine-nek (Zend: Zeev Suraski és Andi Gutmans nevéből). A Zend Engine először lexikális, majd szintaktikai elemzést végez az egész fájlon, majd, ha nem talál hibát, akkor előállítja az úgynevezett opcode-ot, a PHP-szkript futtatható változatát. Ez ugyan nem gépi kód, de a Javában, a .NET-nyelvcsaládban, vagy az ActionScript-ben alkalmazotthoz hasonló úgynevezett bájtkód, amit a motor részét képező virtuális gép képes futtatni.

A továbbiakban a szintaktikai hibákat értjük fordítás, az opcode végrehajtása közben jelentkezőket pedig futás idejű hiba alatt.

A program elindítása után számtalan olyan esemény következhet be, amely megakadályozza a program hibamentes futását. A PHP-értelmező észleli ezeket a hibákat, és képes rájuk reagálni.

A fordítási idejű hibák esetén a program el sem indul, ezért a klasszikus interpretált nyelvekkel (BASIC) szemben ilyenkor egyetlen utasítás sem lesz végrehajtva, csak a hibaüzenet jelenik meg a kimeneten.

A futás idejű hibák esetén azonban az értelmező reakciója kétféle lehet. Állhat csupán hibaüzenet küldéséből, vagy a program hibaüzenet melletti befejezéséből. A leállás okozó hibák kezelésébe nem tudunk beavatkozni azonban az első kategóriába tartozó feldolgozását a hibát okozó programban is el tudjuk végezni. Az ilyen eseményeket éppen ezért nevezzük kezelhető hibáknak. A hibák kezelése alatt azok érzékelését, és a hibára adott programozott reakciót értjük.

A PHP a fent említett, nagyon egyszerű, beépített hibakezeléssel rendelkezik. A hibák érzékelésekor üzenet jelenik meg a kimeneten, és a hiba típusától függően leállítja, vagy tovább engedi a program futását.

Természetesen előfordulhatnak olyan körülmények is, amelyeket a PHP nem tekint hibának, de a mégis megakadályoznák a program helyes működését. Az értelmező például minden reakció nélkül halad tovább, ha a **`mysql_query()`**

függvénnyel elküldött SQL-mondatot nem tudja végrehajtani az adatbázis-kezelő rendszer, de a programunk ilyenkor valószínűleg nem fog helyesen működni. Az ilyen, úgynevezett felhasználói hibák kezelésére a PHP közvetlenül nem képes.

10.3 HIBAKEZELÉS DIE() UTASÍTÁSSAL

A PHP-értelmező által figyelmen kívül hagyott, de a programunk működését veszélyeztető körülményeket magunknak kell kezelnünk. Ha például a **mysql_query()** függvénnyel elküldött SQL-mondat hibás, és semmilyen eredményhalmazt nem kapunk vissza, akkor mindössze annyi történik, hogy a függvény visszatérési értéke erőforrás helyett hamis érték lesz. A PHP továbbhalad, de a program további utasításai (**mysql_fetch_row()**) már hibát eredményezhetnek, ha fel akarják dolgozni azt a változót, amiben most erőforrás helyett logikai **false** érték van.

Az ilyen hibákat a program által kezelt adatok, és a meghívott függvények visszatérési értékeinek figyelésével ellenőrzésével érzékelhetjük a keletkezés helyén:

```
if (!mysql_query(…))…
```

Kezelésükre eddig egyféle módszert, a **die()** utasítás használatát ismertük meg. A **die()** hívásakor a kimenetre küldi a paraméterként kapott szöveget, majd leállítja a program futását. A hibakezelésnek ez a módszere legalább annyira drasztikus, mint a PHP beépített lehetőségei, ugyanis nem ad lehetőséget arra, hogy a program újra valamilyen definiált állapotba kerüljön. A használatából származó egyetlen előny, hogy akkor is meghívható, amikor a PHP nem talál hibát, és pontosan megadható az üzenet szövege.

```
if (!mysql_query("rossz lekérdezés"))
die("Lekérdezési hiba!");
```

Forrás: error_die.php

10.4 SAJÁT HIBAKEZELÉS

Ha valódi, saját hibakezelést akarunk megvalósítani, akkor egységes technikát kell alkalmaznunk a PHP által érzékelhető és a nem érzékelhető hibák kezelésére. Ennek első lépése a PHP hibaüzeneteinek letiltása, a második, hogy minden hibát azonos programkóddal dolgozzunk fel.

10.4.1 PHP hibaüzeneteinek letiltása

A PHP értelmező által érzékelhető hibákat 15 kategóriába, úgynevezett hibatípusba soroljuk. A típusokat kettő hatványaiként előállítható számkódok (2^0 - 2^14, 1..16384) azonosítják, amelyek konstansok (állandók) formájában állnak rendelkezésre a programnyelvben. Az alábbi táblázat a legfontosabb hibatípusokat tartalmazza.

Érték	Konstans	Leírás
1	E_ERROR	Fatális, nem kezelhető futás idejű hiba. A program végrehajtása megszakad.
2	E_WARNING	Figyelmeztetés kezelhető futásidejű hibára. A program végrehajtása folytatódik.
4	E_PARSE	Fordítás idejű hiba. A kódelemző generálja, még a futás előtt. A program végrehajtása nem kezdődik meg.
8	E_NOTICE	Futásidejű megjegyzés. Az értelmező lehetséges hibaforrást talált, ami lehetséges hibaforrás. A végrehajtás nem szakad meg, ugyanis nem biztos, hogy a valóban bekövetkezik a hiba.
256	E_USER_ERROR	A felhasználó által dobott problémát jelző hiba. Súlyossága az E_ERROR-hoz hasonló, azonban ezt nem a PHP, hanem a felhasználó generálja a trigger_error () függvénnyel.
512	E_USER_WARNING	A felhasználó által, a trigger_error () függvénnyel generált E_WARNING súlyú figyelmeztetés.
1024	E_USER_NOTICE	A felhasználó által generált, E_NOTICE súlyú hiba.
4096	E_RECOVERABLE_ERROR	Elkapható, de végzetes hiba. Olyan hibát jelöl, ami nem állította le a PHP-motort. Ha nem kezeljük a saját hibakezelőben, akkor leállítja a szkriptet.
8192	E_DEPRECATED	Olyan hiba, amit nem javasolt utasítások használatakor küld az interpreter.
32767	E_ALL	Ez a konstans (a PHP 5.4 előtt az E_STRICT kivételével) az összes hibát és figyelmeztetést jelenti egyben.

Alapértelmezés szerint minden hibatípus hibaüzenetet eredményez, azonban a PHP **`error_reporting()`** függvényével ez a működés felülbírálható. A függvénnyel beállíthatjuk, hogy melyik kategória hibaüzeneteit engedjük megjelenni.

Az **`error_reporting()`** paramétere egy kétbájtos bináris szám, aminek helyiértékei az egyes hibaszinteket jelölik. A függvény meghívása után csak a szám bekapcsolt (1 érékű) bitjeinek megfelelő típusokba tartozó hibák üzenetei jelennek meg.

Az függvény paraméterének megadásakor az egyes szinteket jelző konstansok közötti bináris **vagy** operátorral (`|`) állítjuk elő. Az **`error_reporting(0)`** a fordítási hibákon kívül mindent letilt. A PHP ilyenkor természetesen változatlanul érzékeli a „letiltott” hibákat, csak azt, hogy nem írja ki a hibaüzeneteket.

```
error_reporting(E_WARNING | E_NOTICE )
```

A fenti esetben csak a figyelmeztetések és a megjegyzések üzenetei jelennek meg a kimeneten.

10.4.2 Hibakezelő függvény

A **`die()`** függvény hívásakor csak egy üzenettel és a program azonnali megszakításával reagálhatunk a hibákra. A függvény sem differenciált beavatkozásra, sem a hiba esetleges korrigálására, sem pedig a továbblépésre nem biztosít lehetőségeket.

Az azonnali leállítás hiányosságait kiküszöbölő hibakezelési technikák egyike a saját **hibakezelő függvény** használata. A hibakezelő az úgynevezett **callback** függvények közé tartozik. Ezek a függvényeket a PHP értelmező **automatikusan hívja meg** valamilyen **esemény bekövetkeztekor**, és **paraméterként** átadja nekik az eseményt **jellemző adatokat**. A callback-függvények szintaktikailag csak annyiban különböznek a többi függvénytől, hogy formális paraméterlistájuk kötött. Ez azért van, mert a hívásakor a függvény **paraméterváltozóiba** az esemény **meghatározott leíró adatait** helyezi a PHP.

A hibakezelő függvény hiba esetén indul el, és legalább két paramétert kap. Formális paraméterlistájának éppen ezért l**egalább két paraméterváltozót** kell tartalmaznia. Az elsőbe a **hibatípus** (a hiba kategóriája), a másodikba a hibához kapcsolódó **hibaüzenet** szövege kerül. Ezeken kívül még három további paraméterváltozót adhatunk meg. Ha ezek szerepelnek, akkor a PHP elhelyezi bennük a hibát kiváltó kód **fájljának** nevét, az érintett **programsor** számát, valamint a hiba helyén elérhető **változókat** tartalmazó tömböt is.

```
  function
hibakezelő(hibatípus,üzenet,fájl,sor,kontextus){
     hibakezelő_utasítások
     return true|false
  }
```

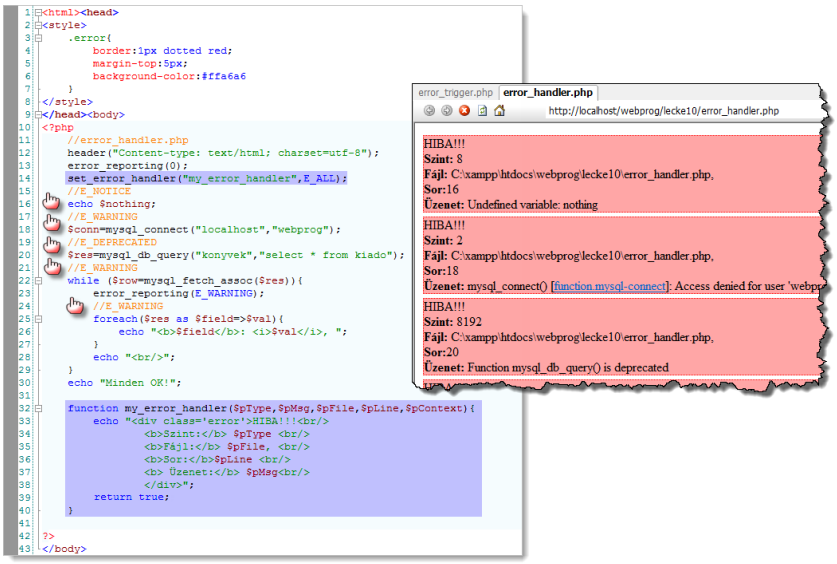
A kötött formális paraméterlista mellett a callback-függvények másik sajátossága, hogy automatikus hívásuk csak akkor kezdődik meg, ha **regisztráljuk** a függvényt. A regisztráció tudatja az értelmezővel, hogy a továbbiakban ezt a függvényt kell meghívni a megfelelő esemény bekövetkeztekor.

A hibakezelő függvény regisztrálása a **set_error_handler()** függvénnyel történik.

```
  set_error_handler("hibakezelő", [hibatípus])
```

A *függvénynév* a callback-függvény neve, a *hibatípus* pedig azoknak a hibáknak megadására szolgál, amelyek kezelését a függvénnyel akarjuk végeztetni. A hibatípusokat az **error_reporting()** függvénynél látott módon, konstansokkal lehet megadni. Az itt nem jelzett, többi hibát változatlanul a PHP fogja kezelni. Az **E_ALL** konstans alkalmazása esetén minden hiba a saját hibakezelő függvényhez kerül. A függvényben az átvett paramétereknek megfelelően reagálhatunk a hibára, majd visszatérési értékként logikai eredményt (**true** vagy **false**) adhatunk meg. A **true** sikeres, a **false** sikertelen hibakezelést jelent. Utóbbi esetben, a PHP eredeti hibakezelése lép érvénybe.

Forrás: error_handler.php

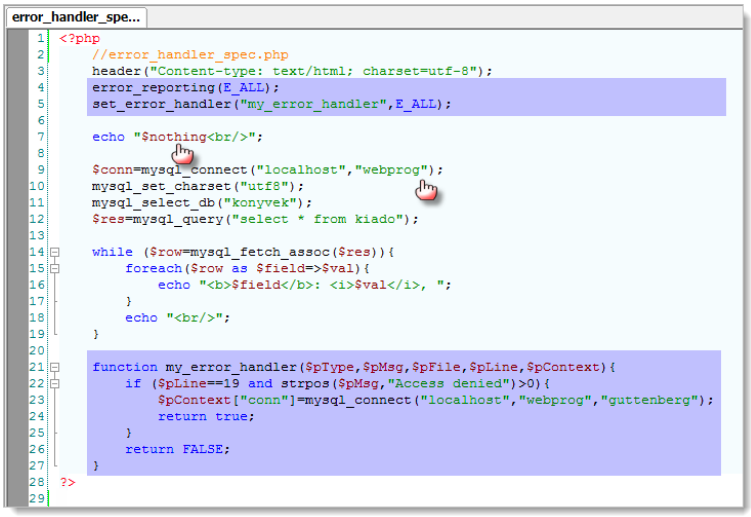


79. ábra Hibakezelő függvény

Az ábrán látható program 13. sorában letiltjuk a PHP saját hibaüzeneteit, és a 14. sorban a ***my_error_handler()*** *függvényt regisztráljuk hibakezelőként. A regisztráció az összes hibatípus megjelölésével történik (****E_ALL****), ezért az ezt követő hibák mind a hibakezelő függvényhez kerülnek. A* ***my_error_handler*** *csak demonstrációs céllal készült, ezért bár csak a hibaadatok megjelenítésével kezeli a hibákat, a 39. sorban mindig sikeresnek minősíti saját működését. Ilyenkor a PHP saját hibaüzenetei akkor sem jelennének meg, ha egyébként nem lennének letiltva.*

A második példában bekapcsolva hagytuk a PHP saját hibaüzeneteit (4. sor), de minden hiba kezelőjeként regisztráltuk a **my_error_handler()** függvényt (5. sor). A hibakezelő csak a 9. sorban bekövetkező hibákra reagál, és csak azokra, amelyek hibaüzentében szerepel az "**Access denied**" szöveg. A példa itt is állatorvosi ló, hiszen nem valószínű, hogy jelszó nélkül próbálunk egy adatbázis szerverhez kapcsolódni, és a hibakezelőtől várjuk, hogy korrigálja a hiányosságot. A példa azonban jól demonstrálja, hogyan kísérelhetjük meg a hiba javítását a hibakezelő függvényben.

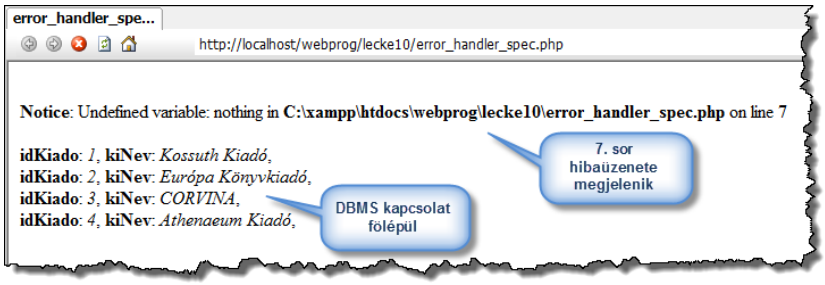
Forrás: error_handler_spec.php



80. ábra Hiba korrigálása a hibakezelőben

Ha a **`my_error_handler()`** az említett hibával találkozik, akkor a **`mysql_connect()`** paramétereit jelszóval kiegészítve megismétli a függvény hívását. A visszatérési értéket a hiba helyén elérhető változókat tartalmazó tömbön (**`$pContext`**) keresztül teszi a **`$conn`** változóba (**`$pContext["conn"]`**). Ezután **`true`** értéket visszaadva befejezi a hibakezelést. A hibakezelő sikeres végrehajtása esetén a vezérlés a hiba keletkezését követő utasításra (10. sor) kerül, így a program sikeresen folytatódhat.

A hibakezelő minden hibát elkap, de csak egyet korrigál, az összes többi esetben **`false`** értéket ad vissza. Eszerint egy kivételével minden hibát visszadob a PHP-értelmezőnek, ami pedig hibaüzenetet generál. A 7. sorban található, nemlétező változóra való hivatkozás miatt például hibaüzenet meg fog jelenni.



81. ábra Példaprogram futása

10.4.3 Hibák dobása

Az eddig látott lehetőségekkel el tudjuk érni, hogy a PHP által érzékelt hibákat saját hibakezelőnkkel dolgozzuk fel. Mivel azonban a hibakezelő callback-függvény a PHP automatikusan hívja meg, változatlanul nem tudunk mit kezdeni azokkal körülményekkel, amik a PHP számára nem jelentenek hibát, de az alkalmazásunk működését veszélyeztetik.

Az ilyen eseteket eddig a **`die()`** függvényhívásával kezeltük. Ha ezeket is a hibakezelő függvény gondjaira akarjuk bízni, akkor ki kell váltanunk a callback-függvény meghívását. A hibák dobását a **`trigger_error()`** függvénnyel végezhetjük el.

```
trigger_error(üzenet, típus)
```

A függvény *üzenet* szövegű, és *típus* hibatípusba sorolt hibát dob a PHP-nak, amivel az meghívja a hibakezelő függvényt. A hibatípusa a felhasználói hibatípusok valamelyike (**`E_USER_ERROR`**, **`E_USER_WARNING`**, **`E_USER_NOTICE`**) lehet, hiba szövege szabadon megadható.

Az ilyen hibagenerálást nevezzük hibadobásnak. A **`trigger_error()`** tehát egy hibát dob, amelyet a hibakezelő függvény elkap, és feldolgoz.

Forrás: error_trigger.php

error_trigger.php

```php
<?php
    //error_trigger.php
    error_reporting(E_ALL);
    header("Content-type: text/html; charset=utf-8");
    set_error_handler("my_error_handler",E_ALL);

    $conn=mysql_connect("localhost","webprog","guttenberg");
    mysql_set_charset("utf8");
    mysql_select_db("konyvek");

    if(!$res=mysql_query("rossz sql monadat")) trigger_error("ROSSZ SQL",E_USER_ERROR);

    while ($row=mysql_fetch_assoc($res)){
        foreach($row as $field=>$val){
            echo "<b>$field</b>: <i>$val</i>, ";
        }
        echo "<br/>";
    }

    function my_error_handler($pType,$pMsg,$pFile,$pLine,$pContext){
        switch($pMsg){
            case "ROSSZ SQL":
                if(!$pContext["res"]=mysql_query("select * from kiado")){
                    die("Itt már tényleg baj van!");
                } else {
                    echo "<h3>SQL mondat javítva!</h3>";
                    return true;
                }
            default:
                return FALSE;
        }
    }
?>
```

82. ábra Hibadobás

Az itt látható példában a hibakezelő minden hibát elkap, de csak azokra reagál amelyek szövege ”ROSSZ SQL”. Ezeket korrigálni is próbálja a **mysql_query()** ismételt meghívásával (23. sor). Siker esetén **true** eredménnyel tér vissza sikertelen próbálkozáskor végleg leállítja a programot (24. sor).

A függvény minden más hibát visszadob a PHP-nek, ezért azokat a beépített hibakezelő dolgozza föl.

10.5 KIVÉTELEK KEZELÉSE

A PHP 5 új hibakezelési technikája a kivételkezelés. A kivétel egy speciális, a programból dobott hibát leíró objektum, ami a PHP beépített **Exception** osztályának, vagy abból származtatott osztálynak a példánya.

A kivételkezelés nagy vonalakban úgy történik, hogy egy hibaobjektum létrehozásával és dobásával kiváltjuk, majd a program egy meghatározott blokkjával elkapjuk és kezeljük a hibát.

10.5.1 try…catch vezérlési szerkezet

A technika megvalósítását a speciális **try**…**catch** vezérlési szerkezet, az a hibaobjektumok **Exception** nevű osztálya és a hibákat dobó **throw()** függvény biztosítja.

A vezérlési szerkezet egy try, és tetszőleges számú catch blokkból áll.

A kivételt a try blokkban egy az Exception, vagy annak származtatott osztályához tartozó hibaobjektum létrehozásával, és a hiba dobásával válthatjuk ki. Utóbbi feladatot a throw() függvény végzi. A throw-val dobott hiba tehát mindig valamelyik hibaosztály (Exception, vagy leszármazottja) objektuma.

Az **Exception** osztály olyan adattagokkal rendelkezik, amelyek egy hiba adatait tárolják (**$message, $code, $file, $line**). Mivel láthatóságuk **protected**, elérésükhöz az osztály metódusait használhatjuk (**getMessage(), getCode(), getFile(), getLine(), getTrace()**).

Az **Exception** általános hibákat leíró osztály. A saját hibatípusok létrehozásához ebből származtatott osztályokat kell készítenünk.

Minden **catch**-blokkban – mintegy a blokk paraméterként – egy megadott hibaosztályba tartozó paraméterobjektum szerepel. A **try** blokkban dobott kivételt az a **catch** blokk kapja el, amelyiknek paraméterobjektuma a hibával azonos osztályba tartozik.

```
try {
   ellenőrzött kód
} catch (hibaosztály paraméterobjektum){
   hibakezelő_kód
}
[catch(hibaosztály
paraméterobjektum){hibakezelő_kód}...]
```

10.5.2 Saját hibaosztály létrehozása

Saját hibaosztályt származtatással, az **Exception** osztályból hozhatunk létre.

```
class SajatHibaOsztály extends Exception{
   function __construct(){
      [konstruktor_utasításai]
      parent::__construct(hibaüzenet, hibakód)
   }
}
```

A származtatott osztályban mindenképpen létre kell hoznunk a konstruktort, amiből meg kell hívnunk a szülő (**Exception**) osztály konstruktorát is.

Forrás: error_try.php

```
class WrongSql extends Exception{
    var $message="Hibás SQL mondat!";
    var $code=3268;
    function __construct(){
        parent::__construct($this->message, $this->code);
    }
}
```

83. ábra Saját hibaosztály

A fenti ábra a **WrongSql** nevű hibaosztályt hozza létre. A konstruktor nem vár paramétereket, a hiba szövegének (**$message**), és kódjának (**$code**) értéke rögzített. Mivel azonban az **Exception** osztály konstruktorának szüksége van ezekre a paraméterekre, a szülő konstruktor hívásakor átadjuk az adatokat.

10.5.3 Kivétel dobása

A kivétel dobása a hibaobjektum létrehozásával kezdődik:

```
$hiba=new Exception(”Hibaszöveg”,327858);
```

A hiba kiváltásához a **trow()** függvény paraméterként ezt az objektumot kell megadni:

```
throw($hiba);
```

Az objektumot nem feltétlenül kell változóba tenni, a függvény paramétere lehet az objektumot létrehozó kifejezés is:

```
throw(new Exception(”Hibaszöveg”,327858));
```

Ha a programunk tartalmazza a WrongSql hibaosztályt is, akkor ilyen típusú hibát az alábbi kóddal dobhatunk:

```
throw(new WrongSql());
```

Most nincs szükség az üzenet és a kód megadására, hiszen a **WrongSql**-ben ezek rögzítettek.

10.5.4 Kivétel elkapása

A try blokkban dobott hibát a catch-blokkok közül az kapja el, amelyiknek paramétere a dobott hibával azonos hibaosztály objektuma.

Forrás: try_part.php

```
<?php
    try{
        if(!$res=mysql_query("rossz sql monadat")) throw(new WrongSql());
        //...
    }
    catch(WrongSql $e){
        if(!$res=mysql_query("select * from kiado")) throw(new Exception("FATÁLIS HIBA",1));
    }
    catch(Exception $e){
        echo "Hiba: ".$e->getMessage()."<br/>";
        echo "Hibakód: ".$e->getCode()."<br/>";
        echo "Hiba sora: ".$e->getLine()."<br/>";
        echo "Hiba sora: ".var_export($e->getTrace(),true)."<br/>";
        die();
    }
?>
```

84. ábra Több catch ág

A fenti példa 3. sorában **WrongSql** típusú hibát dobunk. Ezt a 6. sorban kezdődő **catch** kapja el. Ha a programban bármilyen más kivétel keletkezne, az a 9. sorban kezdődő **catch** blokkhoz kerülne.

A **catch**-blokkok feldolgozhatják a hibát, de akár tovább is dobhatják egy újabb **throw()** utasítással. A fenti példában ez történik a 7. sorban, ha a **mysql_query()** megismételt hívása is sikertelen.

Ennek akkor van értelme, ha egymásba ágyazott kivételkezelő vezérlési szerkezeteket használunk. Ilyenkor ugyanis a beágyazott kivételkezelő szerkezet **catch**-ágában dobott hiba egy szinttel följebb kerül, és a befoglaló **try**…**catch** megfelelő **catch**-ága kapja el. Ez történik akkor is, amikor a beágyazott kivételkezelő egyik **catch**-ága sem kapja el a hibát.

Forrás: error_try.php

```
1  <?php
2      //error_try.php
3      class WrongSql extends Exception{
4          var $message="Hibás SQL mondat!";
5          var $code=3268;
6          function __construct(){
7              parent::__construct($this->message, $this->code);
8          }
9      }
10
11     header("Conten                    html; charset=utf-8");
12     try{
13         $conn=mysq                    calhost","webprog","gu
14         mysql_set_cha                 set("utf8");
15         mysql_select_db("konyvek");
16         try{
17             if(!$res=mysql_query("rossz sql monadat")) throw(new WrongSql());
18         }
19         catch(WrongSql $e){
20             if(!$res=mysql_query("select * from kiado")) throw(new Exception("FATÁLIS HIBA",1));
21         }
22         while ($row=mysql_fetch_assoc($res)){
23             foreach($row as $field=>$val){
24                 echo "<b>$field</b>: <i>$val</i>, ";
25             }
26             echo "<br/>";
27         }
28
29     }
30     catch(Exception $e){
31         echo "Hiba: ".$e->getMessage()."<br/>";
32         echo "Hibakód: ".$e->getCode()."<br/>";
33         echo "Hiba sora: ".$e->getLine()."<br/>";
34         echo "Hiba sora: ".var_export($e->getTrace(),true)."<br/>";
35         die();
36     }
37 ?>
```


85. ábra Egymásba ágyazott try…catch

A példában egy saját hibaosztály létrehozását látjuk (**WrongSQL**). Ezt követően két egymásba ágyazott **try...catch**-blokkban található a programkód. Ha a beágyazott try-blokkban nem sikerül a lekérdezés, akkor **WrongSql** típusú hibát dobunk, amit az egyetlen **catch**-ág el is kap. Ez megpróbálja a korrigálni a hibát, de ha nem jár sikerrel, továbbdobja a kivételt, amit ezután a külső **try...catch** általános hibák elkapására alkalmas **catch** ága dolgoz fel.

Fontos tudni, hogy a PHP-ben csak a `throw()` függvénnyel dobott kivételek kezelhetők. A PHP által érzékelt különböző hibatípusok nem dobnak kivételt, tehát ezek kezelésére továbbra is a kivételkezelő függvényeket használhatjuk.

10.6 ÖSSZEFOGLALÁS, KÉRDÉSEK

10.6.1 Összefoglalás

A PHP hibakezelésével foglalkozó leckénkben megtanultuk, hogy az **`error_reporting()`** függvény hívásával meghatározhatjuk a kimentre kerülő hibaüzenetek körét. A saját hibakezelést a **`set_error_handler()`** függvénnyel regisztrálható, kötött paraméterlistájú callback-függvénnyel végezhetjük el. A függvény **`true`** visszatérési értékkel jelzi az elkapott hiba sikeres feldolgozását, **`false`** eredménnyel a sikertelenséget. A hibakezelő függvénnyel visszadobott hibákat a PHP beépített hibakezelése dolgozza fel.

A hibakezelés másik lehetősége a kivételkezelés, ami azonban csak a **`throw()`** utasításokkal dobott hibák feldolgozására alkalmas. A technika az **`Exception`** hibaosztály és leszármazottainak objektumain alapul.

A **`try`** blokkban létrehozott hiba objektumokat a **`throw`** utasítással dobhatjuk. Egy **`try`** blokkhoz több **`catch`** is tartozhat. A hibaobjektumot az a **`catch`** ág kapja el, amelynek paraméterobjektuma a hibával azonos hibaosztályba tartozik. A **`try...catch`** szerkezetek egymásba ágyazhatók. A beágyazott szinten nem kezelt, vagy továbbdobott hibákat a befoglaló **`try..catch`** fogja feldolgozni.

10.6.2 Önellenőrző kérdések

1. Milyen hiányosságai vannak a die() függvénnyel megvalósított hibakezelésnek?
   - A die() a hibaüzenet kimenetre küldése után azonnal leállítja a program futását, így nem ad lehetőséget a hiba feldolgozására.
2. Hogyan tudja szabályozni a PHP saját hibaüzeneteinek megjelenését?
   - Az error_reporting() függvénnyel, amelynek paramétere a hibatípusok konstansaiból bináris vagy művelet segítségével előállítható szám.

3. Mik azok a callback-függvények?

- Olyan meghatározott formális paraméterlistával rendelkező függvények, amelyek hívását a PHP valamilyen esemény bekövetkezésekor automatikusan elvégzi. A callback-függvény paraméterként kapja meg az esemény adatait. Ebbe a kategóriába tartoznak a saját hibakezelésre alkalmas függvények is.

4. Hogyan készíthet saját kivételosztályt?

- A saját kivételosztályt az Exception osztályból kell származtatni. A származtatott hibaosztály konstruktorából mindenképpen meg kell hívni a szülő konstruktorát!

5. Melyik catch blokk kezeli le a try blokkban dobott hibákat?

- Az, amelyiknek paraméterobjektuma a dobott hibával azonos hibaosztályba tartozik. Ha nincs ilyen, akkor a hibát az Exception osztályú paraméterobjektummal rendelkező catch, vagy egymásba ágyazott kivételkezelés esetén a befoglaló try…catch megfelelő blokkja kapja el.

11. LECKE: A KLIENSOLDALI PROGRAMOZÁS ALAPJAI

11.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

Tananyagunk túlnyomó részében a weblapok szerveroldali dinamikus előállításával foglalkoztunk. Sokáig tartotta magát az az irányzat, ami szerint a webalkalmazások minden műveletét a szerveren futó programokkal kell elvégezni. A kliensoldalra csak a felület megjelenítését és a felhasználói adatbevitel feltöltését szabad bízni.

A korszerű webalkalmazásokban jelentkező új elvárások – a felhasználói adatbevitel validálása, szerverrel folyó, de a weblap újratöltését nem igénylő kommunikáció lebonyolítása, a felhasználói élmény emelése stb. – teljesítése érdekében azonban egyre hangsúlyosabbá, napjainkra pedig megkerülhetetlenné vált a kliensoldali programozás.

Míg a szerveroldalon futó programok esetében a PHP nyelv csak dominál, a weblapok kliens oldali dinamizálásában a JavaScript gyakorlatilag egyeduralkodó.

Bár a JavaScript-programozás alapjai viszonylag egyszerűek – annál is inkább, mert a PHP szintaktikája is nagyon hasonló, – a nyelv ismeretanyaga önálló tananyagot igényelne. Most következő leckénkben éppen ezért messze a teljesség igénye nélkül, néhány példa kiemelésével pillantunk be a JavaScript használatában rejlő lehetőségekbe.

A leckében olvashat a nyelv szintaktikájáról, a kód weblapokba integrálásnak módszereiről, és futásának módjáról. Megtanulhatja, hogyan kötheti eseményekhez a JavaScript függvények futtatását, hogyan végezhet alapvető kliensoldali validálást, és bepillantást nyerhet a kliens és a szerver közötti XML-alapú aszinkron kommunikációjába.

11.2 A JAVASCRIPT

A JavaScript interpretált, weblapok kliensoldali programozására kifejlesztett nyelv. A programkód a szerverről letöltődő weblapokba ágyazható, vagy azokhoz csatolható. Értelmezését és futtatását a böngészőkbe épített JavaScript-motor végzi. Bár elvétve találkozhatunk olyan webkliensekkel is, amelyek nem támogatják használatát, a modern böngészők mindegyikét ellátják JavaScript-motorral.

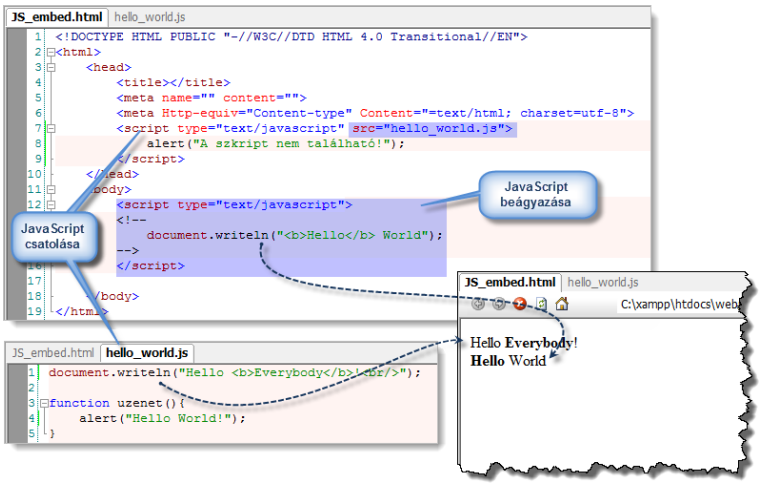
A nyelv megalkotója, Brendan Eich a Netscape munkatársaként, 1995-ban fejlesztette ki az akkor még Mocha, később LiveScript névre keresztelt nyelvet, ami a Netscape Navigator 2.0 1996-ban megjelenő végleges verziójába már JavaScript néven került be. A JavaScript 1997 óta szabványos ECMA-nyelv, amit ECMA-262 szabvány ír le, 1998-ban pedig az ISO is szabványosította (ISO 16262).

Az évek során roppant népszerűvé vált, és a nagy böngészőgyártók mindegyike implementálta valamelyik verzióját. A JavaScript jelenleg legújabb 2.0 változatát még egy böngésző sem támogatja, de az Internet Explorer 9 az 1.3, a Crome 20 az 1.7, a FireFox 14 pedig a JavaScript 1.9 verzióban készült szkriptek feldolgozására képes.

11.3 JAVASCRIPT KÓD BEILLESZTÉSE A WEBLAPBA

A JavaScript-kódot a HTML **<SCRIPT></SCRIPT>** elemével helyezhetjük el a web dokumentumban. Az utasításokat közvetlenül beágyazhatjuk a jelölő nyitó és záró eleme közé, de külső fájlban elhelyezhet kódot is csatolhatunk. Az első esetben a **<SCRIPT>** jelölők közé gépeljük a programsorokat, a másodikban (általában **js** kiterjesztésű) külső fájlban helyezzük el programot, és a **<SCRIPT>** jelölő **SRC** attribútumával adjuk meg a fájl útvonalát. Ilyenkor is gépelhetünk utasításokat a jelölőpárba, de ezek csak akkor fognak futni, ha a csatolt fájl nem található.

Forrás: JS_embed.html; hello_world.js



86. ábra Csatolt és beágyazott JavaScript

A nyelv rendkívül sokoldalú lehetőségeket biztosít a weblapok manipulálására. A bővebb ismertetésig csak a dokumentum tartalom dinamikus létrehozásának legegyszerűbb módját, a **document.writeln()**, és az **alert()** utasításokat használjuk példáinkban. A **document.writeln()** a weblapba írja a paramétereként megadott HTML-szöveget, az **alert()** pedig előugró üzenetablakban jeleníti meg paraméterét.

A **<SCRIPT>** jelölőket elhelyezhetjük a **HEAD** és a **BODY** tagben is. A **HEAD**-be ágyazott kód az oldal előállításának első mozzanataként fut le, a **BODY**-ban elhelyezett utasítások azonban a beágyazás helyén fejtik ki hatásukat.

Ez az oka annak, hogy a példában előbb jelent meg a **Hello Everybody!** szöveg, mint a **Hello World!**

A JS úgynevezett objektum központú nyelv, de a „hagyományos" procedurális programozásra is alkalmas. A kód programtörzsre és alprogramokra osztható.

Bár a **BODY**-ban elhelyezett JavaScript-kóddal közvetlenül manipulálhatjuk az oldalt, a gyakorlat általában mégis az, hogy külső állományba illesztjük és a **HEAD**-ben csatoljuk a szkriptet. Ilyenkor a programtörzsben lévő utasítások a **BODY** renderelésének első lépéseként futnak le, az alprogramok azonban csak hívásra indulnak el.

Ez az oka annak, hogy az ábra csatolt állományának **uzenet()** függvénye nem indult el.

A **<SCRIPT>** jelölőt nem ismerő böngészők figyelmen kívül hagyják a tagot és megjelenítik a weblapon beágyazott JavaScript kódot. Ennek elkerülése érdekében tanácsos a **<SCRIPT>**-en belül elhelyezett utasításokat HTML-megjegyzésként feltűntetni.

```
<!--
   JS kód
//-->
```

11.4 A JAVASCRIPT SZINTAKTIKÁJA

A JavaScript ECMA-szabványú nyelv. A szabványt, illetve annak elemeit több más nyelvben átvették, így szintaktikai szabályai számos esetben visszaköszönnek. A PHP alapvető szintaxisa például nagyon hasonlít a JavaScript-éhez.

Az utasításokat pontosvesszővel zárjuk, vagy külön sorokba írjuk. Nem kötelező, ugyan, de általában mindkettőt betartjuk.

A JavaScript a PHP-val szemben érzékeny a kis és nagybetűk közötti különbségekre. Ez érvényes a különböző azonosítókra, és az utasításokra is. A helytelenül írt utasításokat az értelmező nem tudja feldolgozni és hibaüzenetet küld.

A programkód építőelemei a literálok, változók, kifejezések, vezérlési szerkezetek, függvény- és objektumdeklarációk.

11.4.1 Literálok

A programkódban közvetlenül elhelyezett értékek a PHP-hoz hasonlóan egész és valós számok, szövegek, és logikai értékek lehetnek. A logikai literálok a **`true`**, illetve a **`false`** kulcsszavakkal írhatók le.

Az egész számok arab számjegyeket és előjelet tartalmazhatnak, a valós számok ezen kívül tizedes elválasztót. Az egészek lehetnek decimálisak, oktálisak és hexadecimálisak. A valós számokat exponenciális alakban is ábrázolhatjuk.

Forrás: literals.html

literals.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <head>
        <title></title>
        <meta name="" content="">
        <meta Http-equiv="Content-type" Content="=text/html; charset=utf-8">
    </head>
    <body>
    <script type="text/javascript">
        <!--
            //Egész
            document.writeln(111);document.writeln("<br/>");
            //Oktális egész
            document.writeln(0111);document.writeln("<br/>");
            //Hexadecimális egész
            document.writeln(0x111);document.writeln("<br/>");
            //Valós
            document.writeln(1.11);document.writeln("<br/>");
            //Valós exponenciális alakban
            document.writeln(111e-2);document.writeln("<br/>");
            //Szöveg
            document.writeln('Hello"World!"');document.writeln("<br/>");
            //Logikai
            document.writeln(true);document.writeln("<br/>");
            //NULL
            document.writeln(null);document.writeln("<br/>");
        //-->
    </script>
    </body>
</html>
```

87. ábra Literálok

A szövegek páratlan (' '), vagy páros (" ") idézőjelek közé zárt tetszőleges karaktersorozatok. A szövegben elhelyezett escape jellel (\) bevezetett speciális karaktereket értelmezi és megfelelően helyettesíti a JavaScript.

```
\b - visszatörlés
\f - lapdobás
\r – kocsi vissza
\n - új sor
\\ - escape \
```

Ha szövegben szöveghatároló karaktert akarunk elhelyezni, akkor escape-karakterrel kell bevezetni őket, de azt is megtehetjük, hogy egy szöveghatárolón belül a másik szöveghatárolót használjuk.

```
\" – páros aposztróf "
\' – páratlan aposztróf '
```

```
"<p align=\"left\">…</p>"
"<p align='left'>…</p>"
```

11.4.2 Változók

A JavaScript a PHP-hez hasonlóan gyengén típusos. Megkülönbözteti a szám, szöveg, logikai és objektum típusokat, azonban a változók implicit módon deklarálhatók, azaz típusokat nem kell megadni a létrehozáskor. A típust a változó értéke, illetve felhasználásnak környezete határozza meg.

A változók neve aláhúzás (_) jellel, vagy betűvel kezdődhet, majd betűvel vagy számmal folytatódhat. Az azonosítókban az 1.3 verziótól kezdve Unicode karaktereket is használhatunk.

A deklarálás lehet egyszerű érték-hozzárendelés, ami előtt opcionálisan szerepeltethetjük a **`var`** kulcsszót. Ez nem kötelező, azonban erősen ajánlott, mert befolyásolja a változók érvényességi körét, és alkalmas arra is, hogy értékadás nélkül deklaráljunk változót. A deklarált, de értéket nem tartalmazó változók speciális, **`undefined`** értékkel rendelkeznek.

Forrás: variables.html

literals.html variables.html

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
2  <html>
3      <head>
4          <title></title>
5          <meta name="" content="">
6          <meta Http-equiv="Content-type" Content="=text/html; charset=utf-8">
7      </head>
8      <body>
9          <script type="text/javascript">
10         <!--
11             a=123;
12             var b=321;
13             var c="Hello World";
14             var msg=new String("Hello Everybody");
15             var igaz_e=false
16             function uzenet(){
17                 var msg="Hello World!";
18                 alert(msg);
19             }
20         //-->
21         </script>
22     </body>
23 </html>
```

Globális

Lokális

88. ábra Változók létrehozása

A főprogramban deklarált változók globálisak lesznek, elérhetők a program bármely környezetében.

Az alprogramokban deklarált változók lokálisak. Ha a függvényben olyan változót használunk, amely már globálisan létezik, akkor az alprogramban csak a **`var`** kulcsszóval deklarált változók lesznek lokálisak. A JavaScript hibaüzenetet küld, ha nem deklarált változóra hivatkozunk.

11.4.3 Objektum

A nyelv speciális típusa az adattagokat és metódusokat egységbe záró adatszerkezet, az objektum. A JavaScriptben valójában minden adat objektum típusú, még a függvények is. Az objektumok hierarchikusan kapcsolódnak egymáshoz, egy gyermekobjektum örökli a szülő metódusait és tulajdonságait. Mivel a tulajdonságok objektum típusúak is lehetnek, az egészen összetett adatszerkezetek is leírhatók.

Az objektumok rendszere némileg hasonló a PHP OOP-lehetőségeihez, azonban a JavaScript-ben nem lehet osztályokat létrehozni, hanem magukból az objektumokból tudunk újabb objektumokat származtatni. A származtatott objektumok felülírhatják illetve speciális adattagokkal és metódusokkal egészíthetik ki a szülő jellemzőit. Új objektumváltozót a **`new`** kulcsszóval és az objektum konstruktorának meghívásával hozhatunk létre. A konstruktor az a függvény, ami az objektum adattagjainak és metódusainak meghatározására szolgál.

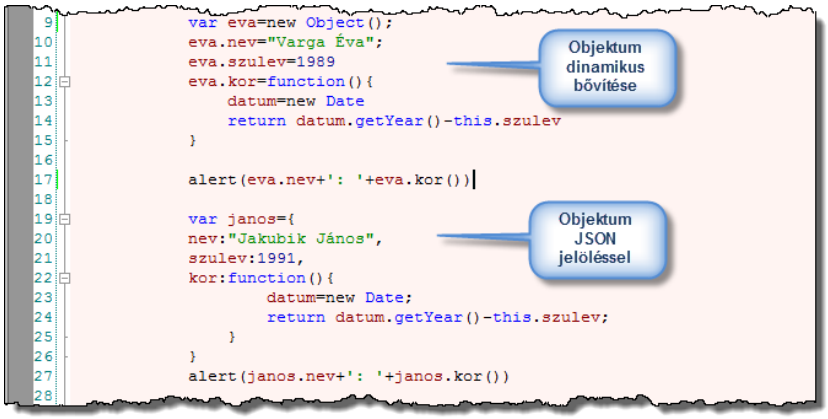
A legáltalánosabb objektum az **Object**, minden további objektum ebből származik. Speciális beépített objektumtípusok még a **Boolean**, **String**, **Number**, az **Array**, **Date** a **Function** és a **Global**. A Global objektum azokat a metódusokat és tulajdonságokat tárolja, amelyek minden kontextusban rendelkezésre állnak és hívásuk objektumnév nélkül is lehetséges.

Az Object objektumból származó új objektumot az alábbi formában hozhatunk létre.

```
var ures_objektum = new Object()
```

Az üres objektum dinamikusan bővíthető adattagokkal és metódusokkal is.

Forrás: JS_objects.html




```
var eva=new Object();
eva.nev="Varga Éva";
eva.szulev=1989
eva.kor=function(){
    datum=new Date
    return datum.getYear()-this.szulev
}

alert(eva.nev+': '+eva.kor())

var janos={
nev:"Jakubik János",
szulev:1991,
kor:function(){
        datum=new Date;
        return datum.getYear()-this.szulev;
    }
}
alert(janos.nev+': '+janos.kor())
```

89. ábra Objektumok létrehozása

A JavaScript objektumai az úgynevezett JSON-jelöléssel (JavaScript Object Notation) literálként is leírhatók. Ilyekor kapcsos zárójelek között, vesszőkkel elválasztva soroljuk fel objektum adattagjainak és metódusainak neveit, az azonosítóktól kettősponttal választjuk el az értékeiket.

11.4.4 Szöveg

Szöveg típusú objektum a szöveg objektum konstruktorával állatható elő:

```
var ures_szoveg= new String();
var szoveg_ertekkel=new String("Hello World")
```

Az objektumok metódusai és adattagjai pontszintaxissal érhetők el. A **String** típusú objektumok adattagja például a **length**, ami a szöveg karakterekben számolt hosszát tartalmazza, metódus a **toUpperCase()**, ami pedig a szöveg nagybetűs változatát adja visszatérési értékként.

```
alert (szoveg_ertekkel.toUpperCase())
//HELLO WORLD
```

11.4.5 Tömbök

A tömb a JavaScript speciális objektuma, amelynek létrehozása a tömb objektum konstruktorának meghívásával történik:

```
var a=new  Array()
```

A tömbök nulla bázisúak, dinamikusak, inhomogének, lehetnek numerikusak és asszociatívak , sőt egyszerre numerikusak és asszociatívak is.

```
t1=new Array();
t1["nev"]="Béla";
t1[2]=1962
```

Mivel a tömbök objektumok, szintén ábrázolhatók JSON-jelöléssel.

```
t1= {nev:"Béla",szulev:1962};
```

11.4.6 Kifejezések

Az adatokkal végzett műveletek leírásának módja JavaScript-ben is a kifejezés. A nyelv operátorai alapvetően megegyeznek a PHP-nál tanultakkal. Fontos különbség azonban, hogy a szöveg konkatenációjának operátora nem a pont (.) hanem a plusz (+) jel, továbbá, hogy a logikai **és** illetve **vagy** művelet csak **&&** és **||** karakterekkel írható le. Az **and** illetve **or** operátorok nem használhatók.

11.4.7 Vezérlési szerkezetek:

A JavaScript vezérlési szerkezetei nagyon hasonlítanak a PHP-ben tanultakhoz. Azonos módon használható az **if...else** és a **switch** , a **while**,

do...while, és a **for** szerkezet. A jelenleg használt verziókban nincs azonban **foreach** utasítás. Az objektumok, és tömbök bejárására ehelyett használható a **for(***ciklusváltozó* **in** *objektum***){...}** szerkezet, amely egy tömb összes elemének bejárását lehetővé teszi. A ciklus minden ismétléskor a ciklusváltozóba teszi az aktuális elem **indexét** vagy **kulcsát.**

```
t1= {nev:"Béla",szulev:1962};

for(elem in t1){
  document.writeln(t1[elem]+"<br/>")
}
//Béla
//1962
```

11.4.8 Függvények

A JavaScript a függvények deklarálásban is hasonlít a PHP-hez, azonban a formális paraméterlista megsértése nem okoz hibát.

```
function mai_nap(){
  var dt =new Date()
  var dtStr=dt.toLocaleDateString()
  return dtStr
}
document.writeln(mai_nap("Nem várt paraméter"))
//2012. június 28
```

11.5 A DOM

A JavaScript tehát objektumalapú nyelv, amelynek elsődleges célja a weblapot fölépítő objektumok manipulálása. Önmagában azonban nem rendelkezik olyan objektumtípusokkal, amelyek a weblapokon megjelenő elemeknek felelnének meg. Éppen azért már a Netscape megalkotta és már böngészőjének 2.0 verzióban bevezette az úgynevezett DOM modellt (Document Object Modell).

A DOM a böngészőablak, és az abba tölthető tartalom összes lehetséges objektumát, az objektumok tulajdonságait és metódusait, valamit hierarchikus kapcsolatrendszerüket leíró modell. A modell alapjául szolgál a JavaScriptbe integrált DOM API-nak, programozói interfésznek, aminek segítségével programjainkban azonosíthatjuk és manipulálhatjuk a böngészőablak tartalmát fölépítő objektumokat.

Bár a DOM – a JavaScript-tel szemben – sosem vált szabvánnyá, szinte minden böngészőgyártó átvette. Ugyanakkor mindannyian kisebb-nagyobb

eltéréseket tartalmazó modellt alakítottak ki, és saját böngészőjük DOM API-ját ennek megfelelően implementálták. Ez vezetett oda, hogy a JavaScript szabványos volta ellenére nagyon nehézzé vált a böngészőfüggetlen JavaScript programok írása.

A W3C éppen ezért kifejlesztette és 1998-ban hivatalos ajánlássá tette az saját DOM Level 1 modelljét, majd 2000-ben a DOM Level 2-t.

A böngészőgyártók az inkompatibilitások okozta problémák súlyát lassacskán belátva hajlani látszanak a W3C ajánlásainak támogatására. A böngészőjükbe integrált DOM API-t egyre inkább a W3C modelljéhez igazítják, így az eltérő implementációk egyre kevesebb problémát okoznak.

A W3C modellje több részre, al-modellre tagolódik. A Core DOM a dokumentumok alapvető szerkezeti elemeit, a HTML DOM, az XML DOM, és a CSS DOM a böngészőkkel kezelt különböző dokumentumtípusok objektumait modellezik

11.5.1 Néhány fontos DOM-objektum

A DOM összességében rengeteg objektumot, metódust és tulajdonságot tesz elérhetővé. Az alábbiakban kiemelünk néhány fontosabb objektumot majd gyakorlati példákon keresztül mutatunk be néhányat.

- **`Navigator:`** a böngészőt leíró tulajdonságokat tartalmazó objektum.
- **`Screen:`** a teljes látható böngészőablak kezelését biztosítja.
- **`Window:`** egy-egy frame jellemzőihez enged hozzáférni.
- **`History:`** a meglátogatott oldalak közti mozgás programozását biztosítja
- **`Location:`** az éppen letöltött címtől tartalmaz információkat, de metódusaival lehetővé teszi más oldalak letöltését is.
- **`Document:`** A weblapok programozása szempontjából a legfontosabb objektum, ami a letöltött weblap összes lehetséges elemét fogja össze.

11.5.2 Document

A Document típusú objektumon keresztül elérhető a weblap összes többi eleme, de számos metódussal is rendelkezik. Ezek közül egyet már használtunk is. Ez pedig a **`writeln()`** ami a letöltött dokumentumba írja a paraméterét.

```
document.writeln("Hello World");
```

Az olvasó minden bizonnyal észrevette, hogy a szövegben nagybetűs Document, szót a példákban kis d-vel írjuk. A magyarázat az, hogy a DOM nagybetűkkel jelzi a modell alkotóelemeinek neveit. A JavaScript-ben az objektumpéldányokkal dolgozunk, amelyek neveit kisbetűkkel szokás írni.

Szintén a **document** objektum metódusa a **write()**, a **getElemetnsByName()**, a **getElementyByTagName()**, és a **getElementbyId()**

A DOM teljes, de még érintőleges áttekintésére sincs lehetőségünk, ezért csak néhány olyan objektumát sorolunk fel, amiket a következő példákban használni fogunk. Az alábbi címen megtalálja a W3C oktató célú, de meglehetősen részletes DOM-leírását.

http://www.w3schools.com/jsref/default.asp

13. link A DOM áttekinthető referenciája példákkal

A document a DOM egyik legfontosabb objektuma, ami a letöltött weblap elemeinek megfelelő objektumokat tartalmazza. Az elemek mindegyike egy **element** objektum. Az **element** objektumokat típusuknak megfelelő elemcsoportokba rendezi a DOM. A csoportokat szintén objektumokkal írja le.

A **forms** objektum például a dokumentumban található összes űrlap jelölőket tartalmazó objektum. Az **images** a weblap képeit fogja össze, **links** hivatkozások egyszerű elérést biztosítja.

Elemek kiválasztása

Minden elem két azonosításra az elem kiválasztására alkalmas attribútummal rendelkezik Az **id** értékének a teljes dokumentumban, a **name** szövegének az elemet tartalmazó objektumon belül kell egyedinek lennie.

Az alábbi sor a **name** attribútum értékét használva a dokumentum összes űrlapja közül az "**urlap**" nevű elemet, annak "**vezetek**" nevű beágyazott

elemét, illetve az elem pillanatnyi értékét választja ki, és helyezi a **vnev** változóba.

```
var vnev=document.forms["urlap"]["vezetek"].value;
```

Természetesen csak akkor működik helyesen, ha a dokumentumban valóban megvannak a megfelelő elemek.

```
<form name="urlap" method="GET" >
  <input name="vezetek" type="text" />
  <input type="submit" value="Elküld" />
</form>
```

Az elemek kiválasztásának másik technikája az **id** attribútumra épül.

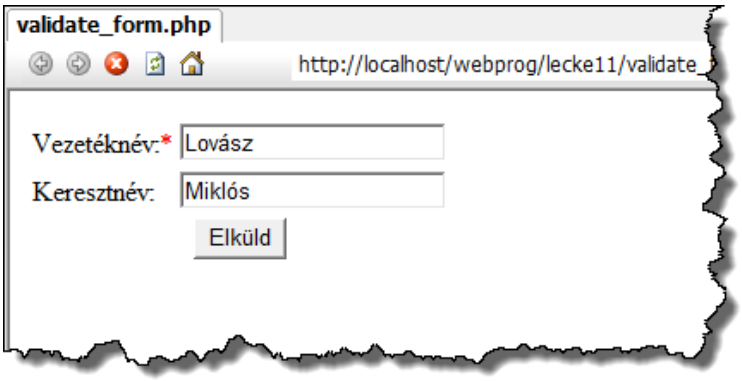
A **document** objektum **getElementById()** metódusa a teljes dokumentum elemeit figyelembe véve kiválasztja azt az egyet, amelynek **id** attribútuma megegyezik a metódus paraméterében átadott értékkel.

```
var elm=document.getElementById("button1")
```

Elemek manipulálása

Az elemeket azért választjuk ki, hogy megvizsgáljuk, vagy megváltoztassuk tulajdonságaikat. Az űrlapokon található beviteli elemek aktuális értékei az **value** tulajdonságokban vannak. Ez lekérdezhető, és megváltoztatható, olvasható/írható tulajdonság.

Forrás: validate_form.php



90. ábra Űrlap elemei

```
var vnev=document.forms["urlap"]["vezetek"].value;
//Lovász
```

Az **innerHTML** tulajdonság az elembe ágyazott HTML-tartalmat jelenti. Szintén írható-olvasható tulajdonság. Ez teszi lehetővé, hogy a dokumentum egy adott elemén belüli tartalmat, azaz a weblap egy meghatározott részét megváltoztassuk.

```
var part=document.getElementById("valaszdiv");
part.innerHTML ="<b>Új tartalom!</b>";
```

A **sytle** tulajdonság szintén az elemek egyik jellemzője. Önmaga is objektum, amin keresztül elérjük az elem **style** attribútumában megadható formátumok egy részét. Használhatósága ugyan nem terjed ki az összes CSS formátumra, de mivel az objektumon elérhető tulajdonságok olvashatók és írhatók, látványos vizuális hatások keltésére alkalmas.

```
var gomb=document.getElementById("gomb1");
gomb.style.backgroundColor="#7a8cf5";
```

11.6 DOM-PÉLDÁK

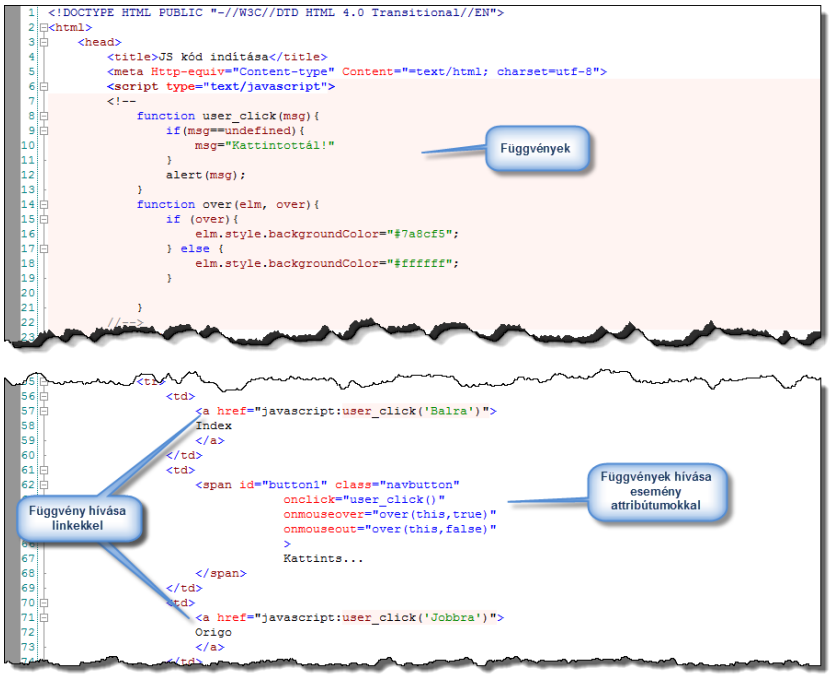
A következőkben a DOM alkalmazásának három példáját mutatjuk be. A példaprogramok mind megtalálgató a lecke mappájában.

11.6.1 JavaScript program interaktív indítása

Forrás: startJS.html

Láttuk, hogy a **BODY**-ba beágyazott JavaScript-kódok azonnal lefutnak, amikor a böngésző a weblap adott részét állítja elő. Ez persze távolról sem biztosítja azt az interaktivitást, amit egy dinamikus weblaptól várunk, hiszen ehhez arra lenne szükség, hogy valamilyen felhasználói beavatkozás indítsa el a programot.

Erre kétféle lehetőség van. Az egyik a speciális hivatkozás, link, a másik az elemekhez rendelhető esemény attribútumok használata.



91. ábra Függvények hívásai

A JavaScript-kód hivatkozással való indítását az teszi lehetővé, hogy a linkek **HREF** attribútumában **URL**-cím helyett **a javascript:** előtaggal kezdett, utasításokat, így paramétereket átadó függvényhívásokat is elhelyezzük. A hívott függvénynek természetesen beágyazott, vagy csatolt kódban létezni kell az oldalon.

Az alábbi hivatkozás elindítja a **user_click()** függvényt és paraméterként átadja a 'Balra' szót.

```
<a href="javascript:user_click('Balra')">
```

A szkriptek linkekkel való indításánál jóval kifinomultabb lehetőségeket biztosítanak a HTML 4-ben bevezetett esemény attribútumok. A HTML 4 óta a jelölők attribútumai esemény attribútumokkal egészültek ki. Az esemény attribútumok értéke tetszőleges JavaScript-kód lehet, de általában ide is függvényhívásokat teszünk. A kódok akkor futnak le, amikor az attribútum nevével jelzett esemény bekövetkezik. A leggyakoribb és szinte minden elem esetében használható esemény attribútumokat az alábbi táblázatok tartalmazzák:

Billentyűzet események		
Elemek	**Attribútum**	**Esemény**
Minden elem, kivéve: base, bdo, br, frame, frameset, head, html, iframe, meta, param, script, style, title	onkeydown	Billentyűlenyomására bekövetkező esemény.
	onkeypress	Billentyűlenyomás-fölengedésre bekövetkező esemény
	onkeyup	Billentyűfölengedésére bekövetkező esemény

Egér események		
Elemek	**Attribútum**	**Esemény**
Minden elem, kivéve: base, bdo, br, frame, frameset, head, html, iframe, meta, param, script, style, title	onclick	Egérkattintásra bekövetkező esemény
	ondblclick	Dupla egérkattintásra bekövetkező esemény
	onmousedown	Egérgomb lenyomására bekövetkező esemény
	onmousemove	Egérmutató mozgatására bekövetkező esemény
	onmouseout	Egérmutató elemről való elmozdítására bekövetkező esemény
	onmouseover	Egérmutató elemre mozdítására bekövetkező esemény
	onmouseup	Egérgomb fölengedésére bekövetkező esemény

A következő példa azt mutatja, hogyan rendelhetünk JavaScript-kódot egy egyszerű **SPAN** elemen bekövetkező eseményekhez.

```
<span class="navbutton"
       onclick="user_click()"
       onmouseover="over(this,true)"
       onmouseout="over(this,false)">
       Kattints...
</span>
```

A fenti példa a **** elem három esemény attribútumát is használja. Az **onclick** esemény – az előbb látott hivatkozásokhoz hasonlóan – a **user_click()** függvényt indítja el, de nem ad át paramétert.

Emlékezzük vissza, hogy a JavaScript esetében az aktuális paraméterlistának nem kell megfelelnie a formális paraméterlista tartalmának!

Az **onmouseover** és az **onmouseout** attribútumok egyaránt az **over()** függvényt hívják. A **this** kulcsszó az adott objektum, tehát ebben az esetben a kérdéses **SPAN** elem hivatkozása, a második paraméter pedig logikai érték. A két eseményt (**onmouseover/onmouseout**) tehát ugyanaz a függvény (**over()**) dolgozza föl. Ez első paraméter tudatja a függvénnyel, hogy a weblap melyik elemén történt az akció – így a függvény akár más elemeket is tud kezelni – a második pedig azt, hogy rá, vagy lehúztuk-e az egeret.

Az over függvény első paramétere egy elem hivatkozása (ezt küldjük el híváskor a this kulcsszóval), a második azt hivatott jelezni, hogy rá, vagy lehúzták az egeret az elemről.

Említettük, hogy a DOM nemcsak elemek, hanem azok stílusának elérését is biztosítja. Ezt használja ki a függvény azzal, hogy az elem hivatkozását tartalmazó **elm** paraméter **style** attribútumán keresztül szabályozza az elem háttérszínét. A szín attól függően fog változni, hogy ráhúztuk az egeret az elemre, vagy éppen eltávolítottuk róla.

```
<script type="text/javascript">
<!--
   function user_click(msg){
      if(msg==undefined){
         msg="Kattintottál!"
      }
      alert(msg);
   }
   function over(elm, over){
      if (over){
         elm.style.backgroundColor="#7a8cf5";
      } else {
         elm.style.backgroundColor="#ffffff";
      }

   }
//-->
</script>
```

A linkekkel és a **SPAN** elem **onclick** eseményével a **user_click()** függvényt indítjuk. A linkek hívásai adnak át paramétert, az **onclick** eseményben lévő függvényhívás nem. A függvényben ezért elsőként azt vizsgáljuk meg, hogy van-e értéke az **msg** paraméternek. Ha nincs, akkor a „Kattintottál!” értéket állítunk be, majd üzenetben megjelenítjük az **msg** változó értékét.

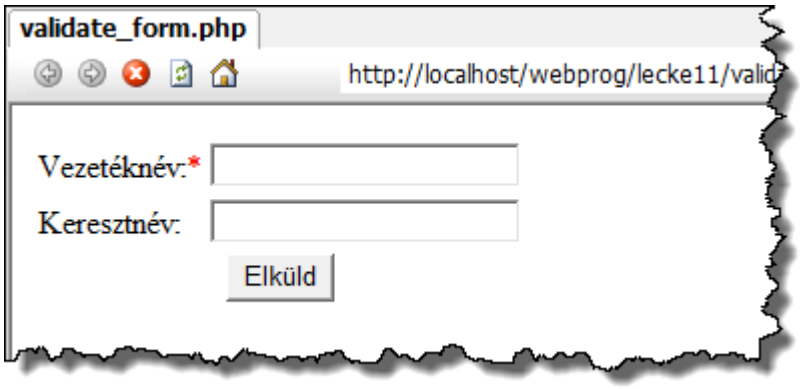
```
function user_click(msg){
     if(msg==undefined){
        msg="Kattintottál!"
     }
     alert(msg);
   }
```

11.6.2 Űrlap validálása

Forrás: validate_form.php

A DOM használatának egyik leggyakoribb esete, amikor egy kitöltött űrlap elemeinek értékét akarjuk ellenőrizni, mielőtt az azokat a szerver oldalon futó szkriptenek továbbítjuk.

Az alábbi példában egy űrlap egyik mezőjét kötelezően kitöltendőként jelöljük meg, és az űrlap elküldése előtt ellenőrizzük a helyes kitöltést.



92. ábra Validálandó űrlap

```
  <form name="urlap" method="GET"
       onsubmit="return form_control()">
    <table>
      <tr>
        <td>Vezetéknév:<span
class="required">*</span></td>
        <td><input name="vezetek" type="text" /></td>
      </tr>
      <tr>
        <td>Keresztnév:</td>
        <td><input name="kereszt" type="text" /></td>
      </tr>
      <tr>
        <td colspan="2" class="buttonbar">
          <input type="submit" value="Elküld" />
        </td>
      </tr>
    </table>
  </form>
```

A **validate_form.php** -vel előállított oldal űrlapja két szövegmezőt jelenít meg (**"vezetek"**, **"kereszt"**) amiknek értékeit az oldalt előállító szkriptnek (**validate_form.php**) küldi el. A **form onsubmit** esemény attribútuma akkor fut le, amikor a **submit** gombra kattintunk, illetve, mielőtt az űrlap adatait elküldené a böngésző. Ha az eseményben megadott JavaScript kód **true** eredménnyel zárul, akkor a küldés megtörténik, ellenkező esetben a böngésző megszakítja az adatok feltöltését.

A példában a **return form_control()** azt jelenti, hogy az esemény a **form_control()** függvény visszatérési értékével zárul.

```
  <!--
     function form_control(){
       var
vnev=document.forms["urlap"]["vezetek"].value;
       if (vnev==null || vnev=="")
       {
         alert("A vezetéknév kitöltése kötelező!");
         return false;
       }
       return true
     }
  //-->
  </script>
```

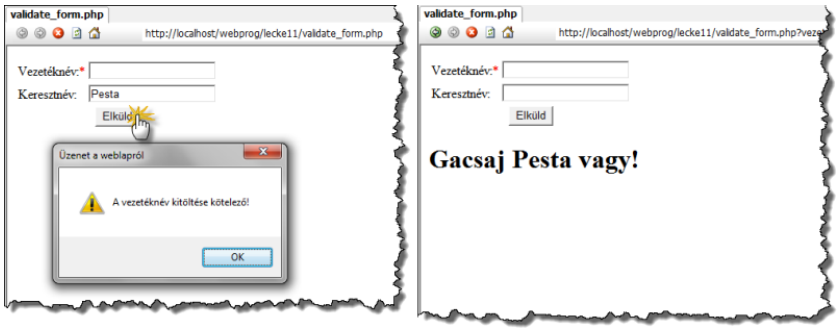
A **form_control()** függvény a DOM lehetőségeit használva azonosítja a dokumentum "**urlap**" nevű űrlap objektumának "**vezetek**" nevű elemét és értékét a **vnev** változóba teszi.

Ha az érték nem létezik (**null**), vagy üres szöveg (**""**) akkor üzenetet jelenít meg, és **false** eredményt ad vissza. Ellenkező esetben **true** a visszatérési érték.

Az űrlap a weblapot létrehozó szerver oldali szkriptnek küldi el az adatokat tehát a példában megtaláljuk a feldolgozást végző egyszerű php-szkriptet is.

A **validate_form.php** újraküldi az egész oldalt, de ha volt feltöltött „**vezetek**” nevű változó, akkor egy abból összeállított üzenetet is beszúrja a weblap alsó részébe.

```
  <?php
     if (isset($_GET["vezetek"])){
       $vnev=$_GET["vezetek"];
       if (isset($_GET["kereszt"])){
         $knev=$_GET["kereszt"];
       }else {
         $knev="";
       }
       echo "<h1>$vnev $knev vagy!</h1>";
     }
  ?>
```



93. ábra Hibaüzenet, és helyes kitöltés után visszakapott oldal

11.6.3 AJAX

Többször említettük már, hogy a WWW fundamentumát képező HTTP protokoll nagyszerűségét éppen egyszerűségének köszönheti. Ugyanakkor állapotmentessége, és a kérés-válasz alapú kommunikáció hosszú ideig korlátokat jelentett a web alkalmazások fejlesztésében.

Az állapotmentesség problémájának kezelésével már megismerkedtünk. A kérés-válasz alapú kommunikáció nem csak azt jelenti, hogy a kliens megszólítja a szervert, az pedig válaszol, hanem azt is, hogy a kliens mindig egy teljes dokumentumként kezeli a választ. Ha ez egy weblap, akkor a böngészőablak addigi tartalma törlődik, majd megjelenik a szerver által küldött oldal.

Ha bármilyen felhasználói interakció (kattintás, gombnyomás, szövegbevitel...) szerver oldali feldolgozást igényel, akkor a szerver a kliens kérésében elküldött adatra csak teljes weblap visszaküldésével tud válaszolni. Elképzelhető, hogy a felhasználó tevékenysége nyomán csupán a weblap kicsiny részletének kell megváltoznia, mégis az egész oldal újratöltődik.

Ezt problémát küszöböli ki az úgynevezett **AJAX** (Asynchronous JavaScript and XML) technika, amely lehetővé teszi a HTTP alapú kliens-szerver kommunikációt weblap egészének újratöltése nélkül.

Az AJAX lehetőségeit az XML DOM-ban megfogalmazott **`XMLHttpRequest`** objektum biztosítja. Az **`XMLHttpRequest`** HTTP-kérések küldésére és a válaszok fogadására alkalmas, JavaScript-tel vezérelhető objektum. Működése viszonylag egyszerű.

Ha az AJAX technikájával akarunk kommunikálni a szerverrel, akkor elsőként létre kell hozni egy **`XMLHttpRequest`** típusú objektumot. Ezzel fogjuk elküldeni a szervernek a HTTP-kérést, és ezzel fogadjuk a választ.

Az objektumhoz hozzá kell rendelni egy callback-függvényt, ami akkor fog elindulni, amikor a szerver válasza megérkezik.

Az objektumnak meg kell adni a kérés tulajdonságait (URL, metódus, fejléc adatok...) majd utasítani kell az objektumot, a kérés elküldésére.

A kommunikáció lehet szinkron és aszinkron típusú. Szinkron üzenetváltás alatt a böngésző felfüggeszti a szkript futását, és a folytatással vár a válasz megérkezéséig. Amikor a válasz megérkezik a továbbfutó szkript feldolgozza a szerver üzenetét.

Az aszinkron kommunikáció esetén lesz szerepe a callback-függvénynek. Ilyenkor a kérést küldő JavaScript-kód befejeződik, a válasz megérkezésekor azonban elindul a callback-függvény, és paraméterként megkapja a szerver üzenet tartalmát.

Az AJAX-technológia szépsége, hogy a válasz bármilyen típusú adat lehet. Egyszerű szövegtől kezdve a HTML- vagy XML-dokumentumokon keresztül akár a JSON-jelöléssel megadott JavaScript objektumokig bármi. A feldolgozás csak válaszüzenet fogadó callback-függvényen múlik.

A folyamatot a felhasználónak akár észre sem kell vennie, hiszen az általa használt weboldal az aszinkron kommunikáció miatt folyamatosan működik.

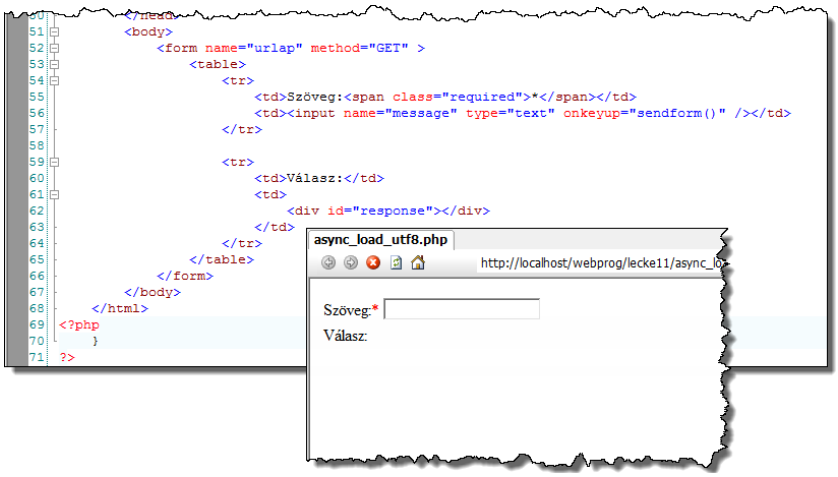
Az alábbi példa aszinkron AJAX-kommunikációt mutat be.

Forrás: async_load_utf8.php

A felhasználó szöveget gépelhet egy űrlap egyszerű szövegmezőjébe. A billentyű leütésekre elinduló JavaScript kód minden egyes leütéskor elküldi az input elem tartalmát a szervernek, amit az **`<b></b>`** elemek közé zárva küld vissza. A callback-függvény ilyenkor a weblap egészének megváltoztatása nélkül egyetlen táblázatcella tartalmába írja a kapott szöveget.

A felhasználó a "message" nevű INPUT-elembe gépel. Minden billentyű felengedésre elindul a sendform() JS függvény.

A válasz szöveg a "response" azonosítójú DIV-be kerül.



94. ábra Az űrlap és megjelenése a weblapon

Az AJAX kommunikációt biztosító objektumot a 20–27 sorok közötti kód állítja elő. Az elágazásra azért van szükség, mert az Internet Explorer korábbi verzióiban ActiveX elemként implementálták a **`XMLHttpRequest`** objektumot. Ez a kódrészlet azonnal lefut, amikor az oldal letöltődik, így a kommunikációt biztosító **`XMLHttpRequest`** típusú **`xmlhttp`** globális változó azonnal rendelkezésre áll.

95. ábra JavaScript

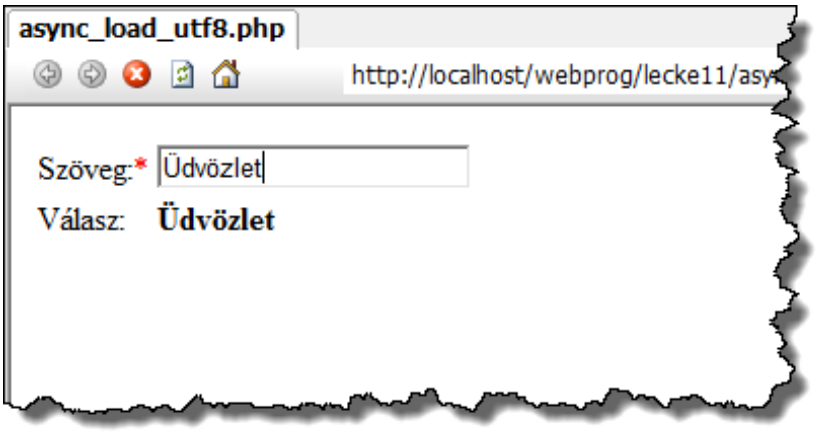
Az **XMLHttpRequest** típusú objektumok **onreadystatechange** eseménytulajdonságának értékeként azt a callback-függvényt kell megadni, amelyet válasz megérkezésekor hív majd meg a JavaScript. Ebben az esetben ez a **getresponse()** függvény lesz (29. sor).

> Figyeljük meg, hogy a függvény nevét zárójelek () nélkül adjuk meg, ugyanis nem meghívni akarjuk a függvényt, hanem hivatkozását akarjuk az **xmlhttp.onreadystatechange** tulajdonságba tenni.

A callback-függvényben megvizsgáljuk a választ.

Az **XMLHttpRequest** objektumok **readyState** tulajdonságának 4-es értéke jelzi, hogy a kérés a válasz megérkezésével fejeződött be. A **status** a szerver válaszának válaszkódja. A **200**-as érték jelzi, hogy a szerver hiba mentesen tudta kezelni a kérést. A kérésre érkezett válasz akkor feldolgozható, ha az **xmlhttp.readyState==4 && xmlhttp.status==200** logikai kifejezés igaz (32. sor). Szöveges tartalom esetén, a válasz az **XMLHttpRequest** típusú objektum **responseText** adattagjában található.

Ha a szerver válasza megérkezett, akkor nincs más dolgunk, mint az oldal megfelelő elemébe tenni a válasz szövegét. A 33., 34. programsorok a dokumentum **response** azonosítójú elemébe ágyazott tartalmat a válasz szövegére cseréli ki. A szöveg megjelenik a felhasználó előtt.



96. ábra A szerver válasza nem törli az odalt

Természetesen ahhoz, hogy a szerver válaszoljon, el is kell küldeni a kérést! A **sendform()** függvény a leütések alkalmával indul el. Kiolvassa az INPUT-elem tartalmát, létrehozza a szervernek elküldhető URL-t. A küldéshez **GET** metódust használ, ezért az URL tartalmazza a szerver oldali szkript nevét, és az űrlapon begépelt szöveget is.

> Az encodeURIComponent(msg) függvényhívás URI-ben továbbítható formátumúra konvertálja a szöveget.

Miután az URL rendelkezésre áll, megnyitja a kapcsolatot (40.sor), elküldi a kérést (41. sor)

A kapcsolat megnyitásában használt **xmlhttp.open** függvény első paramétere a metódus, a második az URL, a harmadik, logikai érték pedig azt jelzi, hogy aszinkron (**true**), vagy szinkron (**false**) módon kezelje-e a kérést az **XMLHttpRequest** objektum.

A példa állomány tartalmazza a szerveroldali szkriptet is, ami azonban csak akkor fut le, ha van GET metódussal feltöltött **msg** azonosítójú adat.

async_load_utf8.php

```
1  <?php
2      if (isset($_GET["msg"])){
3          $vnev=urldecode($_GET["msg"]);
4          if (isset($_GET["msg"])){
5              $msg=urldecode($_GET["msg"]);
6          }
7          header("Content-type:text/plain charset=utf-8");
8          echo "<b>$msg</b>";
9      } else {
10 ?>
11     <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
12     <html>
13         <head>
```

97. ábra Szerver oldali szkript

11.7 ÖSSZEFOGLALÁS, KÉRDÉSEK

11.7.1 Összefoglalás

Tananyagunk utolsó leckéjében betekintést nyertük a kliensoldali programozás feladataiba, JavaScript programozási nyelv használatába. Megtanultuk, hogy a JavaScript ECMA-szabványnak megfelelő nyelv, amelynek szintaktikája sok más nyelvhez, többek között a PHP-hez is nagyon hasonló.

A JavaScript kódokat a HTML **<SCRIPT>** jelölőibe ágyazhatjuk, vagy a jelölő **SRC** attribútumával csatolhatjuk.

A legelterjedtebb módszer, hogy a programot külső, általában **JS** kiterjesztésű fájlokban tároljuk, és a HEAD-ben elhelyezetett **<SCRIPT>** jelölőkkel csatoljuk. Ilyenkor a főprogram azonnal végrehajtásra kerül, a függvények csak hívásra futnak.

A függvényeket meghívhatjuk hivatkozások segítségével, de a dokumentum különböző elemeinek esemény attribútumaival is. A HTML 4 óta számos ilyen attribútumot használhatunk. Az értékükként megadott JavaScript-kódok az esemény bekövetkezésekor automatikusan lefutnak.

A JavaScript-ről megállapítottuk, hogy a weblapok dinamikus átalakítását nem annyira maga nyelv, hanem a Document Object Modell implementációja, a DOM API teszi lehetővé. A DOM a böngésző környezetében előforduló számtalan objektum hierarchikus rendszere. A DOM API segítségével azonosíthatjuk és a dinamikusan átalakíthatjuk a weblap elemeit. A különböző böngésző gyártók eltérő DOM-modelleket implementálták ezért a nagy böngészők korábbi verziói között jelentős inkompatibilitás mutatkozott. A böngészőgyártók egyre inkább elfogadják a W3C DOM ajánlásait, ezért az inkompatibilitásból adódó problémák némileg csökkentek, de a böngésző független JS-programok írása ma is komoly feladat.

Leckénkben néhány példán keresztül megvizsgáltuk a DOM használatát. Láttuk hogyan lehet nyomógombokkal és linkekkel JavaScript-kódokat indítani, megvizsgáltunk egy űrlap-validálási példát, láttuk, hogyan lehet a weblap újratöltése nélkül kommunikálni a szerverrel, és hogyan lehet megváltoztatni a weblap egy résztét.

11.7.2 Önellenőrző kérdések

1. Hogyan tud JavaScript programot illeszteni a weblapjába?
   - A HTML <SCRIPT></SCRIPT> elemével. A programot közvetlenül beágyazhatjuk a jelölőbe, de az SRC attribútummal csatolhatjuk is.
2. Mi a különbség a függvényekbe és a főprogramba írt kód futása között?
   - A főprogramba írt kód a szkript betöltődése után indul el, az alprogramok csak hívásra futnak.
3. Milyen vezérlési szerkezetet használhatunk a tömbök, objektumok bejárására?
   - A for(*ciklusváltozó* in *objekum*){} szerkezetet.
4. Mi az a JSON-jelölés?

- A JavaScript, objektumok literálként történő ábrázolására alkalmas módszere, a JavaScript Object Notation.

5. Hogyan tudja valamilyen műveletre kiválasztani a weblap egy bizonyos elemét?

- Erre több technika is kínálkozik, de a legegyszerűbb az ID alapján történő kiválasztás. Ehhez a document objektum getElementById() metódusát használhatjuk.

12. ÖSSZEFOGLALÁS

12.1 TARTALMI ÖSSZEFOGLALÁS

A 11 lecke áttanulmányozásával tananyagunk végére értünk. Most következő utolsó leckénkben az előző leckék rövid összefoglalóját a leckékben elsajátítható ismeretek summázatát találja.

12.1.1 Bevezetés

Első leckénk a tananyag fölépítését, a leckék szerkezetét, mutatta be. Az olvasó megismerhette a szövegben található formátumokat, tanulási tanácsokat kapott, és áttekinthette a tananyag tematikáját. Itt olvashatott a használt szoftverekről, és itt tölthette le a leckében használható forrásokat.

12.1.2 Web alkalmazások elemei és működésük

A második lecke a world wide web születését, és a szolgáltatás alapját képező kliens-szerver architektúrát, az URL-címzést, és a HTTP protokollt mutatta be.

12.1.3 Szerveroldali programozás PHP nyelven

Ebben a leckében ismerkedhetett meg a PHP nyelv kialakulásának történetével, a PHP-programok weblapokba ágyazásával és a programok futásának módjával. A 3. leckében az XAMPP csomag és a CodeLobster telepítésével alakítottuk ki a tanuláshoz szükséges szoftver környezetet.

12.1.4 Literálok, változók, tömbök

A meglehetősen terjedelmes 4. lecke a PHP alkalmazások alapvető nyelvi elemeit a literálokat, változókat, konstansokat, és tömböket ismertette.

12.1.5 A PHP vezérlési szerkezetei

Az 5. leckében ismerkedhetett meg a PHP vezérlési szerkezeteivel, a két és többágú elágazásokkal az elől és a hátul tesztelő, valamit a léptető ciklusokkal.

12.1.6 Kliens oldali adatok feldolgozása, állapotkezelés

A 6. leckére már elegendő ismerettel rendelkezett ahhoz, hogy megtanulja fogadni és feldolgozni a kliens oldalról érkező adatokat. Ebben a leckében ismertük meg a $_GET, és $_POST tömböket, és ellenőrzésük módját.

Ebbe a leckében tanultunk az állapotkezelés három lehetőségéről a rejtett INPUT elemek, a cookie-k és a munkamenetek használatáról.

12.1.7 Alprogramok és paraméterátadás, beépített függvények

A 7. leckében a procedurális programozás strukturális elemeiről az alprogramokról tanultunk. Megismertük a paraméterezés jelentőségét, a lokális és globális hatókörök jelentését. Megtanultuk a cím- és értékparaméterek közötti különbséget, a statikus változók használatát, és a változó számú paraméter kezelését.

12.1.8 MySQL-adatbázisok kezelése PHP-ben

A MySQL-ről szóló lecke bemutatta a procedurális technikával használható, függvényalapú MySQL API-t. Megtanultuk, hogyan lehet kapcsolatot fölépíteni a MySQL szerverekkel, mire kell figyelni az SQL-mondatok összeállításakor, hogyan lehet elküldeni az utasításokat a szervernek és hogyan lehet feldolgozni a választ.

12.1.9 Objektumorientált programozás a PHP-ben

A 9. lecke az osztályok készítésével foglalkozott. Megismertük az objektumorientált programozás előnyeit, megtanultuk, hogyan hozhatunk létre saját osztályokat.

12.1.10 Hibakeresés, hiba- és kivételkezelés

A hiba és a kivételkezelés a webalkalmazás készítésének egyik legfontosabb eleme. Az itt tanult technikákkal biztosíthatjuk, hogy alkalmazásunk ne kerülhessen definiálatlan állapotba.

12.1.11 A kliens oldali programozás alapjai

Utolsó leckénkben a JavaScript megismerésével bepillantottunk a kliens oldali programozás lehetőségeibe. A nyelv szintaxisán kívül olvashattunk a Document Object Modellről és példákat láttunk a JavaScript kódok interaktív indítására, az űrlapok validálására és az AJAX-technika használatára.

12.1.12 Tartalmi összefoglalás

Jelen, utolsó leckénkben a fenti összefoglalás alapján tekintettük át a tananyagban tanultakat.

12.2 FOGALMAK

12.2.1 Bevezetés

12.2.2 Web alkalmazások elemei és működésük

WWW, HTML, szerver-kliens, URL, HTTP, üzenet, statikus weblap, dinamikus weblap, szerver oldali program, kliens oldali program

12.2.3 Szerveroldali programozás PHP nyelven

PHP, PHP-jelölő, a PHP nyelvtana, program feldolgozása, kimenet

12.2.4 Literálok, változók, tömbök

szöveg literál, szám literál, logikai literál, kifejezés, operátor, operandus, típus, változó, konstans, tömb

12.2.5 A PHP vezérlési szerkezetei

elágazás,ciklus, elől tesztelő ciklus, hátul tesztelő ciklus, léptető ciklus

12.2.6 Kliens oldali adatok feldolgozása, állapotkezelés

felhasználói felület, adatküldés GET metódussal, állapotkezelés, rejtett input mezők, sütik, munkamenetek, fájlfeltöltés

12.2.7 Alprogramok és paraméterátadás, beépített függvények

függvények, beépített függvények, function kulcsszó, paraméter, formális paraméterlista, aktuális paraméterlista, értékparaméter, címparaméter, változók érvényességi köre, globális változó, lokális változó

12.2.8 MySQL-adatbázisok kezelése PHP-ben

MySQL API, mysql_connect, mysql_close, mysql_select_db, mysql_query, mysql_free_result, mysql_fetch_row, mysql_fecth_assoc, mysql_insert_id

12.2.9 Objektumorientált programozás a PHP-ben

OOP, egységbezártság, újrahasznosítás, öröklődés, polimorfizmus, oop nyelvek, osztály, metódus, adattag, konstruktor, destruktor, láthatóság, statikus metódus

12.2.10 Hibakeresés, hiba- és kivételkezelés

fordításidejű hiba, futás idejű hiba, die, error_reporting, trigger_error, callback függvény, set_error_handler, try…catch

12.2.11 A kliens oldali programozás alapjai

JavaScript, JavaScript szintaxis, DOM, interaktív indítás, validálás, AJAX

13. KIEGÉSZÍTÉSEK

13.1 IRODALOMJEGYZÉK

13.1.1 Hivatkozások

Könyv

EICHORN, Joshua: *Az Ajax alapjai*. Budapest, Panem Kft., 2008.

MONCUR, Michael: Tanuljuk meg a JavaScript (2.0) használatát 24 óra alatt. Budapest, Kiskapu, 2006.

SCHLOSSNAGLE, George: *PHP fejlesztés felsőfokon*:. Budapest, Kiskapu Kiadó, 2004.

ZANDSTRA, Matt: *Tanuljuk meg a PHP5 használatát 24 óra alatt*:. Budapest, Kiskapu Kiadó, 2005.

Elektronikus dokumentumok / források

NAGY GUSZTÁV: *Webes szabványok* Budapest, 2010. [elektronikus dokumentum] [2012.07.01.] <URL: http://kobakbt.hu/jegyzet/Webes_szabvanyok_0.1.pdf>

NAGY GUSZTÁV: *Web programozás alapismeretek.* Budapest, Ad Librum, 2011. [elektronikus dokumentum] [2012.07.05.] <URL: http://nagygusztav.hu/sites/default/files/csatol/web_programozas_-_szurke.pdf>

PHP DOCUMENTATION GROUP: *PHP Manual*, PHP Documentation Group 2012. [elektronikus dokumentum] [2012.06.30.] <URL: http://www.php.net/docs.php>

SZŰGYI Zalán: *A PHP programozási nyelv.* Budapest, ELTE, 2008. [elektronikus dokumentum] [2012.07.01.] <URL: http://nyelvek.inf.elte.hu/leirasok/PHP/>

TIMOTHY, Berners-Lee – FIELDING R.: *Hypertext Transfer Protocol.* W3C, 1996. [online] [2012.07.05.] <URL: http://tools.ietf.org/html/rfc1945 >

WORLD WIDE WEB CONSORTIUM: *Document Object Model (DOM).* W3C, 209. [online] [2012.06.10] <URL: http://www.w3.org/DOM/ >