Írta:
HOLLÓ CSABA

ÜZLETI WEBTECHNOLÓGIÁK

Egyetemi tananyag

TYPOTEX

2011

COPYRIGHT: © 2011–2016, Dr. Holló Csaba, Szegedi Tudományegyetem Természettudományi és Informatikai Kar Szoftverfejlesztés Tanszék

LEKTORÁLTA: Dr. Vámossy Zoltán, Óbudai Egyetem Neumann János Informatikai Kar Szoftvertechnológia Intézet

Creative Commons NonCommercial-NoDerivs 3.0 (CC BY-NC-ND 3.0)

A szerző nevének feltüntetése mellett nem kereskedelmi céllal szabadon másolható, terjeszthető, megjelentethető és előadható, de nem módosítható.

TÁMOGATÁS:

Készült a TÁMOP-4.1.2-08/1/A-2009-0008 számú, „Tananyagfejlesztés mérnök informatikus, programtervező informatikus és gazdaságinformatikus képzésekhez” című projekt keretében.

ISBN 978-963-279-507-2

KÉSZÜLT: a Typotex Kiadó gondozásában

FELELŐS VEZETŐ: Votisky Zsuzsa

AZ ELEKTRONIKUS KIADÁST ELŐKÉSZÍTETTE: Erő Zsuzsa

KULCSSZAVAK:

XHTML, CSS, webprogramozás, JavaScript, Java szervlet, JavaServer Pages, AJAX, akadálymentesség.

ÖSSZEFOGLALÁS:

A megjelenítő eszközök sokfélesége, illetve a fogyatékkal élő emberek támogatása miatt egyre fontosabb szerepet kap a weboldalak szabványos megalkotása, melynek legfontosabb eszközeit a jegyzet az alapoktól indulva mutatja be. Egy modern weboldal azonban a felhasználóval, illetve a felhasználók között történő kommunikáció különböző szintjeit kell biztosítsa, amihez szükséges a statikus kódnak a kiszolgáló, illetve az ügyfél gépén futó programokkal történő kiegészítése. A jegyzetben, kizárólag a Java nyelv ismeretét feltételezve, az alapoktól indulva mutatjuk be a Java alapú kiszolgáló oldali webprogramozás legalapvetőbb feladatait, illetve az ügyfél oldali webprogramozás JavaScript nyelv általi lehetőségeit. Különálló fejezetben foglalkozunk a manapság divatos AJAX kommunikáció megvalósítási lehetőségeivel, melynek tartalma akkor is érthető, ha az olvasó Java szervletek helyett más kiszolgáló oldali nyelvet (például PHP-t) ismer. Végül betekintést nyerünk két különböző megközelítést használó keretrendszerbe, és röviden szólunk a web design legfontosabb kérdéseiről. A jegyzet a tartalom sokszínűsége és terjedelmi korlátai miatt dokumentációnak nem tekinthető, viszont oktatási tananyagként készítése során kiemelt szempont volt a legfontosabb módszerek rendszerezett bemutatása és a tartalom érthetősége.

Tartalomjegyzék

 © Holló Csaba, SzTE

© Holló Csaba, SzTE

Előszó

Ez a jegyzet a TÁMOP-4.1.2-08/1/A-2009-0008 sz. „Tananyagfejlesztés és tartalomfejlesztés különös tekintettel a matematikai, természettudományi, műszaki és informatikai képzésekre” c. pályázat ([1]) keretében készült. A tananyag a pályázat előírásainak megfelelően első sorban a gazdaságinformatikus MSc szak számára lett kifejlesztve ([2]).

A jegyzet célja olyan, első sorban ügyfél oldali, webes technológiák bemutatása, melyek segítik üzleti jellegű alkalmazások elkészítését. Ha azt az ismeretanyagot tekintjük, amit egy ilyen kurzus keretében célszerű lenne tárgyalni, azt tapasztaljuk, hogy az egyes témákról külön-külön is több száz vagy ezer oldalas könyvek születtek. Ugyanakkor, a gazdaságinformatikus BSc Képzési és kimeneti követelményeiben ([3]) nem előírt a webprogramozás elsajátítása, ezért annak érdekében, hogy az érintett hallgatók számára a jegyzet és a hozzá kapcsolódó kurzus érthető legyen, továbbá a hallgatók annak elsajátítása után rendelkezzenek olyan alapokkal, melyekre érdemes fejlettebb technológiákat építeni, elengedhetetlen a weboldalkészítés alapoktól induló bemutatása. A jegyzetnek és a hozzá tartozó kurzusnak azonban meglehetősen szűk terjedelmi korlátai vannak, melyek a fenti megfontolások következtében a jegyzet tartalmára és szerkezetére vonatkozóan az alábbi korlátozásokat teszik szükségessé.

Mivel az érthetőség és a megfelelő alapozás érdekében nem kerülhető el az alapvető témakörök részletesebb bemutatása, ezért nem lehetséges minden témakör azonos mélységű tárgyalása. Ily módon viszonylag részletesen fogjuk tárgyalni azokat a témaköröket, melyek a használt keretrendszerektől függetlenül alacsonyabb szinten mindig megjelennek ((X)HTML, CSS, JavaScript, AJAX), kevésbé részletesen az AJAX és keretrendszerek megértéséhez szükséges kiszolgáló oldali alaptechnológiákat (szervlet, JSP), majd tömören bemutatunk két különböző megközelítést használó keretrendszert (JSF, Google Web Toolkit), és az ügyfél oldali programozásban nagyon fontos Web design ismeretek legfontosabb szempontjait. Csak nagyon indokolt esetben helyezünk el teljes méretű példákat, ugyanakkor melegen ajánlható az olvasónak, hogy a tárgyalt lehetőségeket saját készítésű példák által is kipróbálja, ugyanis ebben a témában semmi sem helyettesítheti a saját tapasztalatot. A kurzuson részt vevő hallgatók az elméleti órákhoz kapcsolódó gyakorlatokon számos konkrét példát tanulmányoznak.

A jegyzet a részletesebben tárgyalt témakörök esetében sem tekinthető teljes dokumentációnak, a cél inkább az, hogy a lehetőségeket szemléltessük. A kurzus anyagához kapcsolódó további ismereteket az olvasó az egyes fejezeteknél megadott irodalmakban talál.

Érdemes tisztázni azt is, hogy az egyes fejezetek megértéséhez milyen előismeretek szükségesek. A HTML-t és CSS-t bemutató fejezet alapvető informatikai (szövegszerkesztés, böngészők használata) ismereteken kívül semmilyen komolyabb előismeretet nem feltételez, a többi fejezet feltételezi az előző fejezetek és a Java nyelv illetve fejlesztőrendszerek ismeretét. Az AJAX-ról szóló fejezet akkor is érthető lesz, ha az olvasó Java szervletek helyett más kiszolgáló oldali nyelvet (például PHP-t) ismer.

Végül itt szeretnék köszönetet mondani Dr. Vámossy Zoltánnak a kézirat igen alapos és gondos lektorálásáért, hasznos tanácsaiért és konstruktív észrevételeiért.

1. Bevezetés

Az információk továbbítására már évszázadokkal ezelőtt különféle módszereket kidolgoztak. Kezdetben volt az írott sajtó és az üzenetek galambok lábain történő továbbítása, majd feltalálták a távírógépet, rádiót, televíziót, a számítógépet és a számítógépek közötti kommunikációt. Azonban ezek által még mindig kevesen tudtak üzenetet küldeni, vagy viszonylag kevés emberhez tudott eljutni az információ, és nem feltétlen azokhoz, akiket érdekelt.

Az első weboldal 1991. augusztus 6-án került közzétételre, melyet Sir Timothy John (azaz "Tim") Berners-Lee ([87]) készített a CERN munkatársaként. Ezt már, megfelelő hardver- és szoftvertámogatással, földrajzi korlátozások nélkül, bárki megnézhette, akit érdekelt. Azóta az információk minél szélesebb körben történő elérhetővé tételének célját messze meghaladta a Web, hiszen manapság közösségi csoportok létesítésének és kommunikációjának, tudástárak építésének, információk keresésének, ügyintézéseknek és üzleti tevékenységeknek a technikai hátterét is biztosítja. Mindez azonban úgy vált lehetségessé, hogy az eredeti ötletet számos további fejlesztés egészítette ki, melynek eredményeképpen az oldalakon már nem csak megjeleníteni lehet az információkat, hanem az oldalakhoz társított programok segítségével a weboldal képes kommunikálni a felhasználóval és számos tevékenységet automatikusan elvégezni.

További információkat a web fejlődéséről az olvasó a [17] oldalból kiindulva találhat.

2. XHTML és CSS

2.1. Alapfogalmak

2.1.1. Mi a HTML, XHTML és CSS?

Az információk megjelenítéséhez szükség volt egy nyelvre, mely segítségével le lehetett írni, hogy mit és hogyan akarunk megjeleníteni, továbbá egy programra, mely ezeket a leírásokat értelmezte és megjelenítette. A nyelvet elnevezték HTML-nek, a programot pedig böngészőnek.

A **H**yper **T**ext **M**arkup **L**anguage (HTML) ([30]) egy leírónyelv, mely definiálja a tartalom struktúráját (pl. címsor, felsorolás, táblázat stb.), melyet a különböző böngészők, beállításaiktól függően is különbözőképpen jeleníthetnek meg, továbbá segítségével a megjelenítést mi is szabályozhatjuk. A leírást sima szöveges parancsokkal adhatjuk meg, ugyanakkor a dokumentum különböző részei között, vagy különböző dokumentumok között kapcsolatot létesíthetünk. A HTML nyelvben lehetőség van internetes szolgáltatások (FTP, levelező program) és más alkalmazások indítására szolgáló hivatkozásokat is elhelyezni. A HTML dokumentum tehát lényegében strukturáló és formázó parancsokból és a megjelenítendő tartalomból álló szöveges dokumentum, mely bármilyen szövegszerkesztővel megírható. Ugyanakkor léteznek HTML szerkesztő programok is, melyek kódkiemeléssel, hibakereséssel, megjelenítéssel és automatikus kódgenerálással segítik a weblapok vizuális készítését (például TextPad, MS Front Page, Macromedia Dreamweaver stb.). Sok más célból írt program (pl. Word, Excel, Power Point stb.) is képes böngészők által értelmezhető kódot készíteni. Figyelni kell azonban arra, hogy az automatikusan generált kód sok felesleges elemet tartalmazhat, továbbá olyan speciális karakterkészletet és formázást definiálhat, mely csak bizonyos körülmények között teszi lehetővé a tartalom megfelelő megjelenítését.

Az igények növekedésével a HTML számos hiányossága is felszínre került. A megváltozott munkaképességű emberek támogatására, és a megjelenítő eszközök sokfélesége mellett az oldalak elérhetőségének biztosítására a leírókód használata nem hatékony. Ezért szükségessé vált a HTML átírása, melynek eredményeképpen lehetővé vált a tartalom és a megjelenítés különválasztása olya módon, hogy a megjelenítést úgynevezett rangsorolt stíluslapokban (**C**ascading **S**tyle **S**heets) ([31]) átláthatóbban és hatékonyabban szabályozhatjuk.

A HTML másik hiányossága, hogy azt szabályrendszerének következetlensége és többértelműsége miatt csak speciális feldolgozóval (parser) lehet elemezni. Annak érdekében, hogy a forráskód más XML feldolgozókkal együttműködjön, és belőle modern nyelvekben előforduló dokumentum típusú objektum példány készíthető legyen, megalkották az XHTML-t (e**X**tensible **HTML**, [40]), ami lényegében a HTML újraírása XML alapokon. Ily módon lehetővé válik olyan korszerű keretrendszerek használata is, melyek nem támogatják a HTML közvetlen módosítását. A HTML-hez hasonlóan az XHTML fájlban a megjelenítés is megadható, de lehetőség van stíluslapok használatára is.

A jelenlegi stabil verziók HTML 4.01, XHTML 1.0, melyek előnyeit magába foglaló és azokat kiegészítő HTML 5-ös végleges (ajánlott) változata 2014-re várható ([18]). A CSS jelenlegi ajánlott változata a CSS 2, melynek továbbfejlesztésére több projekt is létezik ([31]).

A továbbiakban az XHTML 1.0 és CSS 2 szintaxisának és használatának legfontosabb elemeivel fogunk megismerkedni. Az olvasó további információkat a fejezet alapvető forrásaként is szolgáló [5], [6], [7], [8], [10], [30], [31] irodalmakban találhat.

2.1.2. Objektumok definíciója

Azokat a strukturális egységeket, melyek egy weblap tartalmát alkotják, objektumoknak fogjuk nevezni. Ilyen objektumok lehetnek például egy címsor, egy bekezdés vagy egy kép. Az objektumok tartalmazhatják egymást, például egy táblázatban sorok vannak, a sorokban cellák vannak, a cellákban lehet valami tartalom (például szöveg vagy kép). A legtöbb objektum esetén az XHTML kódban meg kell adni, hogy az hol kezdődik és hol ér véget, ehhez úgynevezett *tag-okat* használunk. Az angol *tag* szó magyarul címkét, jelzőt jelent, de elfogadható megnevezésként a legtöbbször magyarul is a *tag* szót használjuk. Így tehát az objektumoknak van egy nyitó és záró tagjuk. Ha az objektumok tartalmazzák egymást, akkor fontos, hogy a beágyazott objektum kezdő és záró tagjával együtt a beágyazó objektum kezdő és záró tagja között legyen. Minden tag < és > jelek között helyezkedik el. Például, ha van 3 olyan objektumunk, melyek esetén az 1. objektum tartalmazza a 2. és 3. objektumokat, akkor ezek a következőképpen adhatók meg:

```
<1. objektum nyitó tagja>
  <2. objektum nyitó tagja>
  <2. objektum záró tagja>
  <3. objektum nyitó tagja>
  <3. objektum záró tagja>
<1. objektum záró tagja>
```

A kezdő tag mindig tartalmazza az objektum típusát és bizonyos esetekben az objektum egyes tulajdonságainak megnevezését és értékét. A záró tag tartalmazza egy / jel után az objektum típusát. Lássunk egy példát:

```
<p>Az XHTML kód írásakor figyelnünk kell arra, hogy az objektum <em>kötelező tulajdonságait</em> adjuk meg, és <em> ne felejtsük el a tagot lezárni</em>.</p>
```

Itt a `p` egy bekezdést, az `em` pedig kiemelt fontosságú szöveget határoz meg.

XHTML-ben az objektum típusát, jellemzőit és azok előre definiált értékeit kisbetűvel kell írni. Minden jellemzőnek egyértelműen meg kell adni az értékét és azt idézőjelek között kell feltűntetni. Abban az esetben, ha a jellemző feltűntetése önmagában is elegendő információt hordoz, azaz logikailag a név feltűntetése érték nélkül is elegendő lenne, a jellemző értékének annak nevét kell adni. A következő példában az `option` objektummal egy választási lehetőséget adunk meg, a `selected` jellemzővel pedig azt írjuk elő, hogy a csoportba tartozó lehetőségek közül az adott objektum legyen alapértelmezetten kiválasztva. Nyilvánvalóan a `selected` jellemző jelenléte önmagában elegendő lenne, de az XHTML szabályai szerint minden jellemzőnek értéket kell adni, ezért az objektum szintaxisa a következőképpen fog kinézni: `<option selected="selected">kijelölt választási lehetőség</option>`.

Az objektumok megjelenítése előre definiált sémák szerint történik. Két ilyen fontos séma a *blokkszintű*, illetve a *sorszintű* (folyamatos, inline) megjelenítés. A blokkszintű elem megjelenését dobozszerűen kell elképzelni, tartalma külön sorokban jelenik meg és tartalmazhat más dobozszintű vagy sorszintű elemeket is. A sorszintű elem a szövegen belül folyamatosan jelenik meg és csak sorszintű elemeket tartalmazhat. Megjegyzendő, hogy az objektumok alapértelmezett megjelenítési sémája indokolt esetben megváltoztatható.

Az XHTML-ben előre definiált jelentéssel bíró objektumok mellett mi is definiálhatunk saját strukturális egységeket `div` és `span` objektumok segítségével. Ezek az objektumok a böngésző számára nem hordoznak tartalomra vonatkozó információt, ezért ezek használata olyan esetekben ajánlott, amikor az előre definiált objektumok nem alkalmasak a céljainkra.

A kettő közötti különbség a tartalom típusában, illetve a megjelenítési sémában rejlik, ugyanis alapértelmezésként a `div` blokkszintű, míg a `span` pedig sorszintű elemnek számít.

A fejezet elején azt mondtuk, hogy a legtöbb objektumnak nyitó és záró tagja is van. Léteznek azonban olyan objektumok is, melyek nem tartalmazhatnak más objektumokat, ezért esetükben nincs értelme két tagot használni. Ilyen esetben elegendő lenne a nyitó tag feltűntetése, de XHTML-ben minden tagot le kell zárni, ezért a nyitó tagban ezt is megtesszük a következő szintaxissal: `<nyitó tag />`. Például, ha a tartalom adott helyén egy képet szeretnénk csatolni, akkor azt a következőképpen tehetjük meg: `<img src="kep.jpg" alt="a kép szöveges leírása" />`.

Ahhoz, hogy az objektumokat formázás vagy más műveletek céljából azonosítani tudjuk, szükség lehet egyedi vagy csoportos azonosító hozzárendelésére. Egyedi azonosítót az `id` tulajdonsággal tudunk megadni és szabályosan más objektum nem rendelkezhet ugyanazzal az azonosítóval. Egy objektumhoz csoportos azonosítót a `class` tulajdonsággal tudunk hozzárendelni, és ezt akkor szoktuk használni, amikor több objektumra közös feladatot vagy megjelenítést szeretnénk definiálni.

2.1.3. A CSS nyelvtan alapvető szabályai

Amint már említettük, a CSS segítségével megjelenítési tulajdonságokkal láthatjuk el a leíró-kódban levő objektumokat. Megadhatjuk például a szövegek színét, betűtípusát, az objektumok hátterét, szegélyét vagy a weboldalon elfoglalt helyét.

Amikor megjelenítési tulajdonságokat adunk meg, mindenekelőtt meg kell határoznunk, hogy azok pontosan mely objektumra, vagy objektumok mely csoportjára vonatkoznak. Erre szolgálnak CSS-ben az úgynevezett *kijelölők*. Minden kijelölőhöz tartozik egy *meghatározás*, mely a megjelenítés leírását tartalmazza.

A meghatározást mindig `{` és `}` jelek közé tesszük. A megjelenítést úgy tudjuk leírni, hogy a megfelelő tulajdonságoknak a kívánt értékeket adjuk. A meghatározásban a tulajdonságot `:`-al választjuk el az értékétől, a tulajdonság:érték párokat pedig `;`-vel választjuk el egymástól. Az értéket csak akkor tesszük idézőjelek közé, ha az több szóból áll. Bizonyos esetekben egy tulajdonságnak több értéket is megadhatunk, akkor ezeket `,`-vel választjuk el egymástól. Például a kijelölt objektumok (jelen esetben a bekezdések) betűcsaládját és betűszínét a következőképpen adhatjuk meg:

```
p {font-family: "Times New Roman", Times, serif; color: blue;}.
```

Bizonyos esetekben több tulajdonság értékét rövidebben is megadhatjuk oly módon, hogy csak a tulajdonságok értékeit soroljuk fel, egymástól szóközzel elválasztva. Például, ha bizonyos objektumoknak (jelen esetben a képeknek) 5px vastagságú, vonalkázott, kék színű szegélyt akarunk adni, akkor külön értéket adhatunk rendre a `border-width`, `border-style`, illetve `border-color` tulajdonságoknak, vagy mindezek megadhatók egyszerre is a következőképpen:

```
img {border: 5px dashed blue;}.
```

De hogyan jelölhetjük ki a formázni kívánt objektumokat? Erre többféle kijelölő létezik: elemkijelölő, azonosítókijelölő, osztálykijelölő és összetett kijelölő. A megjelenítési tulajdonságok elemkijelölő esetén a dokumentum összes adott típusú objektumára, osztálykijelölő esetén az összes azonos `class` attribútum értékkel rendelkező objektumára, míg azonosítókijelölő esetén csak az adott `id` azonosítóval rendelkező objektumra vonatkoznak. Összetett kijelölőt akkor használunk, amikor a kívánt objektum vagy objektumok csoportja több kijelölő együttes használatával jelölhető ki.

Elemkijelölőt úgy adhatunk meg, hogy egyszerűen leírjuk a típus nevét. Például, ha azt szeretnénk, hogy az összes bekezdés zöld színű háttérrel és kék színű betűkkel jelenjen meg, akkor ezt a következőképpen írhatjuk: `p{background-color:green; color:blue;}`. Osztálykijelölő esetén egy `.` után írjuk a közös `class` értéket. Például ha az összes *kiemeles* `class` értékkel rendelkező objektum tartalmát szeretnénk kék betűkkel írni, akkor ezt a következőképpen tehetjük meg: `.kiemeles{color:blue;}`. Egy objektum több osztályhoz is tartozhat egyszerre, ebben az esetben a `class` attribútum értékeit szóközzel választjuk el egymástól. Azonosítókijelölő esetén egy # jel után írjuk az egyedi azonosító értékét. Például, ha csak egy *fontos* azonosítóval rendelkező objektum tartalmát szeretnénk kék betűkkel írni, akkor azt kell megadnunk, hogy `#fontos{color:blue;}`.

A `class` illetve `id` attribútumok értéke olyan azonosító lehet, mely az angol ábécé betűit, számokat és _ jelt tartalmaz és betűvel kezdődik.

Összetett kijelölők esetén a fentieket együtt használjuk. Például, ha azt akarjuk megadni, hogy egy *lablec* azonosítójú szakasz kifejezés osztályú bekezdéseinek karakterei legyenek kék színűek, akkor azt írjuk, hogy:

```
#lablec p.kifejezes {color:blue;}.
```

Abban az esetben, ha több kijelölőhöz ugyanolyan meghatározást akarunk rendelni, akkor a kijelölőket `,`-vel elválasztva felsorolhatjuk a közös meghatározás előtt. Például, ha azt akarjuk, hogy az összes bekezdés és az összes kiemelt szöveg kék színű legyen, akkor írhatjuk azt, hogy `p, em {color:blue;}`. Vigyáznunk kell azonban arra, hogy ha a `,`-t elhagyjuk, akkor a meghatározás csak a bekezdésekben levő kiemelt szövegrészekre lesz érvényes.

2.1.4. A weblap részei

Az (X)HTML dokumentum leírása előtt meg kell adnunk a dokumentumtípus meghatározást ([5]).

2.1.4.1. A dokumentumtípus meghatározás

A dokumentumtípus meghatározás (DTD, azaz **D**ocument **T**ype **D**efinition) keretében deklaráljuk, hogy az oldalt a HTML mely változatának szabályrendszerének megfelelően írjuk meg. Azt mondjuk, hogy a HTML kód *érvényes*, ha valóban a kiválasztott DTD szabályrendszerének megfelelően van megírva, és elvileg ez kellene biztosítsa, hogy a tartalom minden böngészőben ugyanúgy jelenjen meg. Sajnos még nem minden böngésző támogatja teljes egészében a szabványokat, ezért az egyforma megjelenés nem teljesen működik. A HTML kód érvényességének ellenőrzése alapvetően a http://validator.w3.org/ címen lehetséges. Az érvényesség ellenőrzésére bizonyos HTML szerkesztőprogramok, illetve böngésző-kiegészítők is képesek. Például egy Mozilla Firefoxban telepíthető ilyen kiegészítő letölthető a https://addons.mozilla.org/hu/firefox/addon/249 címről HTML Validator néven.

Az XHTML dokumentumokhoz háromféle dokumentumtípus meghatározás tartozhat: szigorú, átmeneti és keretkészletes.

Szigorú dokumentumtípus meghatározás esetén csak szerkezeti felépítésre vonatkozó tagokat és jellemzőket használhatunk. Az XHTML 1.0 változatát használó szigorú dokumentumtípus meghatározás a következőképpen adható meg:

```
<!DOCTYPE html PUBLIC
     "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Átmeneti dokumentumtípus meghatározás esetén az XHTML kódban használhatók egyes nem szabványos, megjelenítést beállító elemek és jellemzők is. Például használható, de el-

avult elemnek számít a szöveg igazítását szabályozó `align` jellemző, melyet CSS-ben is megadhatunk a `text-align` tulajdonság segítségével. Előfordulnak olyan helyzetek is, amikor az átmeneti szabványban elfogadott jellemzők szigorú szabvány szerint elfogadható CSS-el történő kiváltása sokkal bonyolultabb, ugyanakkor érdemes mérlegelni, hogy a nem szigorú szabvány szerinti megoldások jövőbeli működése bizonytalan. Az átmeneti dokumentumtípus meghatározás megadásának szintaxisa:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

A keretkészletes dokumentumtípus meghatározást kereteket tartalmazó weblapokhoz használják, és a következőképpen lehet megadni:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

A kereteket később fogjuk részletesen tárgyalni.

XHTML dokumentumok esetén a W3C javasolja, hogy adjuk meg az XML meghatározást és névteret a következőképpen. Az XML meghatározás

```
<?xml version=”1.0” encoding=”utf-8”?>
```

míg az XML névtér megadása:

```
<html xmlns =  "http://www.w3.org/1999/xhtml">.
```

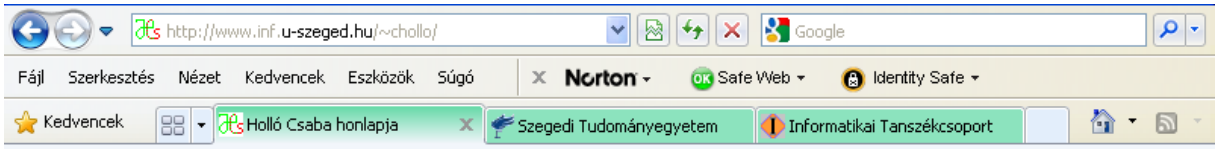
A dokumentumtípus meghatározás után következhet a dokumentum, melyet, `<html>` és `</html>` tagok között fogunk leírni, és mely két fő részből áll: fejrész és törzsrész.

2.1.4.2. A fejrész

A fejrész a `<head>` és `</head>` tagok között adható meg és a weboldallal kapcsolatos meta adatok megadására szolgál. A meta adat adatra vonatkozó adat.

Jelen esetben a weboldalon megjelenítendő adatokra vonatkozó meta adatok lehetnek például a használandó karakterkészlet, a weboldal címe, szerzője, a tartalom kulcsszavai, illetve a használandó CSS stíluslapok. A meta adatokat a keresőprogramok is használják az indexelés és katalogizálás során. A fejrészben csatolhatunk az oldalhoz programkódokat is.

A weblap ablakának címét a `<title>` és `</title>` tagok közé lehet beírni, és az ablak vagy a lap tetején fog megjelenni, továbbá ezt a címet menti alapértelmezésként a böngésző is, amikor a felhasználó felveszi a lapot a kedvencei közé. Lehetőség van a cím előtt, illetve a címsorban egy speciális képet, úgynevezett *favikont* (favorite icon) megjeleníteni ([32]). Ilyen favikonokat és címeket láthatunk az 1. ábrán.



1. ábra: Címek és favikonok

A kép 16x16-os (ajánlott), vagy 32x32-es 8 vagy 24 bites színezésű png, gif vagy ico kép kell legyen, melynek W3C szabványos beillesztése:

```
<head profile="http://www.w3.org/2005/10/profile">
<link rel="icon" type="image/png"
      href="http:// eleresiUtvonal/kep.png ">
```

A profile fájlt azért adjuk meg, mert abban van leírva, hogy mit jelent a `rel` jellemző `icon` értéke. Ahhoz, hogy minden böngésző jól megjelenítse, hasznos lehet csatolni `rel =`

"shortcut icon" értékkel is. Ha ico fájlunk van, akkor type = "image/x-icon"-t kell megadnunk.

Bizonyos meta adatok megadására meta objektumokat használunk, melyek szintaxisa eltér a többi objektumétól. A meta tag name, illetve content jellemzőivel egy tulajdonság nevét, illetve annak értékét adhatjuk meg. A meta tagnak más jellemzői is lehetnek. Például az oldal szerzőjét, kulcsszavait és azoknak nyelvét a következőképpen írhatjuk le:

```
<meta name = ”author” content = ”Nagy Tihamér” />
<meta name = ”keywords” lang = ”hu”
      content = ”zeneszerzés, zenélés” />
```

A name jellemző helyett használható a http-equiv ([33], [34]) olyan információk megadására, melyeket a szerver fejléc információként továbbíthat. A fejléc információkkal később részletesen fogunk foglalkozni, de addig is lássunk két példát a http-equiv gyakori használatára. A tartalom típusát és karakterkészletét a következőképpen írhatjuk le:

```
<meta http-equiv = "Content-Type"
      content = "text/html;charset=ISO-8859-2" />
```

Ha pedig egy lejárati időpontot akarunk megadni a dokumentumnak, mely után azt a böngésző akkor is újra letölti, ha a cache-ben megtalálható, akkor azt így írjuk:

```
<meta http-equiv = "expires"
      content = "Tue, 25 Aug 2011 14:25:27 GMT" />
```

Ha azt akarjuk, hogy a dokumentum mindig újból letöltődjön, akkor múltbeli időpontot kell beállítanunk.

A fejrészben csatolhatunk a weblaphoz egy külső CSS stílusfájlt (melynek szokásos kiterjesztése css), vagy beágyazhatunk csak az adott weboldalra vonatkozó stílusinformációkat is. A külső stíluslap csatolása a fejrészben a címobjektum után kell legyen, és a következőképpen lehetséges:

```
<link rel=”stylesheet” type=”text/css”
    href=”[elérési út/]stilualap.css”
 />
```

ahol a [és] jeleket nem kell kiírni, azok csak a közöttük levő tartalom opcionális jellegét jelölik.

Beágyazott stílusokat akkor szoktunk használni, ha csak azon az oldalon a külső stíluslaphoz képest kisebb változtatásokat szeretnénk. Ezeket a külső stíluslap csatolások után <style type = ”text/css”> és </style> közé írjuk. Ha elképzelhetőnek tartjuk, hogy az oldalt olyan böngészővel is fogják nézni, mely nem ismeri a CSS-t, akkor a megjelenítés szabályozását a <style>-on belül <!-- és --> HTML megjegyzések közé tehetjük azzal a céllal, hogy a böngésző ne vegye figyelembe az általa értelmezhetetlen parancsokat. A CSS-t ismerő böngészők azonban figyelembe veszik a HTML megjegyzések között levő stílusmegadásokat is, ezért amit CSS-ben akarunk megjegyzésbe tenni, azt /* és */ közé kell írnunk.

Egy külső stíluslapba, vagy a beágyazott style tagba egy másik stíluslapot is be tudunk olvastatni az @import url([elérési út/]stilus.css); paranccsal, melynek a stílusmegadások előtt kell szerepelnie.

A külső stílusfájlok használatának több előnye is van. A tartalomfájlok és a stílusfájlok sok esetben különböző személyek által párhuzamosan készíthetők, ezért gyorsulhat a fejlesztés. Ha egy stílusfájlt több weboldalhoz csatolunk, akkor abban egyetlen módosítással az összes csatolt weboldal kinézetét megváltoztathatjuk. Mivel ugyanazokat a megjelenítést szabályozó parancsokat nem kell minden XHTML fájlban ismételten leírni, így azok mérete kisebb,

 © Holló Csaba, SzTE

tartalmuk átláthatóbb lesz és gyorsabban letöltődik. Bizonyos források szerint a keresőprogramok is relevánsabbnak tekintik a forrásfájlban hamarabb előforduló tartalmat, ezért is jobb, ha abban minél kevesebb más jellegű tartalom (formázás, programkód) található. Másfelől, egyetlen tartalmat leíró fájlhoz több megjelenítést szabályozó fájl is társítható, ami akkor lehet előnyös, ha ugyanazt a tartalmat különböző helyzetekben másképp szeretnénk megjeleníteni. Például, a `<link>`, illetve az `@import` esetében is használható egy `media` jellemző, melynek értékeként megadhatjuk, hogy az adott stíluslapot milyen eszközök esetén használja. A `media` jellemzőnek sok lehetséges értéke van, melyek közül csak néhány fontosabbakat sorolunk fel: `all` (minden eszköz), `aural` (hangeszközök), `handheld` (kézi eszközök), `print` (nyomtató), `projection` (kivetítő), `screen` (képernyő). Sajnos a korszerű böngészők sem tudnak minden értéket kezelni, de az `all`, `print`, `projection` értékek többnyire használhatók.

2.1.4.3. A törzsrész

A törzsrész a megjelenítendő tartalmat és annak strukturális leírását tartalmazza. Ilyen strukturális egységek lehetnek például a címsorok, bekezdések, táblázatok, felsorolások, képek vagy űrlapelemek.

A törzsrész tartalma `<body>` és `</body>` között helyezkedik el. Megjelenítési tulajdonságok megadhatók a teljes törzsrészre, illetve az egyes strukturális elemekre vonatkozóan is.

Az XHTML fájlok standard kiterjesztései htm vagy html.

Az eddigi ismereteink birtokában készítsük el első weboldalunkat. Az XHTML fájl, melyet a későbbi egyértelmű hivatkozások érdekében ux1.html néven mentünk el, a következőket fogja tartalmazni:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
 <head profile="http://www.w3.org/2005/10/profile">
  <title> XHTML, CSS 1. példa </title>
  <link type="image/x-icon" rel="icon"
        href="../Kepek/HCs_fav.ico"/>
  <link type="image/x-icon" rel="shortcut icon"
        href="../Kepek/HCs_fav.ico"/>
  <link rel="stylesheet" type="text/css" href="ux1.css" />
  <meta http-equiv="Content-Type"
        content="text/html;charset=ISO-8859-2" />
 </head>
 <body>
  <div id = "linda">
    <p class = "kepek" id = "elso">
      Ez az első kép <em>Lindáról</em>:
      <img src = "../Kepek/kutya1mk.jpg" alt= "Linda képe" />
    </p>
  </div>
 </body>
</html>
```

Nemsokára el fogjuk készíteni a kódban hozzá csatolt, de még nem létező ux1.css stílusfájlt is. Addig azonban, ha az oldalt különböző böngészőkben megnyitjuk, akkor látni fogjuk, hogy az a böngészőkbe beépített alapértelmezett formázásokkal hogyan jelenik meg.

Ugyan a `html`, `head` és `title` objektumokat a böngésző akkor is létrehozza, ha nem szerepelnek a kódban, a szabványos leírásban ezek feltűntetése is kötelező.

2.1.5. Stílusok hatásköre

A böngészők rendelkeznek alapértelmezett stílusokkal, melyek meghatározzák, hogy az egyes objektumok hogyan jelenjenek meg akkor, ha nem szabályozzuk azok megjelenítését. Például ilyen alapértelmezések a fehér színű háttér, fekete színű betűk, aláhúzott kék színű linkek stb. Ugyanakkor a felhasználónak is van lehetősége a böngészőben beállítani néhány tulajdonságot (például a betűtípust, vagy a háttérkép nyomtatását), illetve többnyire saját stíluslapot is megadhat.

Az előző fejezetben kiderült, hogy a weboldal készítőjeként stílusokat megadhatunk az XHTML fájlhoz csatolt külön CSS fájlban, illetve a fejrészbe beágyazva. További stílusinformációk adhatók meg az egyes objektumok nyitó tagjában is a `style` jellemző értékeként. Ezeket nevezzük *szövegközi* stílusoknak, és csak arra az objektumra lesznek érvényesek, melynek nyitó tagjában szerepelnek. Például:

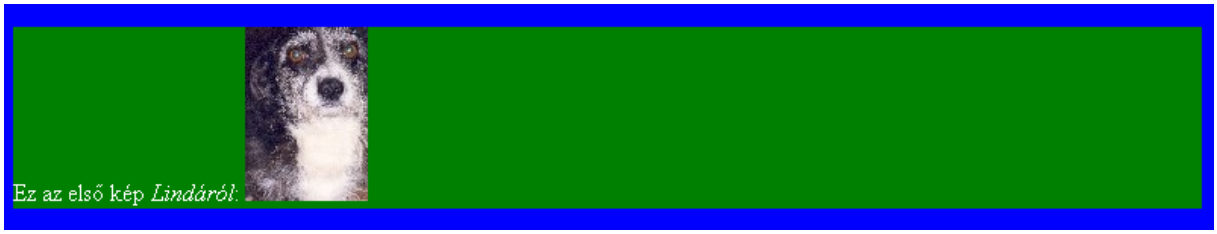
```
<p style="color: blue;"> Csak ez a bekezdés kék. </p>
```

A böngésző az oldal tartalmát fentről lefele olvassa be. Ha közben külső fájlcsatolást talál, akkor annak a tartalmát azon a ponton beolvassa, majd folytatja a weboldal beolvasását fentről lefele. Ily módon egyszer a külső stíluslapok, majd a beágyazott stílusinformációk, végül az objektumok nyitó tagjában elhelyezett stílus meghatározások olvasódnak be. Egymásnak ellentmondó stílusinformációk esetén az utoljára beolvasott lesz érvényes. Így lehet a fejrészben felülírni a közös külső stílusfájlban foglaltakat, illetve az objektumok nyitó tagjában csak arra az objektumra vonatkozóan módosítani az egész weboldalra megadott megjelenítési jellemzőket.

Tudjuk, hogy az objektumok többségét egymásba ágyazhatjuk. A beágyazott objektumok örökölhetik a beágyazó objektum formázásait, több szintű beágyazás esetén is. Például, ha egy bekezdésben van egy kiemelt szövegrész, akkor arra nem csak a saját stílusmegadása lehet érvényes, hanem a bekezdésre megadott formázás is. Azonban az öröklés nem mindig történik meg. Egyrészt, vannak olyan tulajdonságok, melyek nem öröklődnek (például a margó vagy a szegély). Másrészt, egy objektum megjelenítési tulajdonságokat kaphat a beágyazó objektumokon kívül az egyedi, osztály-, illetve összetett kijelölők segítségével megadott stílus meghatározások által is, melyek között ellentmondás állhat fenn. Készítsük el például az ux1.html fájlhoz csatolt `ux1.css` stílusfájlt az alábbi tartalommal:

```
body {background-color:blue; color:green;}
#linda {background-color:green;  color:black;}
p {color:red;}
.kepek {color:blue;}
#elso {color:white;}
```

A bekezdés betűszínének öröklés alapján feketének, elemkijelölő alapján pirosnak, osztályazonosító alapján kéknek, azonosítókijelölő alapján pedig fehérnek kellene lennie. Hogyan fog megjelenni? A szabály az, hogy mindig az adott objektumra legjellemzőbb stílusmegadás lesz érvényes. Ebben a példában, amint azt a 2. ábrán is láthatjuk, a bekezdés szövege fehér színű lesz, mivel az örökölt, elemkijelölővel, illetve osztályazonosítóval megadott tulajdonságok sok más objektumra is érvényesek lehetnek, de az azonosító kijelölővel kifejezetten erre a bekezdésre vonatkozó megjelenítést adtunk meg.



2. ábra: ux1.html

Mi történne a fenti példában, ha nem lenne egyedi azonosító alapján történő formázás megadva (azaz töröljük az `#elso {color:white;}` formázást)? Az öröklésben, az elemkijelölővel vagy az osztálykijelölővel megadott formázás számít jellemzőbbnek? Ilyenkor az osztálykijelölő lesz a legspecifikusabb, ugyanis nyilván azért rendeltük az objektumot az adott csoporthoz, hogy az azokra megadott megjelenítés legyen rá is érvényes, míg az oldal korrekt strukturális felépítését nem tehetjük függővé a megjelenítéstől. De mi történik akkor, ha sem egyedi, sem osztálykijelölő nincs megadva, azaz töröljük a `.kepek {color:blue;}` formázást is? Akkor az elemkijelölő lesz érvényes, ugyanis az adott objektumra nézve jellemzőbb, hogy az objektum milyen típusú, mint az, hogy milyen másik objektumban van benne. Összetett kijelölők esetén látszólag bonyolultabb helyzetek is előállhatnak, de ezek a szempontok mindig egyértelműen meghatározzák, hogy egymásnak ellentmondó esetekben melyik stílus megadás lesz érvényes. Az `ux1.html`-ben például a bekezdés benne van a `linda` egyedi azonosítóval rendelkező `div`-ben, a bekezdésben mégsem érvényesül a `#linda` azonosító kijelölőre megadott betűszín, mert a benne levő bekezdésre az öröklésnél specifikusabb meghatározások is adva vannak. Amikor nincs ellentmondás, akkor viszont az egyes stílus megadások mind érvényessé válnak. Például a bekezdés a `linda` azonosítóval rendelkező `div` zöld háttérszínét örökli, mert az nincs specifikusabb kijelölővel felüldefiniálva.

A megjelenítési tulajdonságok megadásakor érdemes figyelembe venni azt is, hogy a százalékban megadott értékek a korábbi beállítások alapján értékelődnek ki. Végül megemlítendő, hogy sok esetben a CSS hierarchiáját úgy építjük fel, hogy a külső CSS önmagában is épít az elfedésre.

2.1.6. Címsorok

A címsorok a fejezetek és alfejezetek címének megadására szolgálnak és nem keverendők a dokumentum `title` tagjában megadott címével. A címsorok tagolják a weblapon megjelenő szöveget, tartalmukat pedig sok esetben a keresőprogramok is használják az indexelés és katalogizálás során.

A címsoroknak hat szintjét különböztetjük meg, melyeket fontosságuk csökkenő sorrendjében a `h1`, `h2`, …, `h6` tagokkal jelölünk. Előírás szerint egy oldalon legfeljebb egy darab legfelső szintű (azaz `h1`-es) címsor lehet. Mint minden beépített HTML objektumnak, a címsoroknak is van a böngészőkbe beépített alapértelmezett megjelenítési stílusuk: blokkszintű elemeknek számítanak, általában félkövér, és a fontosság csökkenésével egyre kisebb méretű karakterekkel jelennek meg, de CSS segítségével természetesen ezt megváltoztathatjuk.

Fontos megjegyezni, hogy a strukturális egységek azért lettek kialakítva, hogy adott helyzetekben a böngésző, vagy az ahhoz társított (például felolvasó) program azonosítani tudja őket és megfelelően tudja őket kezelni. Az oldalak elérhetőségének biztosítása érdekében nem ajánlott (CSS-sel is megvalósítható) formázás érdekében tartalmilag nem megfelelő strukturális egységnek deklarálni valamilyen tartalmat (például csak vastagítás érdekében címsorként

megadni nem cím jellegű szöveget), vagy fordítva, strukturális egység megadása helyett a tartalmat csak ahhoz hasonlóan formázni.

2.2. Szöveges tartalmak kialakítása és megjelenítése

2.2.1. Karakterek és mértékegységek

A karakterek meghatározásánál meg kell adnunk a használt karakterkódolást, a betűtípust (mely a karakterek megjelenítését határozza meg) és a megjelenítendő karakter nevét vagy kódját.

A 2.1.4.2. fejezetben láttuk, hogy a karakterkódolást ([35], [36], [37]) a fejrészben adjuk meg a következőképpen:

```
<meta http-equiv = "Content-Type"
      content = "text/html;charset=ISO-8859-2" />
```

A magyar nyelvben leggyakrabban használt karakterkódolások a Latin-2 (közép-európai, ISO/IEC 8859-2), illetve az UTF-8.

A betűtípus stílustulajdonság ([38]), mely a `font-family` kijelölő segítségével adható meg. A `font-family` néhány lehetséges értéke: `Verdana`, `Arial`, `Georgia`, `Courier`, `Courier New`, `Times New Roman`. Nem bízhatunk abban, hogy a felhasználó gépén az általunk használni kívánt betűcsalád telepítve van, és abban sem, hogy a felhasználó csak a mi kedvünkért azt majd telepíteni fogja, ezért ajánlott többféle betűcsalád felsorolása.

A betűk mérete a `font-size` kijelölővel adható meg. A méretet megadhatjuk pontban, pixelben, százalékértékben, `em`-egységben, `em`-egységben, vagy az alábbi értékek valamelyikével: `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`, `smaller`, `larger`. Képernyőn való megjelenítéshez nem ajánlott a méret pontban történő megadása, mert nem egyértelmű, hogy egy pontnak hány pixelt fog megfeleltetni. Hasonlóan, az előre definiált méretnevek értelmezése is böngészőfüggő lehet. Ugyanakkor a pixelben való méret megadás sem ajánlott, mert elvileg nem teszi lehetővé gyengén látók számára a betűk méretének növelését. A böngészők ebben az esetben is biztosíthatják a betűméret növelésének lehetőségét, de ez nem várható el tőlük. Az `em`-egység az adott betűtípusban az `M` betű magasságát, az `ex`-egység pedig az `x` betű magasságát jelöli. A böngészők az x méretét többnyire az `em` felének veszik. Ezen mértékegységek segítségével más objektumok is méretezhetők a felhasználó által beállított betűmérethez viszonyítva. A százalékérték számítása a szöveget beágyazó szülőelem betűmérete alapján történik. A fentiek miatt a betűméret beállításánál ajánlott az `em`-egység vagy százalékérték használata. Továbbá figyelni kell arra, hogy a betűméret növelésekor is a teljes szöveg látható maradjon.

A betűk stílusa a `font-style` kijelölővel adható meg és lehetséges értékei: `italic`, `oblique`, `normal`, melyek dőlt, ferde, vagy normál megjelenítést írnak elő.

Kiskapitális betűket a `font-variant` (betűváltozat) kijelölő `small-caps` értékével kaphatunk. Ez azt jelenti, hogy a kis betűk is a nagy betűkkel azonos alakúak lesznek, csak valamivel kisebb méretben.

A betűvastagságot a `font-weight`, kijelölővel lehet megadni. Több lehetséges értéke van, melyekkel azonban nem minden betűtípus rendelkezik, a leggyakrabban használt értékek a `normal` és a `bold`.

A fenti stílusok értékei megadhatók egymás után felsorolva a `font` kijelölő értékeként is az alábbi sorrendben: betűváltozat, betűstílus, betűvastagság, betűméret, betűcsalád. Ajánlott legalább a betűméret és betűcsalád megadása.

A tagoló karaktereknek (sorvég, tabulátor, szóköz) alapértelmezés szerint nincs hatásuk a szöveg elrendezésére, ugyanis a böngésző egyetlen szóközzel helyettesíti őket, és további parancsok hiányában a szöveget a böngésző akkor tördeli új sorba, amikor az ablakban a sor végére ér. Például, nagyobb betűméret esetén, egy sor kevesebb karaktert fog tartalmazni.

Speciális karakterek megadására úgynevezett *karakterentitásokat* ([39]) használunk. Ezek karaktereknek megfelelő HTML kódok vagy hexadecimális értékek, melyekkel ékezetes karaktereket is írhatunk, illetve tagoló karakterek hatását is kiválthatjuk. Például a szóköz karaktert az ` ` vagy ` ` karakterentitással írathatjuk ki, akár többet is egymás után, ebben az esetben a böngésző nem fogja összevonni őket. Egy másik példaként említhetjük a < és > jeleket, melyek XHTML-ben a tagokat határolják, ezért ha ilyen jeleket ki akarunk íratni, akkor helyettük az `<` vagy `&#60;`, illetve `>` vagy `&#62;` karakterentitásokat kell használnunk.

2.2.2. Szövegek tagolása

Eddig megismerkedtünk a bekezdésekkel, illetve a szövegek tagolására is alkalmas `div` és `span` objektumokkal. Számos további beépített objektum létezik, melyekkel speciális jellegű szövegtartalmakat határozhatunk meg, ezek közül fogunk megnézni néhány fontosabbat. Megjegyzendő, hogy a felsoroltak mindegyikének tartalma a kezdő és záró tagok között helyezkedik el.

Rövidítést az `abbr` taggal tudunk megadni, ugyanakkor illik valahogyan azt is hozzárendelni, hogy mit rövidít. Ehhez többnyire a `title` tulajdonságot szoktuk használni, de figyelni kell arra, hogy az nyomtatásban nem fog megjelenni, tehát ott más megoldást kell keresni. A `cite` taggal más információforrást lehet megjelölni, míg `blockquote`-vel idézetet tudunk megadni. Már említésre került, hogy egy szövegrész fontosságának kihangsúlyozására az `em` tagot használjuk, melynek semmi köze nincs az `em` mértékegységhez. Lehetséges ugyanakkor bizonyos szövegrészek még erősebb kihangsúlyozása a `strong` taggal. Felső és alsó indexet `sup`, illetve `sub` objektumokként adhatunk meg. Számítógépes kódokat `code` objektumként, billentyűzetről beviendő utasításokat `kbd` objektumként lehet megadni. Végül meg kell említenünk a `pre` tagot is, melynek használata bizonyos helyzetekben nagyon kényelmes, mert megtartja az eredeti dokumentum szerinti szóközöket és sortöréseket. Használható például ASCII karakterekkel előállított képekhez vagy egyszerű táblázatok kialakítására, ugyanakkor vitatható, hogy ez mennyire tekinthető strukturális egységnek. Látni fogjuk a továbbiakban, hogy a tagoló karakterek figyelembevételének szabályozása CSS-ben is lehetséges.

2.2.3. Szövegekre vonatkozó stílustulajdonságok

Szöveg balra, jobbra, illetve középre igazítását a `text-align` tulajdonság `left`, `right` és `center` értékeivel lehet előírni.

Ha azt szeretnénk, hogy egy szövegrész aláhúzott, föléhúzott, áthúzott vagy villogó legyen, akkor a `text-decoration` stílustulajdonságot kell használnunk az `underline`, `overline`, `line-through`, illetve `blink` értékekkel, melyek közül több is megadható egyszerre.

Lehetőség van a forráskódban elhelyezett szöveg betűit a megjelenítésben nagybetűsre, kisbetűsre, illetve szavanként nagy kezdőbetűsre alakítani a `text-transform` tulajdonság `uppercase`, `lowercase`, `capitalize` értékeivel. Az utóbbi esetben megjegyzendő, hogy a szavak további betűi változatlanul jelennek meg és böngészőfüggő lehet, hogy mit tekint egy szónak.

A karakterek közötti távolságot a `letter-spacing` tulajdonsággal lehet állítani. Negatív érték megadása esetén a karakterek közelebb kerülnek egymáshoz.

A sor magassága adja meg, hogy az egyes sorok milyen távolságra legyenek egymástól. Alapértelmezett értéke a karakterek méretétől függ, értéke a `line-height` kijelölővel állítható be és szabályosan csak pozitív érték lehet.

A bekezdések első sorának behúzásának mértékét a `text-indent` tulajdonsággal adhatjuk meg, mely negatív érték esetén behúzás helyett kilógást fog eredményezni.

A tagoló karakterek figyelembevételének szabályozására a `white-space` jellemző használható, melynek `normal` értéke esetén az alapértelmezést kapjuk, azaz a megjelenítés során az egymás utáni tagoló karakterek egyetlen szóközzé lesznek összevonva és a sortörés kifejezetten ilyen parancsot kiváltó (például blokkszintű) objektum hiányában akkor lesz, amikor a tartalom már látható módon nem fér el az adott sorban. A tulajdonság `nowrap` értéke esetén a böngésző az ablak szélén sem fogja automatikusan megtörni a sorokat, viszont szükség esetén megjelenít egy görgetősávot. Sortörést bárhol kiválthatunk a `<br />` taggal is. Másfelől, ha a `white-space` tulajdonságnak a `pre` értéket adjuk, akkor a tagoló karakterek mind figyelembe lesznek véve. Köztes megoldásként a `pre-wrap` értékkel azt lehet előírni, hogy vegye figyelembe a tagoló karaktereket, de automatikus sortörést is alkalmazzon, míg a `pre-line` érték esetén megőrizze a sortörést, egyébként az alapértelmezés szerint járjon el. Sajnos ezeket a lehetőségeket (jellemzően az utolsó kettőt) a böngészők jelenleg még nem mind ismerik, ezért alkalmazásuk előtt érdemes kipróbálni őket a megcélzott böngészőkben.

2.3. Objektumok felépítése és megjelenítése

2.3.1. Színek

Színeket többféle objektumnak megadhatunk, általában a `color` jellemző értékeként. Színt kaphatnak például a karakterek, vonalak, sőt még a háttér is, azonban a szín meghatározása minden esetben ugyanúgy történik. A színek additív színkeveréssel jönnek létre a piros, zöld és kék színek intenzitásának megadásával, ezt nevezzük RGB kódolásnak ([43]).

A legelterjedtebb módszer szerint a három alapszín intenzitását kétjegyű hexadecimális számokkal adjuk meg ([44]), ily módon egy szín kódja #$r_1r_2g_1g_2b_1b_2$, ahol r_1r_2 a piros, g_1g_2 a zöld és b_1b_2 a kék szín intenzitását adja meg. Például a `#FF0000` a piros, a `#00FF00` a zöld, a `#0000FF` a kék, a `#000000` a fekete és a `#FFFFFF` a fehér színeket határozzák meg. Ily módon 256 * 256 * 256 = 16.777.216 színárnyalatot kaphatunk. Abban az esetben, ha r_1= r_2=r, g_1= g_2=g, és b_1= b_2 =b, akkor rövidítve is írható: #rgb formában. Például a `#A0C` jelentése `#AA00CC`.

A `00`, `33`, `66`, `99`, `CC`, `FF` karakter párok kombinációiból előállított összesen 216 színt *webtűrő színeknek* nevezzük, mely ezek elvileg minden eszközön és operációs rendszeren majdnem azonos árnyalatban kellene megjelenjenek.

Az alapszínek intenzitását, ugyanilyen sorrendben, százalékban, vagy 0 és 255 közötti egész számként is megadhatjuk az `rgb(100%, 100%, 100%)`, illetve `rgb(255, 255, 255)` formában.

Végül 16 színt angol nyelvű nevükkel is megadhatunk: `white` (fehér), `black` (fekete), `red` (piros), `green` (zöld), `blue` (kék), `gray` (szürke), `silver` (ezüstszürke), `yellow` (sárga), `purple` (lila), `aqua` (akvamarinkék), `navy` (mélykék), `teal` (pávakék), `fuchsia` (mályvaszín), `maroon` (gesztenyebarna), `lime` (világoszöld), `olive` (olajzöld).

2.3.2. Háttér

Az objektumoknak a `background-color` tulajdonság értékeként háttérszínt, a `background-image: url(`*képnév*`)` stílusmegadással pedig háttérképet adhatunk meg, ahol a *képnév* a háttérként használni kívánt kép elérési útja. Háttérkép esetén a `background-position` tulajdonsággal megadhatjuk annak elhelyezkedését, melynek értéke a szokásos mértékegységekkel megadottakon kívül lehet `top` (felül), `bottom` (alul), `center` (középen), `left` (balra), `right` (jobbra).

A `background-attachment` tulajdonság alapértelmezett `scroll` értékével azt adhatjuk meg, hogy a tartalom gördítésénél a háttérkép is gördüljön, míg a `fixed` érték esetén az a tartalom gördítése esetén sem mozdul el.

A `background-repeat` tulajdonsággal tudjuk szabályozni, hogy a háttérkép mozaikszerűen ismétlődjön-e: `no-repeat` érték esetén nem ismétlődik, `repeat-x` esetén vízszintesen, `repeat-y` esetén függőlegesen, míg az alapértelmezett `repeat` esetén mindkét irányban ismétlődik.

2.3.3. A dobozmodell

A dobozmodell ([45], [46]) egy vizuális formázási modell, mely a W3C előírásait tartalmazza a dokumentumban elhelyezett elemek megjelenésére. Ennek értelmében a törzsrészben elhelyezett minden objektum egy téglalap alakú területen jelenik meg, melynek kialakítását dobozmodellnek nevezzük. A téglalapban legbelül helyezkedik a tartalom, melyet körbevesz rendre a belső margó (kitöltés, `padding`), a szegély (`border`) és a (külső) margó (`margin`). A tartalmat körülvevő minden elem tulajdonságait megadhatjuk mind a négy irányban egyszerre, vagy külön-külön, ezek értékeit az óramutató járásával egyező irányban felsorolva: felül (`top`), jobbról (`right`), alul (`bottom`) és balról (`left`). A méretet mindegyikre a szokásos mértékegységekkel adhatjuk meg. Százalékos megadás esetén azt a dokumentum teljes szélességéhez, illetve magasságához viszonyítva számolja. A belső margó háttere a tartalom háttere lesz, a külső margó pedig mindig átlátszó. A belső margót a tartalom szegélytől való eltávolítására, a külső margót pedig a szomszédos elemektől való távolság szabályozására szoktuk használni.

A szegélyeknek a méretén kívül megadhatjuk a színét és stílusát, melynek megjelenítése böngészőfüggő lehet. Szegély stílusok: `dotted` (pontozott), `dashed` (szaggatott), `solid` (folytonos vonal), `double` (dupla vonal), `groove` (barázda, bemélyített), `ridge` (perem, kidomborodó), `inset` (süllyesztett), `outset` (kiemelkedő). Például a `valami` azonosítóval rendelkező objektumot a következőképpen formázhatjuk:

```
#valami{ background-color: #fff; margin: 3em 10% 1em 30%;
padding: 1em; border-top: 30px groove #f00;  border-right:
30px inset #00f; border-bottom: 30px ridge #f00; border-left:
30px outset #00f; }
```

Ebben az esetben a belső margó minden irányban `1em` nagyságú lesz, a külső margó felül `3em`, jobbról 10%, alul `1em`, balról 30% lesz, továbbá a szegély egyes irányainak színét, stílusát és méretét külön-külön megadtuk.

A blokkszintű objektumok tartalmára vonatkozóan megadhatjuk annak szélességét és magasságát a `width` és `height` tulajdonságokkal. Százalékos érték esetén azt a beágyazó objektumhoz viszonyítva számolja. A sorközi objektumok méretét a tartalmuk határozza meg.

Szabvány szerint a teljes objektum mérete a fentiek összegeként kapható meg. Megjegyzendő ugyanakkor, hogy az Internet Explorer 8-nál korábbi verziói a tartalom méretébe a belső margót és a szegélyt is beleszámolták.

2.3.4. Helyzetmegadás

Helyzetmegadással az objektumok weboldalon elfoglalt pozícióját határozzuk meg. A helyzetmegadás módja lehet abszolút, viszonyított, rögzített vagy statikus. Tegyük fel, hogy van egy `id="nev"`-el azonosított objektumunk, melyet szeretnénk elhelyezni.

Abszolút helyzetmegadás esetén az elhelyezendő objektumot közvetlenül beágyazó objektum bal felső sarkát tekintjük a koordinátarendszer kiinduló pontjának, azaz (0,0) pozíciónak, és azt adjuk meg, hogy az elhelyezendő objektum ahhoz képest lefele, illetve jobbra milyen távolságra helyezkedjen el. Negatív érték esetén a beágyazó objektum bal felső sarkához képest balra, illetve felfele pozícionálunk. Például, ha a `nev` objektumot az őt beágyazó objektum bal felső sarkához képest 20 pixellel lejjebb és 30 pixellel jobbra akarjuk elhelyezni, akkor ezt a következőképpen írjuk:

```
#nev {position: absolute; top:20px; left: 30px;}
```

A rögzített helyzetmegadás a teljes weboldal bal felső sarkához viszonyítva helyezi el az objektumot a kért pozícióba úgy, hogy az oldal gördítésénél sem fog elmozdulni. Az Internet Explorer 7-nél korábbi verzióiban ez nem működik. A rögzített helyzetmegadás szintaxisa:

```
#nev {position: fixed; top:20px; left: 30px;}
```

Az abszolút és rögzített helyzetmegadások az elhelyezett objektumot kiveszik a normál szövegfolyamból, így annak helyét a normál szövegfolyamban maradó további tartalom foglalja el.

A viszonyított helyzetmegadás az elhelyezett elemet a normál szövegfolyami helyéhez képest elmozdítja a kért mértékben és nem veszi ki azt a normál szövegfolyamból (azaz a normál szövegfolyamban utána következő objektumok változatlan helyen fognak megjelenni). Szintaxisa:

```
#nev {position: relative; top:20px; left: 30px;}
```

A statikus helyzetmegadás a normál szövegfolyamban levő helyén hagyja az elemet és akkor használjuk, ha az objektum más helyzetmegadást örökölt és vissza akarjuk helyezni a normál szövegfolyambeli helyére. Ebben az esetben a pozíciók megadásának nincs hatása, így a szintaxis:

```
#nev { position: static; }
```

További helyzetmegadási lehetőség az úsztatás, mellyel azt lehet előírni, hogy a többi tartalom körülfolyhassa az úsztatott objektumot. Az úsztatás a `float` jellemző `left`, `right` illetve `none` értékével adható meg, melyek esetén az objektum balra, jobbra vagy egyáltalán nem lesz úsztatva, tehát azt jobbról, balról vagy nem folyja körbe a többi tartalom.

Blokkszintű objektumok vízszintes középre igazítása a következőképpen lehetséges:

```
.center {margin-left:auto; margin-right:auto; }
```

Hogyha egy objektumot függőlegesen is középre szeretnénk igazítani, akkor ezt így tehetjük meg:

```
#fkozepre {
  width: szélesség; height: magasság;
  position: absolute;
  top: 50%; left: 50%;
  margin-top: -magasság/2;
  margin-left: -szélesség/2;
}
```

ahol *magasság* és *szélesség* helyett az objektum magasságát és szélességét tűntetjük fel. Azért szükséges negatív margókat megadni, mert az abszolút helyzetmegadás az objektum bal felső sarkát pozícionálja, viszont mi az objektum közepét szeretnénk középre helyezni.

2.3.5. Átfedés, átlátszóság és cím

Az objektumoknak van egy `z-index` tulajdonsága, melynek értékül egy egész számot adhatunk, és amely meghatározza, hogy egymást egészben vagy részben eltakaró objektumok közül az átfedő területen melyik legyen látható. Ha egy objektumnak nem adunk `z-index` értéket, akkor azt örökli a beágyazó objektumtól, ha pedig nincs örökölhető érték, akkor alapértelmezett értéke 0. Azonos `z-index` értékkel rendelkező, egymást átfedő objektumok közül az lesz látható, amelyik nyitó tagja a forráskódban később fordul elő, tehát például a beágyazott objektum eltakarja a beágyazó átfedő részét. A `z-index` működése azonban `position`-el abszolút, viszonyított vagy rögzített módon nem pozícionált objektumok esetében bizonytalan.

Az oldalakon elhelyezett objektumokat (jellemzően képeket, háttereket) az általunk kívánt mértékben átlátszóvá is tudjuk tenni. A megoldás böngészőfüggő. Szabványosan az `opacity` stílustulajdonságnak adhatunk meg egy 0 és 1 közötti értéket azzal a jelentéssel, hogy minél nagyobb értéket adunk meg, az objektum annál kevésbé lest átlátszó (tehát 0 esetén teljesen átlátszó, 1 esetén egyáltalán nem). Nem szabványosan (Internet Explorerben) az átlátszóságot a következőképpen is megadhatjuk: `filter: alpha(opacity=10);`, ahol az `opacity` értéke 0 és 100 közötti egész szám és az érték növelése itt is csökkenti az átlátszóságot (tehát nem átlátszó itt a 100-as értéknél lesz).

Végül, az objektumoknak a `title` XHTML tulajdonsággal egy címet is adhatunk, melyet a böngészők többsége megjelenít, ha a felhasználó az objektum fölé viszi az egeret, vagy fölötte jobb egérgombot nyom.

2.3.6. Vonalak

Vízszintes vonalat a `<hr />` taggal tudunk megjeleníteni, mely alapértelmezésként a tartalom szélességének megfelelő normál vastagságú vonalat húz. Azonban a `width` és `height` CSS jellemzőkkel meg lehet adni a vonal szélességét és magasságát (vastagságát), `color`-al a színét, `border`-el a szegélyét, és mint minden objektumot pozícionálni is lehet.

Nincs olyan XHTML tag, mely függőleges vonalat jelenítene meg, viszont meglévő ismereteinkkel ez is megoldható. Az egyik megoldás, hogy létrehozunk egy megfelelő függőleges vonalat tartalmazó képet és azt beillesztjük. Létezik azonban ennél kisebb memóriaigényű és könnyebben módosítható megoldás is. Létrehozunk egy `div` objektumot, melynek magasságát a függőleges vonal kívánt magasságára, szélességét 0-ra állítjuk, és szegélyként megadjuk a vonal kívánt szélességét, stílusát és színét. Például minden olyan `div` objektum, mely az alábbi csoporthoz tartozik egy 200px magasságú, 2 px szélességű, piros színű, pontozott függőleges vonalként jelenik majd meg:

```
.fv {height: 200px;  width: 0px; border-left:2px dotted #f00;}
```

2.3.7. Nyomtatás formázása

Amint azt már említettük, CSS stíluslapok csatolásánál a `media` jellemző `print` értékével meg lehet adni, hogy nyomtatáskor mely stíluslap legyen használva. Abban az esetben, ha általában minden eszközre vonatkozó stílushoz képest csak néhány módosítást szeretnénk, akkor egyszerűbb a nyomtatási stíluslapot a mindenre érvényes (`media="all"`) stíluslap után beilleszteni és abban csak a különbségeket leírni. Például:

```
<link rel="stylesheet" type="text/css" href="x9a.css" media =
"all"/>
<link rel="stylesheet" type="text/css" href="x9p.css" media =
"print"/>
```

De milyen fontosabb változtatásokat érdemes eszközölni egy nyomtatási stíluslapban ([15])? Egyrészt érdemes elrejteni az olyan tartalmakat, melyeket a felhasználó a kinyomtatott oldalon úgyse tudna hasznosítani. Ilyenek lehetnek például a keresőmezők, menük, linkek és gombok is. Az elrejtésnek két módja van. Az egyik esetben a `visibility` tulajdonságot használjuk, melynek lehetséges értékei `inherit` (örökölt), `visible` (látható) és `hidden` (rejtett). A másik lehetőség a `display` tulajdonság használata a `none` értékkel. Például: `#kereses {display:none;}`. A két megoldás között az a különbség, hogy a `display:none;` kiveszi az objektumot a normál szövegfolyamból, míg a `visibility:hidden;` esetén csak üresen marad az objektum helye. Az objektumok `display`-el történő elrejtése a felhasználó számára papírspórolást is eredményez. Papírtakarékosság szempontjából ugyanakkor hasznos lehet az is, ha a nyomtatási oldalon, a weben megszokott keskeny sorokkal szemben, szélesebb sorokat használunk. Általános vélemény, hogy míg a képernyőn a talp nélküli betűket (pl. `Verdana`, `Arial`, `sans-serif`), nyomtatásban a talpas betűket (pl. `Georgia`, `Times`, `serif`) könnyebb olvasni. A kinyomtatott lapon nincsenek pixelek, ezért nyomtatáskor más mértékegységeket használunk, mint például `cm`, `mm`, `in` (inch, hüvelyk, kb. 25.4 mm), `pt` (pont = `in`/72), `pc` (pica = 12 `pt`). A nyomtatási stílusban ajánlott megadni a margókat és a betűméreteket is. Ugyanakkor érdemes figyelembe venni, hogy a felhasználó számára a böngészőben számos nyomtatási beállítás rendelkezésére áll (pl. nyomtasson-e háttérképet, oldalszámot, URL címeket stb.). A felhasználó többnyire nem nyomtatja ki a hátteret, ezért egyrészt ne bízzunk abban, hogy a nyomtatásban is lesz háttérkép, másrészt gondoskodjuk arról, hogy a nyomtatásban a fehér lapon is láthatók legyenek a betűk. Végül gondoljunk azokra a felhasználókra is, akik nem színes nyomtatóval nyomtatnak, így olyan színeket adjunk meg, amelyek akkor is látszani fognak, ha azokat fekete-fehér nyomtatóval nyomtatják ki.

2.4. Hivatkozások elhelyezése

2.4.1. Webhelyek azonosítása

A webes erőforrásokat azonosíthatjuk hely szerint vagy egyedi azonosítóval. Az elektronikus tartalom azonosítására az interneten jelenleg szinte kizárólag az URL (**U**niversal **R**esource **L**ocator, egységes erőforráscím) azonosítókat használják, amely minden egyes dokumentum esetében annak lelőhelyét (fizikai helyét) adja meg. Például:
`http://www.kiszolgalo.hu/mappa/fajl.html`.

Az elektronikus dokumentumok hosszú távú azonosítására megalkották az URN (**U**niversal **R**esource **N**ame) egyedi azonosítót is, mely egy központi helyen az URL-el együtt tárolva és frissítve lehetővé teszi a dokumentumok helytől független azonosítását. Hogyha a böngészők és a hivatkozások az URN-el azonosítanának, akkor a dokumentumok áthelyezése esetén csak a weblap készítőjének kellene a központi adatbázisban módosítani a dokumentum lelőhelyét, így a felhasználók könyvjelzői nem válnának működésképtelenné, és a hivatkozásokat sem kellene egyenként frissíteni. Ez azonban jelenleg nincs szélesebb körben alkalmazva, így ezt a problémát a könyvjelzők és hivatkozások átírása mellett leggyakrabban az oldalak átirányításával oldják meg. Fontos tehát a jó tervezés, hogy később minél kevesebb áthelyezést kelljen eszközölni, és a tartalom is minél átláthatóbb legyen.

A weboldalak azonosításánál még meg kell említeni, hogy a web szervereken beállítható, hogy ha egy könyvtárban található `index` vagy `default` nevű `htm` vagy `html` kiterjesztésű nevű fájl, akkor a könyvtárhivatkozással automatikusan azt nyissa meg.

2.4.2. A hivatkozás szintaxisa

Egy hivatkozásnak két része van, melyeket gyűjtőnéven *horgony*oknak szoktak nevezni: a *forrás* (amire kattintani lehet) és a *cél* (ahova mutat). A forrás megadása XHTML-ben az `<a>` (anchor) taggal lehetséges: `<a>` XHTML objektum `</a>`.

A kattintható XHTML objektum többnyire szöveg szokott lenni, de más objektum (kép, gomb stb.) is lehet. A cél URI-jét, vagyis azt, hogy hova mutasson a link, az `<a>` tag `href` (hypertext reference) tulajdonságának értékeként adjuk meg. A forrásra kattintva a böngésző a `href` értékeként megadott célt jeleníti meg. Lehetőség van arra is, hogy célként egy weboldalon belül azonosított pontot adjunk meg. Ezt azonosíthatjuk `id`-vel, vagy megadhatjuk a következőképpen: `<a name="celNeve"> Első cél </a>`. A hivatkozás általános alakja:

`<a href=` *"protokoll*`://`*gép*`.`*domain*`[:`*port*`]/`*út*`/` *fájlnév*`[#`*celNeve*`]"> Ide kattints ! </a>`,

melyben a `[]`-ek által jelölt opcionális részt *töredék*nek nevezzük. Például, egy azonos oldalon belüli célt így adhatunk meg:

`<a href="#ide.ugrik"> Ugrás ! </a>`,

míg egy azonos könyvtárban levő másik oldalon levő célt pedig így:

`<a href= "masik.htm#ide.ugrik"> Ugrás! </a>`.

A hivatkozásoknál alapértelmezett és leggyakrabban használt protokoll a `http`, amikor a cél egy helyi vagy távoli szerveren levő HTML dokumentum lehet, viszont a `file` protokollal helyi állományra, az `ftp` protokollal FTP szerveren levő állományra is hivatkozhatunk. A `mailto` protokollal a levelezőrendszer elindítását, az e-mail cím és tárgysor automatikus kitöltését írhatjuk elő a következőképpen:

```
<a href="mailto:chollo@inf.u-szeged.hu?subject=targy"> Levél küldése! </a>.
```

Az állományok helye megadható abszolút vagy relatív elérési úttal. A fejrészben elhelyezhető `base` tag `href` értékeként megadható a weboldalon elhelyezett relatív URL-ek alapértelmezett kiinduló könyvtára, mely viszont abszolút elérési út kell legyen. Például:

```
<head> …
  <base href="http://www.inf.u-szeged.hu/~chollo/gombok/" />
</head>.
```

2.4.3. Hivatkozások formázása

A hivatkozásoknak 4 fő állapota lehet: a felhasználó még nem kereste fel, a felhasználó felkereste, az egér fölötte áll, illetve a hivatkozás használatban van (a hivatkozott oldal betöltése folyamatban van). A hivatkozás állapota azonban nem szerepel az XHTML kódban, hanem a felhasználó tevékenységétől függ, ezért a hivatkozások formázására CSS-ben úgynevezett *álosztály-kijelölőket* ([47]) használunk, melyeket a fenti állapotokra rendre a következőképpen kell megadni: `a:link`, `a:visited`, `a:hover`, `a:active`. Az álosztály-kijelölőket más kijelölőkhöz hasonlóan használhatjuk további kijelölőkkel együtt is, és segítségükkel a hivatkozásokat a négy állapotra vonatkozóan külön-külön formázhatjuk. Például, az `a:hover` segítségével megadható, hogy amikor a felhasználó az egeret a hivatkozás fölé viszi, akkor az más háttérképpel, betűszínnel és betűmérettel rendelkezzen.

Alapértelmezésként a forrást a böngészők általában aláhúzva vagy bekeretezve jelenítik meg. Ezt módosítani az objektum `text-decoration`, illetve `border` tulajdonságának értékadásával lehet (például letiltás a `none` értékkel lehetséges).

A böngésző azokat az oldalakat tekinti felkeresetteknek, amelyek szerepelnek az adott böngésző böngészési előzményei között. Az `a:link` teszteléséhez szükség lehet arra, hogy az oldal újból nem felkeresettnek minősüljön, ehhez azt törölni kell azt az adott böngészőben a böngészési előzmények közül.

2.5. Képek és multimédiás tartalmak beillesztése

2.5.1. Képek beszúrása

Amint ez már említésre került, képeket az `<img>` tag segítségével lehet beilleszteni a tartalomba, melynek kötelezően elvárt jellemzői az `src` és az `alt`. Az `src` jellemző értékeként a beszúrandó képfájl abszolút vagy relatív elérési útját kell megadni, míg az `alt` értékeként egy olyan szöveget, melyet a böngésző fog kiírni, ha a képet nem tudja megjeleníteni. Az `alt` jellemző értékeként első sorban a kép rendeltetését kell megadni, tehát azt, hogy miért van ott (például valamilyen gombként működik), és nem azt, hogy mi látható rajta, ugyanis az első rendű cél az, hogy ha a képet a böngésző (vagy a felolvasóprogram) nem is jeleníti meg, a felhasználó az oldalt akkor is tudja használni. A kép tartalmának részletes leírását elhelyezhetjük egy külön fájlban, melynek elérési útját a `longdesc` tulajdonság értékeként tudjuk megadni. Ajánlott minden esetben megadni a kép méreteit a `width`, illetve `height` tulajdonságok segítségével. Más objektumokhoz hasonlóan képeknek is megadhatunk szegélyt és margót is.

2.5.2. Ügyfél oldali képtérképek létrehozása

Képtérkép segítségével alakzatokat definiálhatunk a képen, melyeket hivatkozásokká alakíthatunk. A képtérképet önálló objektumként definiáljuk, melyet majd hozzá fogunk rendelni ahhoz a képhez, mellyel használni szeretnénk.

Egy képtérkép a `<map>` és `</map>` tagok között helyezkedik el. A `map` tag `name` tulajdonságának értékeként a képtérképnek egy nevet adhatunk, melyet a képtérkép adott képhez való hozzárendelése érdekében az `img` objektum `usemap` tulajdonságának értékeként `#` jel után kell megadni A képtérkép `<area>` és `</area>` tagok között leírt tartalmú, az `area` tag `shape` tulajdonságaként megadott típusú alakzatokat tartalmaz. Az alakzat típusa lehet kör, sokszög vagy téglalap, ennek megfelelően a `shape` tulajdonság értéke lehet `circ`, `poly` vagy `rect`. Az `area` tag `coords` tulajdonságának értékeivel tudjuk meghatározni az alakzat helyét és méretét. Téglalap esetén négy értéket kell megadnunk: először a téglalap bal felső alsó sarkának x, y koordinátáit, majd a jobb alsó sarkának x, y koordinátáit. Sokszög esetén `coords` értékeként felsoroljuk a sokszög csúcspontjainak x és y koordinátáit. Kör esetén a kör középpontjának x és y koordinátáit és a kör sugarának méretét kell megadnunk. Az `area` tag `href` tulajdonságának értékeként a hivatkozás célját tudjuk megadni. Lássunk egy példát képtérképre, melyben egy kört és egy téglalapot definiálunk:

```
<map id="t1" name="t1">
   <area alt = "Kis gyerek" shape="rect"
         coords="0,61,118,392" href="x2.html" />
   <area alt = "Kör" shape="circle"
         coords="33,33,16" href="x5.html" /></map>
```

A képtérképet a következőképpen tudjuk egy képhez kapcsolni: `<img id = "szekelykep" src = "szekely.jpg" alt= "Székely gyerekek" height = "400" width = "238" usemap ="#t1" />`.

2.5.3. Multimédiás tartalmak beillesztése

Multimédiás fájlokat XHTML-be beszúrni a W3C előírásai szerint `object` tagként kell ([48]), azonban ez nem minden böngészőben működik, ezért még próbálkozhatunk a nem szabványos `embed` tagként való elhelyezéssel. Meg kell adni a fájl típusát is, mely sokféle lehet, például `audio/mp3`, `audio/mpeg`, `application/x-shockwave-flash`, `video/quicktime` stb. Bizonyos esetekben Internet Explorerben a fájltípust a Windows-os `classid` értékeként kell megadni. Egy flash fájl beágyazása például így nézhet ki:

```
<object data="ai.swf" type="application/x-shockwave-flash"
        width="288" height="128">
 <param name="movie" value="ai.swf" />
 <p> Akkor írja ki, ha nem tudja a Flash-t megjeleníteni. </p>
</object>
```

A `param` tulajdonsággal a fájl típusától függően számos paramétert lehet beállítani: `autoplay`, `controller`, `autostart`, `showcontrols`, `showstatusbar`, `autorewind`, `src` (ha az `object`-ben nem adtuk meg `data`-val) stb. A beágyazás további részletei nagyon függnek a multimédiás fájl típusától, így azok részletezésétől a jegyzet korlátai miatt eltekintünk.

2.6. Listák

A listákat felsorolásokra használjuk és XHTML-ben három típusuk van: egyszerű (felsorolásjeles) listák, számozott listák és meghatározás listák. A listák egymásba is ágyazhatók. Alapértelmezés szerint a listák és azok felsorolt elemei blokkszintű objektumok. A listák formázásához használhatjuk az eddig megismert CSS stílusokat is. A továbbiakban csak a listákra jellemző speciális tulajdonságokat ismertetjük.

2.6.1. Egyszerű listák

Egy egyszerű (felsorolásjeles) lista (angolul **u**nordered **l**ist) tartalma `<ul>` és `</ul>` tagok között helyezkedik el és felsorolt elemeket tartalmaz. Minden felsorolt elemet (**l**ist **i**tem) `<li>` **és** `</li>` tagok között kell elhelyezni. Például: `<ul> <li>liszt</li> <li>tojás</li> <li>só</li> <li>cukor</li> <li>vaj</li> </ul>`. Az `ul` objektum `list-style-type` CSS tulajdonságának értékeként megadhatjuk, hogy a felsorolt elemek előtt milyen jeleket jelenítsen meg: `disc` (teli kör), `circle` (üres kör), `square` (négyzet). Felsorolásjelként használhatunk képet is, ebben az esetben a `list-style-image` CSS tulajdonságot használjuk. Például `list-style-image: url(kep.gif)`. A kép relatív útvonalát stíluslap helyéhez viszonyítva kell megadni. Az `ul` objektum `list-style-position` CSS tulajdonságának alapértelmezett `outside`. illetve `inside` értékével megadhatjuk azt is, hogy a felsorolás-elemek a lista dobozán kívül vagy belül helyezkedjenek el. A lista tulajdonságait összevontan is megadhatjuk a `list-style` tulajdonság értékeként a következő sorrendben: `list-style-type`, `list-style-image`, `list-style-position`.

2.6.2. Számozott listák

Számozott lista (**o**rdered **l**ist) elemei előtt arab vagy római számok, illetve betűk állhatnak, tartalma pedig `<ol>` és `</ol>` tagok között helyezkedik el. A felsorolt elemek az egyszerű listákhoz hasonlóan `<li>` és `</li>` tagok között kell legyenek.

A sorszámok típusának megadására a `list-style-type` CSS tulajdonságot használjuk, melynek értékei a következők lehetnek: `none` (nincs), `decimal` (arab szám), `decimal-leading-zero` (arab szám kezdő nullával), `upper-alpha` (nagybetűk), `lower-alpha` (kisbetűk), `upper-roman` (római számok nagybetűkkel írva), `lower-roman` (római számok kisbetűkkel írva).

Alapértelmezés szerint a lista címkéi arab számok és az elemek számozása 1-el, illetve A vagy a betűkkel kezdődik. Ettől eltérő kezdő értéket az `ol` objektum `start` jellemzőjével adhatunk meg, melynek értéke arab szám kell legyen akkor is, ha a számozás római szám vagy betű lesz. Például `<ol start= ”3”> <li> harmadik </li> <li> negyedik </li> </ol>`. A `start` jellemző azonban csak átmeneti szabvány szerint használható, ezért erre egy szabályosabb, de bonyolultabb megoldást a számlálóknál fogunk látni.

2.6.3. Meghatározás listák

Definíciók leírására meghatározás listákat használhatunk, melyek minden felsorolt eleme két részből áll: a definiálandó fogalomból és a meghatározásból. A teljes meghatározás listát (**d**efinition **l**ist) `<dl>` és `</dl>` tagok között helyezzük el. Minden felsorolt elem esetén a definiált fogalmat (**d**efinition **t**erm) `<dt>` és `</dt>`, míg a meghatározást (**d**efiniton **d**ata) `<dd>` és `</dd>` között kell elhelyezni. Lássunk egy olyan meghatározás listát, melyben két fogalmat „definiálunk”:

```
<dl>
  <dt> Hardver: </dt>
  <dd> a számítógép fizikailag megfogható részeinek összessége
</dd>
  <dt> Szoftver: </dt>
  <dd> a számítógép memóriájában elhelyezkedő, azokat működtető program </dd>
</dl>.
```

2.6.4. Listák használata menük készítésére

A listákat gyakran szokták menük készítésére használni, mivel a menüpontok felsorolt elemeknek tekinthetők. Az alábbiakban megnézünk néhány alaptechnikát, melyek menük készítése során használhatók.

Először is, a felsorolt elemek hivatkozások lesznek. Általában nem akarjuk, hogy ezek szövege aláhúzottan jelenjen meg, tehát érdemes `text-decoration: none;`-al ezt letiltani. Ha háttérváltoztató menüpontokat akarunk készíteni, akkor a `hover` kijelölővel kell megfelelő formázást hozzárendelnünk. Egy megfelelő stílusú szegéllyel (pl. `outset`) a hivatkozásoknak gombszerű kinézetet is adhatunk. Érdemes `padding`-al eltávolítani a hivatkozás tartalmát a szegélytől, a `<li>` tagoknak pedig `margin` tulajdonságot megadni azért, hogy az egyes menüpontok is eltávolodjanak egymástól. Valószínűleg nem szeretnénk, hogy a lista elemeiként felsorolt menüpontok előtt felsorolásjelek legyenek, ezt a `list-style-type:none;`-al lehet letiltani.

A lista elemei blokkszintűek, alapértelmezésként egymás alatt fognak megjelenni, így függőleges menüt fogunk kapni. Hogyan lehet vízszintes menüt készíteni listával?

Az egyik lehetséges megoldás, hogy előírjuk a menüpontok (`<li>` tagok) úsztatását, ily módon azok egymás mellé fognak kerülni. A másik lehetőség a `<li>` tagok megjelenítésének módosítása sorszintűvé. De hogyan lehetséges ez?

© Holló Csaba, SzTE

A `display` CSS tulajdonság `block`, illetve `inline` értékeivel előírható, hogy egy objektum blokkszintű vagy sorszintű elemként viselkedjen. Ily módon elérhető, hogy a normál esetben blokkszintű elemek sorszintűként, a normál esetben sorszintű elemek pedig blokkszintűként jelenjenek meg. Mindez természetesen a listákra is érvényes. Tehát a `<li>` tagok `display` tulajdonságát `inline`-ra állítva is készíthetünk vízszintes menüt.

Végül meg fogjuk nézni, hogy miként tudunk többszintű, egér fölé vitelekor előbukkanó menüt készíteni. Az egyértelműség kedvéért feltételezzük, hogy a menüket `ul` listaként valósítjuk meg, a megoldás könnyen alkalmazható lesz más esetben is. Többszintű menüt úgy kapunk, hogy a főmenü valamely eleme, a saját feliratán kívül, a `<li>` és `</li>` tagok között egy teljes újabb listát (például `<ul>` … `</ul>`) tartalmaz. A kérdés az, hogy hogyan érhetjük el azt, hogy ez a beágyazott lista csak akkor legyen látható, amikor az egeret az adott menüpont fölé visszük? Alapértelmezésképpen elrejtjük:

```
ul ul {display: none;},
```

ugyanakkor előírjuk, hogy amikor a felhasználó az egeret a menüpont fölé viszi, akkor jelenjen meg:

```
ul li:hover ul { display: block; }.
```

Hogyha a beágyazott menübe is be van ágyazva még egy menü, akkor:

```
ul li:hover ul ul { display: none; } és
ul ul li:hover ul { display: block; }.
```

Az ilyen menük készítésekor azonban vegyük figyelembe, hogy a felhasználó nem memóriajátékot játszik, tehát csak akkor használjuk, ha az eredetileg nem látszó menüpontok nagyon könnyen megjegyezhetők.

Természetesen sok ötletet lehetne még írni a menük szépítésére, de a jegyzet korlátai miatt a téma további részletezésétől eltekintünk.

2.7. Számlálók

CSS-ben definiálhatunk számlálókat ([49], [50]), melyek segítségével automatikusan számozhatjuk például a fejezeteket, alfejezeteket és a felsorolások elemeit.

A számláló neve tetszőleges azonosító név lehet. Első példánkban `cim` és `alcim` nevű számlálók lesznek. Használat előtt ajánlott a számlálókat `counter-reset`-el inicializálni. Például: `body {counter-reset: cim;} h2 {counter-reset: alcim;}`. De hogyan fogjuk a számlálókat kiíratni?

Lehetőség van arra, hogy `:before`, illetve `:after` használatával előírjuk adott elemek előtt vagy után tetszőleges tartalom automatikus megjelenítését. Továbbá egy számláló értékét a `counter(szamlalo)` adja meg. Így tehát ha a `h2`-es szintű címsorok előtt meg szeretnénk jeleníteni a fejezet számát, akkor ezt írjuk:

```
h2:before { content: counter(cim) ". fejezet:"; }.
```

Csakhogy így a következő `h2`-es fejezetnek is ugyanolyan száma lenne, ezért a fejezet számának kiírásakor növelnünk kell a `cim` fejezetszámlálót. Továbbá, ha azt szeretnénk, hogy az elfejezetek számozása az új fejezetben újrakezdődjön, akkor az alfejezet számlálót (`alcim`) újra inicializálnunk kell. Tehát a következőt tesszük:

```
h2 {counter-increment: cim; counter-reset: alcim;}
h2:before {content: counter(cim) ". fejezet:";}
h3 {counter-increment: alcim;}
h3:before {content: counter(cim) "." counter(alcim) " ";}
```

Az <ol> tag start jellemzőjének használata is kiváltható számláló segítségével. Ehhez a beazonosított <ol id="elso">-ben tetszőleges értékkel inicializáljuk a számlálót, melyet minden <li> előtt megjelenítünk és növelünk.

```
ol#elso {counter-reset: szamlalo 2;}
ol#elso li:before {content: counter(szamlalo) ". ";
                   counter-increment: szamlalo;}
```

Végül, ha a <li> elemek a display tulajdonságát beállítjuk block vagy inline értékre, akkor azok automatikus számozása nem fog megjelenni.

2.8. Táblázatok

A táblázat adatoknak bizonyos szempontok szerint egymás mellé, illetve egymás alá írása következtében létrejövő sorokba és oszlopokba rendezett összessége.

A táblázatok soraiban és oszlopaiban levő tartalmak vízszintes illetve függőleges igazítása nagyon egyszerűen megvalósítható, ezért gyakran szokták a táblázatokat az oldalak elrendezésének kialakítására is használni. Mégis azt lehet tanácsolni, hogy kerüljük ezt a megoldást, mert bizonyos böngészők, illetve más (például felolvasó) programok helytelenül értelmezhetik a táblázatban elrendezett tartalmat (például nem megfelelő sorrendben olvassák azt fel), ezért az érintett felhasználók dolgát nagyon megnehezítjük. A weboldalak elrendezésének kialakítása megvalósítható a már ismertetett helyzetmegadások segítségével ([41], [42]).

2.8.1. Táblázatok felépítése

XHTML-ben a táblázat sorokból és azokon belül cellákból áll. Az egymás melletti cellák alkotják a sorokat, az egymás alattiak pedig az oszlopokat. A táblázat teljes tartalma <table> és </table>, egy sor <tr> és </tr> tagok között helyezhető el. Egy cella lehet <th> és </th> között elhelyezkedő fejléc információkat tartalmazó cella, vagy <td> és </td> között adatokat tartalmazó cella. A táblázatnak lehet egy <caption> és </caption> között leírt címsora, melynek közvetlenül a <table> tag után kell elhelyezkednie (akkor is, ha a táblázat alatt akarjuk megjeleníteni).

A táblázat három strukturális részre osztható. <thead> és </thead> között fejlécsorok csoportját, <tfoot> és </tfoot> között pedig láblécsorok csoportját határozhatjuk meg, melyek több oldalas táblázatoknál nyomtatáskor szabvány szerint minden oldalon megjelennek. <tbody> és </tbody> között adatsorok csoportját írjuk le.

Egy táblázat egyetlen fejlécsor csoportot és egyetlen láblécsor csoportot tartalmazhat, továbbá a tfoot tagnak a thead után és a tbody előtt kell lennie. Egy táblázatnak tartalmaznia kell legalább egy adatsor csoportot. Ha a táblázat csak egy ilyen csoportot tartalmaz, és nem tartalmaz sem thead-et, sem tfoot-ot, akkor a tbody megadása elhagyható, de a szkriptekkel történő korrekt feldolgozhatóság érdekében akkor sem ajánlott.

Lehetőség van arra, hogy adat vagy fejléc cellát több oszlop szélességűnek, vagy több sor magasságúnak adjunk meg. Ehhez az adott cella rowspan, illetve colspan jellemzőinek értékeként megadjuk a kívánt oszlopok vagy sorok számát. Utóbbi esetben az alatta levő sorokban az adott cellának nem kell ismételten td vagy th tagot megadni, az automatikusan elfoglalja a neki megfelelő helyet.

Az üres cellákba a helyes formázás érdekében ajánlott egy szóközt (-t) beírni.

A táblázatok esetében több jellemző is megadható, melyek azt a cél szolgálják, hogy a böngésző számára átláthatóbb legyen a táblázat tartalma és a felolvasó szoftver azt minél érthetőbben fel tudja olvasni. A table objektum summary tulajdonságának értékeként felolvasó szoftverek számára írt tájékoztató szöveget adhatunk meg a táblázatról, mely egyébként

nem jelenik meg. A cellákat a td és th tagok axis jellemzőjével a fejlécektől eltérő kategóriákba is sorolhatjuk, amelyek alapján más programok (pl. felolvasó szoftverek) jobban tudják értelmezni és feldolgozni őket. Például axis = "Italok". Az adatok fejlécekhez társításának megkönnyítése érdekében, a th tag scope jellemzőjének col, illetve row értékével megadhatjuk, hogy az sor vagy oszlop fejléce. Azonban ennek csak akkor van értelme, ha nincsenek több sor vagy oszlop méretű cellák. Egyébként a cellák headers jellemzőjét használhatjuk, melynek értékeként azon fejléc cellák egyedi azonosítóit sorolhatjuk fel szóközzel elválasztva, melyekhez annak tartalma tartozik.

2.8.2. Táblázatok formázása

A táblázat címsora elhelyezkedhet a táblázat fölött vagy a táblázat alatt. Ezt a caption tag caption-side CSS tulajdonságának top, illetve bottom értékével tudjuk szabályozni.

Az egész táblázat, illetve az egyes cellák szélességét a width tulajdonsággal tudjuk megadni. A táblázat, illetve a cellák köré szegélyeket rajzolhatunk, melyek formázására a szegélyeknél leírtak érvényesek. A táblázatok esetén azonban a table tag border-collapse CSS tulajdonságának collapse, illetve separate értékeivel megadható, hogy az egymás melletti cellák szegélyei összevonásra kerüljenek-e vagy sem. Utóbbi esetben a table tag border-spacing tulajdonságával az is megadható, hogy a cellák milyen távolságra legyenek egymástól.

A cellák text-align, illetve vertical-align CSS tulajdonságával megadható azok vízszintes, illetve függőleges igazítása. Utóbbi esetben a top értékkel felülre, middle-vel középre, baseline-al a sorvezetőhöz, bottom-al pedig a teljes szöveg aljához történő igazítást írhatunk elő.

2.9. Űrlapok

2.9.1. Az űrlapok működése

Az űrlapok arra szolgálnak, hogy a felhasználóktól adatokat kérjünk ([54]). Természetesen ennek akkor van értelme, ha ezeket az adatokat el is tároljuk vagy feldolgozzuk, ezt pedig valamilyen programmal tudjuk megtenni. Jelen fejezetben azt nézzük meg, hogy hogyan tudjuk bekérni az adatokat és azokat továbbítani a megfelelő programhoz vagy a programot tartalmazó weboldalhoz.

Egy űrlap teljes tartalma a <form> és </form> tagok között helyezkedik el.

A form tag action jellemzőjének értékeként kell megadnunk az űrlap elküldésekor végrehajtandó feladatot. Például action= "mailto:chollo@inf.u-szeged.hu" esetén az adatok adott e-mail címre történő küldését,
action="http://szerver.hu/cgi-bin/feldolgoz.cgi" esetén az adatok feldolgoz.cgi programmal történő feldolgozását írjuk elő.

A form tag method jellemzőjével az adatok elküldésének módszerét adhatjuk meg. Az alapértelmezett get érték esetén az adatok az URL-ben kerülnek elküldésre, post érték esetén az adatok a program standard bemenetére kerülnek. Például csatolt fájl küldésénél a post értéket használjuk.

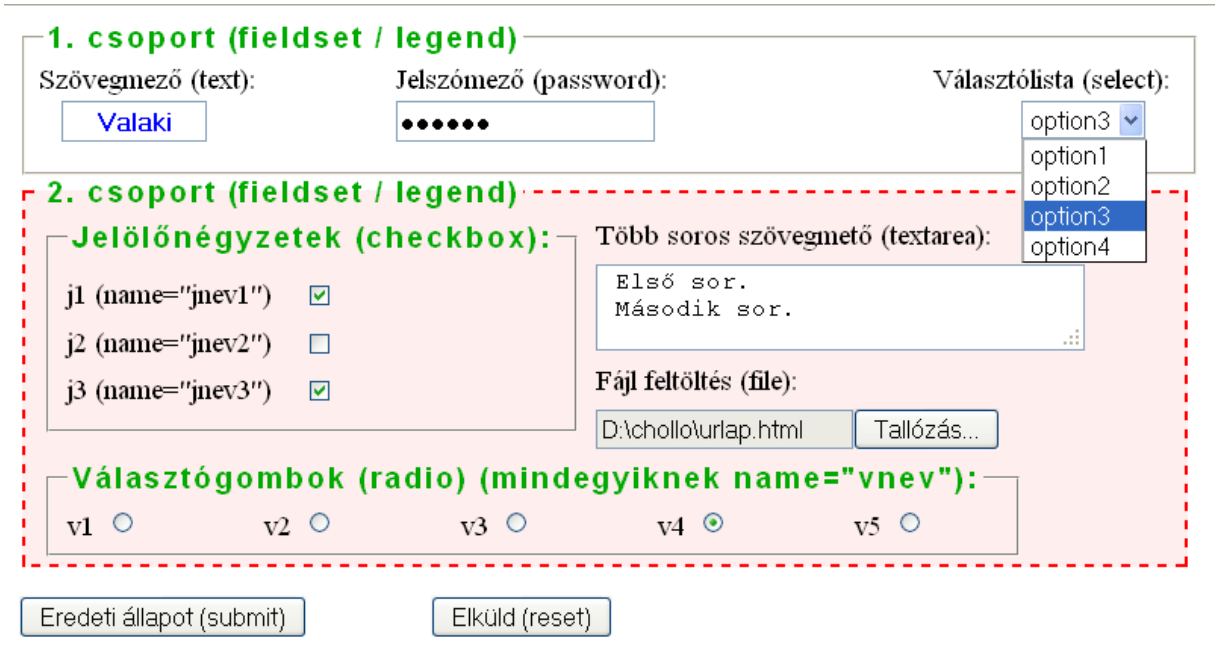
Az elküldendő adatokat kódolni is kell. Az adatok kódolási módszerét a form tag enctype jellemzőjének értékeként adhatjuk meg. Például levélküldéskor a kódolás lehet text/plain, ha az űrlapunk csatolt fájlt is tartalmaz, akkor multipart/form-data. Az alapértelmezett application/x-www-form-urlencoded kódolás az adatokból

egy olyan karaktersorozatot készít, melyben `&` jellel elválasztott *mezőnév=érték* párok lesznek, ahol a *mezőnév* az adott mező `name` jellemzőjének az értéke, az *érték* pedig az adott mezőben felhasználó által kiválasztott vagy beírt érték. A karaktersorozatban az űrlap minden alapból létező vagy felhasználó által kiválasztott mező szerepel, több értékkel rendelkező mező pedig többször is, minden értékével benne lesz. A karaktersorozatban a szóközök + jelekre cserélődnek, ugyanakkor a speciális karaktereket egy `%XX` típusú jelre cseréli, ahol `XX` a karakterek hexadecimális kódja (ISO-LATIN szerint).

Az URL-ben továbbított karaktereket azért kell kódolni, mert csak bizonyos karakterek engedélyezettek (RFC1738, [19]), illetve amik használhatók, azok sem mind biztonságosak (RFC2396, [20], [21]), például a Unicode karakterek sem. Ily módon kódolandó speciális karakternek minősül minden olyan karakter, mely az alábbi kategóriák valamelyikébe esik:

- nem nyomtatható karakterek (ASCII Control characters)
- nem ASCII karakterek (non-ASCII characters, például ékezetes karakterek,)
- fenntartott karakterek (reserved characters: ';', '/', '?', ':', '@', '&', '=', '+', '$', ',')
- határoló és nem biztonságos karakterek (excluded US-ASCII characters: szóköz, '<', '>', '#', '%', '"', '{', '}', '|', '\', '^', '[', ']', '`').

A 3. ábrán láthatunk egy űrlapot, melyen a mezők feliratként azok típusát tűntetjük fel.



3. ábra: Űrlap

2.9.2. Egy értéket kérő mezők

Egy értéket bekérő mezőket XHTML-ben `<input />` tag-al tudunk megvalósítani, melynek `type` jellemzőjével adjuk meg a mező típusát. Nagyon fontos a `name` jellemzővel minden mezőnek nevet adni, ugyanis a mező tartalma ezzel a névvel lesz elküldve, tehát a feldolgozó program ebből fogja tudni, hogy az elküldött érték mely mezőhöz tartozik. A `value` jellemzővel a mezőnek egy alapértelmezett értéket is adhatunk, mely gombok esetén a gomb felirata lesz. A `disabled` jellemző `true` vagy `false` értékével előírhatjuk, hogy a mező le legyen-e tiltva. Alapértelmezett értéke `false`, `true` esetén pedig a letiltást a böngészők több-

nyire formázással is (például halványítással) fogják jelezni. A `readOnly` jellemző `true` értékével letiltható a mező módosítása. Az alapértelmezett érték a `false`, ami annak felel meg, hogy a mező módosítható.

Rövid szöveges érték bevitelére szolgáló *szövegmezőt* az `<input type="text" />` taggal hozhatunk létre. Numerikus értékek bekérése is ezzel lehetséges, ezért szükség lehet az érték megfelelő numerikus típusra való átalakítására.

A szövegmező egy speciális fajtája a *jelszómező*, amikor a `type` jellemző értékének `password`-öt adunk meg. Ebben az esetben a beírt karakterek a mezőben nem lesznek láthatók, viszont figyelnünk kell arra, hogy a bekért érték nem titkosított, ezért az elküldés előtti titkosításról nekünk kell gondoskodnunk.

Szöveg és jelszómezők szélességét a `size` jellemzővel adhatjuk meg. Viszont a mező szélessége nem a beírható, hanem az azokból egyszerre látható karakterek számát korlátozza. Ugyanakkor kellemetlen helyzetek fakadhatnak abból, ha a felhasználónak nagyon sok karakter beírását megengedjük, viszont az adatbázisban való eltároláskor azok egy részét elvetjük. Ezért célszerű korlátozni a beírható karakterek számát is, melyet a `maxlength` jellemző értékeként adhatunk meg.

A `type` jellemző `hidden` értéke esetén *rejtett mezőt* hozunk létre, mely nem látszik a képernyőn, de értékét a program megkapja és feldolgozhatja. Ilyen mezőt technikai célokból használhatunk, később látni fogunk ilyen alkalmazást.

A *választógombok (rádiógombok)* olyan mezők, melyeknek csak két értékük lehet: kiválasztott és nem kiválasztott, továbbá csoportosíthatók oly módon, hogy a csoportból csak egy mező kiválasztása legyen lehetséges. Ilyen mezőket olyan válaszok megadására használunk, melyek esetén több válaszlehetőség közül csak egy választható ki. Választógombot a `type` jellemző `radio` értéke esetén kapunk, egy csoportba pedig az azonos `name` értékkel rendelkező választógombok fognak tartozni.

Olyan kérdések válaszainak megvalósítására, amikor a felsoroltak közül egy vagy több válasz is megjelölhető, *jelölőnégyzeteket* használunk. A jelölőnégyzetnek is két értéke van: kiválasztott és nem kiválasztott, viszont minden jelölőnégyzet kiválasztása a többitől független. Jelölőnégyzetek esetén a `type` jellemző értéke `checkbox` kell legyen.

A választógombok és jelölőnégyzetek alapértelmezés szerint nem kiválasztottként jelennek meg és a felhasználó dolga, hogy az általa kívántakat kiválassza. Ha azt szeretnénk, hogy valamely választógomb vagy jelölőnégyzet már alapból kiválasztottként jelenjen meg, akkor ezek `checked` jellemzőjének `checked` értéket kell adnunk.

Lehetőség van űrlapokon olyan mezőt is megadni, mely segítségével a felhasználó fájlt csatolhat az elküldendő adatokhoz. Az ilyen mező `type` jellemzőjének értéke `file` és többnyire egy tallózó gombot is megjelenít, mellyel kiválasztható a csatolandó fájl.

Az űrlapon elhelyezhetünk olyan gombot is, melyre rákattintva a felhasználó visszaállíthatja a mezők eredeti értékeit, érvénytelenítve az általa eszközölt beírásokat és kiválasztásokat. Az ilyen gomb `reset` típusú mezőnek felel meg és `value`-val megadható a gomb felirata is.

Az űrlapon levő adatok elküldésére szolgáló gombot `submit` típusú input mezővel lehet megvalósítani, melyből akár több is lehet egy űrlapon, és melynek felirata úgyszintén a `value` jellemző értékeként adható meg. Ugyanakkor `image` típusú input mező segítségével lehetőség van arra is, hogy elküldés gomb helyett egy képet jelenítsünk meg, melyre kattintva az adatok elküldődnek. Ebben az esetben az `input` objektum nyitó tagjában meg kell adnunk az `src` jellemző értékeként a képfájlt, `alt` értékeként a kép hiányában megjelenő szöveget,

továbbá `width` és `height`-tal megadhatjuk a kép méretét. Az elküldött adatokhoz hozzáadódnak a felhasználó képen belüli kattintásának koordinátái is.

Végül `button` típusú `input` mezővel olyan általános célú nyomógombot is létrehozhatunk, melynek tetszőleges funkcionalitást programmal adhatunk.

2.9.3. Több soros szövegmező

A *több soros szövegmezőt* hosszabb szöveges információ (például a felhasználó észrevételeinek) bekérésére használjuk. Megvalósítása a `textarea` taggal történik, mely egy téglalap alakú területet jelenít meg, melybe a felhasználó szöveges információt írhat. A téglalap méretét, azaz sorainak és oszlopainak számát a `textarea` tag `rows`, illetve `cols` jellemzőinek értékeként tudjuk megadni. A mezőben alapértelmezésként megjelenítendő szöveget a `<textarea>` és `</textarea>` tagok közé kell beírni. Például: `<textarea name="szoveg" rows = "7" cols = "30"> Ez alapból megjelenik. </textarea>`.

2.9.4. Választólista

A *választólista* egy olyan mező, melyben előre felsorolt lehetőségek közül választhatja ki a felhasználó a számára megfelelőt. A választólista teljes tartalmát `<select>` és `</select>` tagok között helyezzük el. A `select` objektum `multiple` jellemzőjének `multiple` értékével engedélyezhetjük több elem kiválasztását is, azonban ezt a lehetőséget a felhasználóval is tudatnunk kell, ugyanis a választólista formázásából ez nem fog kiderülni.

A `select` tag `size` jellemzőjével megadhatjuk, hogy hány választási lehetőség legyen látható egyszerre. Ezen jellemző hiányában az alapértelmezett érték 1 és ebben az esetben az egyetlen választási lehetőség melletti nyílra kattintva egy lista fog lenyílni, amelyben a többi választási lehetőség is látható és kiválasztható lesz. A `size` jellemző egynél nagyobb értéke esetén a lehetőségeket az eredeti mezőben, a mező jobb oldalán elhelyezkedő nyilak segítségével lehet megnézni és kiválasztani. Minden kiválasztható lehetőséget `<option>` és `</option>` tagok között helyezünk el. Azon lehetőségnél, melyeket már eleve kiválasztottként szeretnénk megjeleníteni, az `option` tag `select` jellemzőjét meg kell adni a `select` értékkel. Minden `option` tag `value` jellemzőjének értékeként megadhatjuk azt az értéket, amely az adott lehetőség kiválasztása esetén a `select` mező értékeként továbbítódni fog a feldolgozó programnak. Amennyiben valamely kiválasztott lehetőség `option` tagjának nincs `value` jellemzője, akkor annak `select` értékeként az `<option>` és `</option>` tagok közötti tartalom továbbítódik.

2.9.5. Elemfelirat

Ha csak az űrlapmezőket jelenítenénk meg, a felhasználó nem tudná, hogy mit kell azokba beírni, ezért azok szomszédságában (mellettem, fölötte, esetleg alatta) magyarázó szöveget is megjelenítünk. Mivel helyzetmegadással bármit, bárhová pozícionálhatunk, a magyarázó szöveg az XHTML kódban bárhol elhelyezkedhet, így az sem biztos, hogy a megfelelő űrlapmezőt megvalósító objektum mellett van. Emiatt a böngésző nem tudhatja, hogy melyik mezőhöz milyen feliratot szántunk, márpedig bizonyos helyzetekben, például felolvasószoftverek használata esetén, fontos ezt egyértelműen meghatározni. Az űrlapmező és az ahhoz tartozó felirat összekapcsolása érdekében a feliratot `<label>` és `</label>` tagok közé kell helyezni, továbbá a `label` tag `for` jellemzőjének értékeként meg kell adni a felirathoz tartozó űrlapmező egyedi azonosítójának értékét ([51], [52]) Ebben az esetben mindegy, hogy az XHTML kódban az űrlapmező és felirata milyen távol vannak, egymástól, a kettő közötti

kapcsolat egyértelmű. Az elemfelirat használatának további előnye, hogy az elemfeliratra kattintva a hozzá tartozó űrlapmező megkapja a vezérlést, így például választógombok és jelölőnégyzetek esetén a felhasználó a négyzet vagy kör helyett az elemfeliratra kényelmesebben kattinthat.

Például:

```
<div id = "felirat"> <label for="nev"> Név: </label> </div>
<div id = "egyeb"> … </div>
<div id="neve">
  <input type="text" id="nev" name = "nevecske" />
</div>
```

2.9.6. Mezőcsoportosítás

A logikailag összetartozó űrlapmezőket csoportosíthatjuk ([51]) oly módon, hogy `<fieldset>` és `</fieldset>` tagok közé tesszük. A csoportnak nevet is adhatunk, melyet `<legend>` és `</legend>` tagok között helyezhetünk el.

2.9.7. Űrlapok formázása

Az űrlapok formázására a szokásos CSS formázási módszereket alkalmazzuk. Érdemes figyelni arra, hogy az űrlapelemek hátterét nem minden böngésző jeleníti meg.

2.10. Billentyűzettel történő vezérlés megvalósítása

2.10.1. Váltás a tabulátor billentyűvel

A tabulátor billentyű nyomkodásával a fókusz végighalad a vezérlőelemeken ([5], [53], [54]). A sorrendet az egyes objektumok `tabindex` jellemzőjének megadásával állíthatjuk be, melyet a következő vezérlőelemekre adhatunk meg: `a`, `area`, `button`, `input`, `object`, `select`, `textarea`.

A `tabindex` értéke egy 0 és 32767 közötti egész szám kell legyen. Például: `<input type = "text" tabindex = "2">`. Először azok a vezérlőelemek fogják megkapni a fókuszt a `tabindex` értékek növekvő sorrendjében, melyekhez rendelve van 0-nál nagyobb `tabindex` érték. A 0 `tabindex` érték ekvivalens azzal, mint hogyha nem is rendeltünk volna hozzá értéket. A letiltott (`disabled`) elemek nem fognak szerepelni a bejárásban. A vezérlőelemek bejárásánál a választógombok esetén csak az egyik gomb kerül sorra, a többire a kurzorvezérlő billentyűkkel léphetünk. A jelölőnégyzetek állapotát a felhasználó a szóköz billentyűvel is megváltoztathatja.

2.10.2. Gyorsbillentyűk definiálása

A gyorsbillentyű egy olyan billentyű, melynek megfelelő billentyűkombinációban való lenyomása kiválasztja egy adott elemet, parancsgomb esetén a létrehozza azt az eseményt, mintha a felhasználó egérrel rákattintott volna és jelölőnégyzetnél megváltoztatja a bejelölés állapotát. De hogyan lehet a gyorsbillentyűt használni? Internet Explorerben, Google Chromeban és Safariban az Alt billentyűvel, Firefoxban az Alt és Shift billentyűkkel együtt kell lenyomni. Opera esetén a Shift és Esc billentyűk együttes lenyomásával billentyűmódba kell váltani, majd azután kell a definiált gyorsbillentyűt lenyomni.

Gyorsbillentyűt az `accesskey` tulajdonsággal rendelhetünk az objektumokhoz. Az `accesskey` értéke az a karakter, melyet böngészőtől függő billentyűkombinációban lenyomva az objektum megkapja a fókuszt. Gyorsbillentyűt a következő objektumokra definiálhatunk: `a`, `area`, `button`, `input`, `label`, `legend`, `textarea`. Űrlapelemek esetén a

gyorsbillentyűt az elemfelirathoz rendeljük hozzá. Vigyázni kell arra, hogy a böngészőkben eleve definiált billentyűkombinációktól különbözőket definiáljunk. Továbbá tájékoztatni kell a felhasználót arról, hogy az egyes objektumokhoz milyen gyorsbillentyűket definiáltunk, ezt többnyire az adott betű eltérő formázásával szoktuk jelezni.

2.11. Keretek

2.11.1. Felosztó keretek

Az oldalt több részre fogjuk osztani ([55]), melyekben tetszés szerinti weboldalak tartalmát jeleníthetjük meg. A beágyazott weboldalak lehetnek mind különbözőek, de megtehetjük azt is, hogy több keretben ugyanazt a weboldalt ágyazzuk be. Ily módon szükségünk lesz a beágyazandó oldalakat leíró fájlokra és egy speciális, az őket tartalmazó oldal struktúráját leíró fájlra. A beágyazott oldalak önmagukban is lehetnek keretek, tehát a hierarchia több szintű is lehet.

A beágyazó oldal dokumentumtípusa keretkészletes kell legyen ([56]). Az oldal törzsrésze `<body>` és `</body>` helyett `<frameset>` és `</frameset>` között fog elhelyezkedni, melybe további `frameset` objektumok ágyazhatók. Egy `frameset` objektummal sorokra vagy oszlopokra oszthatjuk az oldal megfelelő részét (mindkettőre egyetlen `frameset`-tel nem lehet), oly módon hogy annak `cols` vagy `rows` jellemzőjének értékeként vesszővel elválasztva felsoroljuk az oszlopok vagy sorok méreteit, melyből azok száma is egyértelműen kiderül. Egy `frameset` objektumba további `frameset` objektumokat vagy `<frame />` taggal megvalósított kereteket ágyazhatunk be, annyit, ahány részre osztottuk az adott területet. A legfelsőbb szintű frameset objektumba `<noframes>` `<body>` és `</body>` `</noframes>` között olyan XHTML tartalmat helyezhetünk el, mely akkor jelenik meg, ha a böngésző a kereteket nem támogatja.

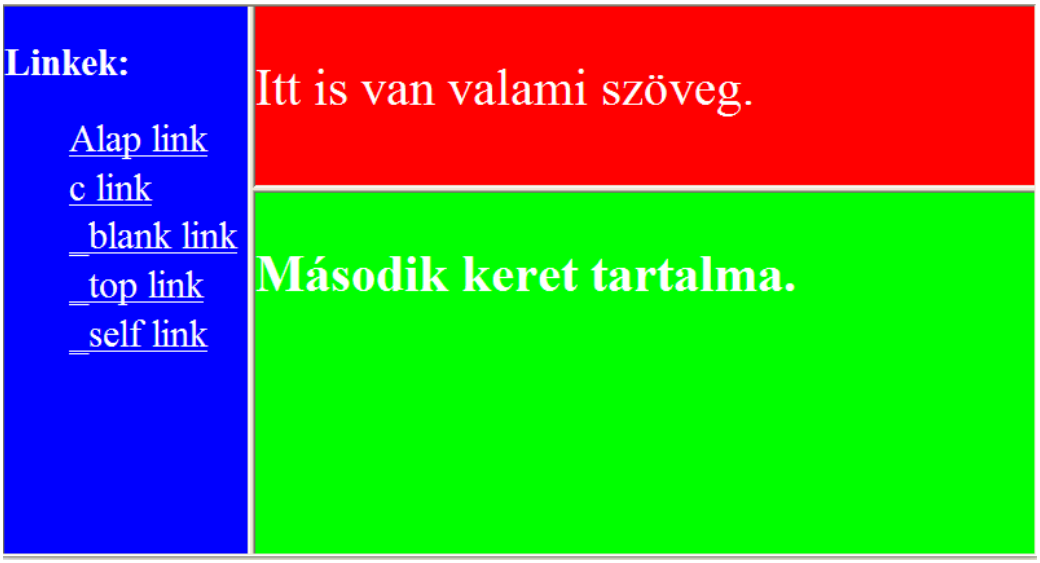
Az előzőekben említésre került, hogy egy keretet a `<frame />` taggal definiálunk. Minden keret esetén az `src` jellemző értékeként meg kell adnunk annak a dokumentumnak az URL-jét, amelyiket meg szeretnénk jeleníteni az adott keretben. A keretnek adhatunk egy nevet is a `name` jellemzővel, ennek akkor van jelentősége, ha egy másik keretben elhelyezett hivatkozás célját az adott keretben szeretnénk megjeleníteni. A `noresize` jellemző `noresize` értékével megtiltható a keret felhasználó általi átméretezése, a `scrolling` jellemző `yes`, `no`, illetve `auto` értékeivel szabályozhatjuk, hogy a keret körül jelenjen-e meg gördítő sáv: `yes` érték esetén mindig lesz, `no` érték esetén sohase lesz, az alapértelmezett `auto` érték esetén pedig akkor lesz, ha a tartalom gördítése szükségessé válik. A gördítősáv letiltása valamennyivel több férőhelyet eredményez az oldalon, de más felbontás használata vagy a karakterek nagyítása esetén lehetetlenné teheti a tartalom egy részének elérését. Lássunk egy példát:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<link rel="stylesheet" type="text/css" href="keret1.css" />
<title> Keretek </title>
<meta http-equiv="Content-Type"
      content="text/html;charset=ISO-8859-2" />
</head>
<frameset cols = "24%,*">
```

 © Holló Csaba, SzTE

```
  <frame src = "kek.html" />
  <frameset rows = "*,2*">
    <frame name = "b" src ="piros.html"
           noresize = "noresize" />
    <frame name = "c"src ="zold.html" />
  </frameset>
  <noframes>
    <body>
      <p> Ezt írja ki, ha nem ismeri a kereteket.</p>
    </body>
  </noframes>
</frameset>
</html>
```

A megjelenő oldal a 4. ábrán látható.



4. ábra: Kereteket tartalmazó oldal

Amint láthatjuk, a rows értékeként megadott `"*,2*"` a területet 1/3, 2/3 részben osztja fel.

A hivatkozásokat megvalósító `<a>` tag `target` jellemzőjének értékeként meg lehet adni, hogy a célt melyik keretben vagy egy másik ablakban nyissa meg. Ha a célt egy másik keretben akarjuk megjeleníteni, akkor `target` értékeként a másik keret `name` jellemzőjének értékét kell megadnunk. A célt a `target` jellemző `"_blank"` értéke esetén új ablakban vagy lapon, `"_self"` érték esetén az aktuális dokumentumot tartalmazó keretben, míg `"_top"` érték esetén az eredeti ablakban, az összes keretet felülírva jeleníthetjük meg. A `target` jellemző azonban csak átmeneti dokumentumtípusban használható.

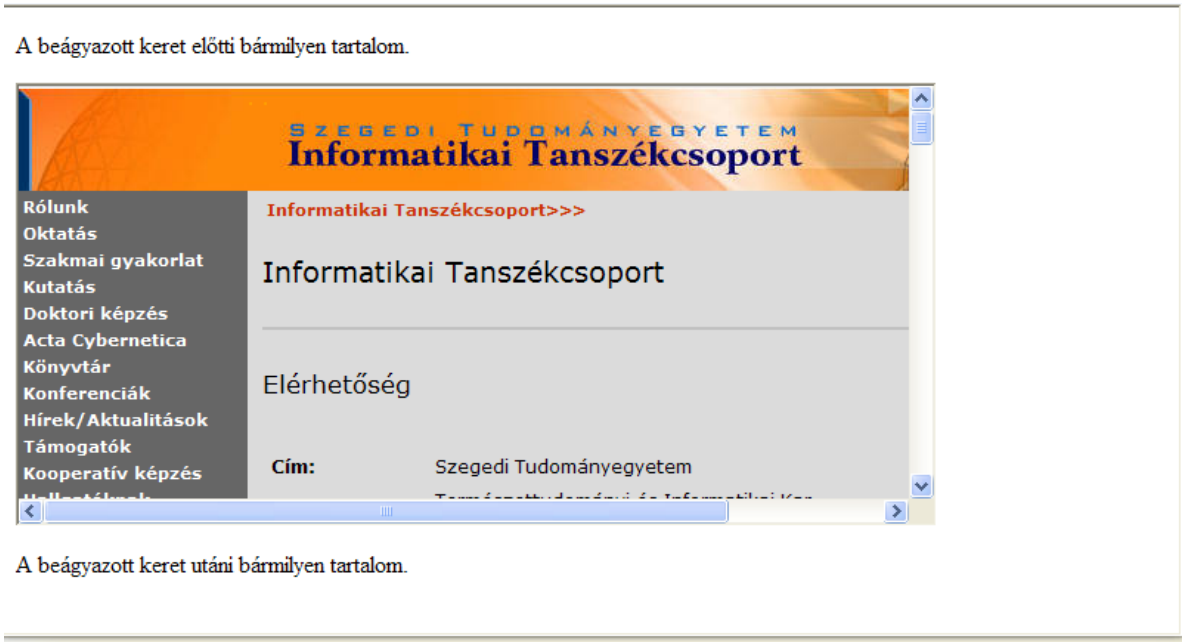
2.11.2. Belső keretek

A belső keret (lebegőkeret, inline frame, [55], [57]) használata esetén a böngészőablak területét nem osztjuk fel több részre, hanem csak egy téglalap alakú részt különítünk el abból valamilyen beágyazott tartalom (például hirdetések) megjelenítésére. Belső keretek használatakor átmeneti dokumentumtípust kell deklarálnunk. Belső keretet `iframe` objektumként tudunk megvalósítani, melynek `src` jellemzőjének kell megadnunk a beágyazandó tartalom URL-jét. Az `<iframe>` és `</iframe>` tagok között olyan tartalmat helyezhetünk el, melyet belső

kereteket nem ismerő böngészők esetén szeretnénk megjeleníteni. Az `iframe` objektum `width` és `height` jellemzőivel a belső keret méreteit adhatjuk meg. Lássunk egy példát belső keret beillesztésére:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
 <head>
  <meta http-equiv="Content-Type"
        content="text/html;charset=ISO-8859-2" />
  <title>Beágyazott keret</title>
 </head>
 <body>
  <p> A beágyazott keret előtti bármilyen tartalom. </p>
  <iframe src="http://www.inf.u-szeged.hu/"
          width="80%" height="300"
   title="Az Informatikai Tanszékcsoport beágyazott honlapja">
   <p> Ez a tartalom jelenik meg akkor, ha a böngésző nem
       támogatja a belső kereteket. </p>
  </iframe>
  <p> A beágyazott keret utáni bármilyen tartalom. </p>
 </body>
</html>
```

A megjelenő oldal a 5. ábrán látható.



5. ábra: Belső keret

Miután áttekintettük az XHTML alapjait, a következő fejezetben a weboldalak ügyfél oldali programozásával fogunk megismerkedni.

3. Ügyféloldali webprogramozás

3.1. Alapfogalmak

Ha interaktív weboldalakat akarunk készíteni, akkor azokhoz programokat kell csatolnunk, melyeket elhelyezhetünk a HTML kódba beágyazva, vagy külön fájlban, sőt a két lehetőséget együtt is használhatjuk. A weboldalakhoz csatolt programokat számos programozási nyelven írhatjuk. Legelterjedtebbek a PHP, Java (Java applet, Java Servlet, JSP), JavaScript, Visual Basic Script. A webprogramozás lehet ügyfél oldali vagy kiszolgáló oldali.

Ügyfél oldali webprogramozás esetén a programok a felhasználó számítógépén futnak, így nem a szervert terhelik, azonban korlátozott adattárolást tesznek lehetővé. Biztonsági okokból az ügyfél oldali programozási nyelvek korlátozottan férhetnek hozzá a felhasználó erőforrásaihoz, továbbá hosszabb ideig megtartandó vagy fontos adatokat azért sem célszerű a felhasználó gépén tárolni, mert a felhasználó korlátozhatná az adatokhoz történő hozzáférést (például kikapcsolhatja a gépét), illetve módosíthatná vagy törölhetné a tárolt adatokat. Az ügyfél oldali programozás azonban jól használható olyan feladatok megoldására, mint az oldal megjelenésének és tartalmának dinamikus változtatása, látványelemek megvalósítása vagy űrlapok mezőinek elküldés előtti ellenőrzése.

Kiszolgáló oldali webprogramozás esetén a programokat a kiszolgáló gép futtatja és olyan feladatok is megoldhatókká válnak, mint a teljes körű adattárolás, adatbázisok használata, vagy a felhasználók jogosultságainak kezelése.

Ideális esetben az ügyfél és a kiszolgáló oldali programozást együtt alkalmazzuk.

Az ügyfél oldali szkripteket a böngészők értelmezik és futtatják. A böngészőkben többnyire be lehet állítani szkriptekben való hibakeresést. Ehhez például Internet Explorerben az Internet-beállítások/Speciális fülben meg kell szüntetni a parancsfájlokban való hibakeresés letiltását (ki kell venni előle a pipát), Firefoxban a hibákat az Eszközök/Hibakonzol, Operaban pedig az Eszközök/Haladó/Hibakonzol menüponttal lehet megnézni.

3.2. A JavaScript nyelvi elemei

A JavaScript részletes leírását az olvasó megtalálja a [7], [8] és [10] irodalmakban, melyek jelen fejezet forrásaiként is szolgálnak.

A JavaScript egy önálló nyelv, mely a Java nyelvtől függetlenül lett kifejlesztve és nem tekinthető a Java nyelv változatának, mivel a két nyelv között számos különbség van.
Jelen jegyzet terjedelmi korlátai miatt azonban, abból kiindulva, hogy az olvasó már ismeri a Java nyelvet, eltekintünk a JavaScript azon alapvető nyelvi elemeinek bemutatásától, melyek a Java nyelvhez hasonlóak, és első sorban a JavaScript sajátosságait, illetve a különbségeket fogjuk bemutatni.

3.2.1. JavaScript parancsok elhelyezése és futtatása

A JavaScript egy értelmezett parancsnyelv, melynek utasításait a böngésző egyenként értelmezi és hajtja végre. Tetszőleges JavaScript parancsokat beilleszthetünk a weboldal fejrészébe vagy törzsébe a következő módon:

```
<script language = "JavaScript"
        type = "text/javascript">
  utasítások
</script>
```

Elhelyezhetjük a parancsokat egy `.js` kiterjesztésű külön fájlban is, melyet a következőképpen csatolunk:

```
<script language = "JavaScript"
        type = "text/javascript"
               src="progi.js">
</script>.
```

A weboldal objektumaihoz társíthatunk tevékenységeket, melyek az objektummal történő adott események esetén kell végrehajtódjanak. Ilyen események lehet például az objektumra történő kattintás, valamely billentyűnek a lenyomása, egy adott űrlapmező elhagyása stb. Az ilyen tevékenységeket, illetve több tevékenység esetén az ezeket csoportosító függvényeket *eseménykezelőknek* nevezzük. Eseménykezelőket elhelyezhetünk a fenti módon beágyazott vagy csatolt szkript kódban, vagy az adott objektum nyitótagjában. Például, ha azt akarjuk előírni, hogy egy nyomógombra való kattintáskor jelenjen meg egy üzenetablak ”Ügyes vagy!” felirattal, akkor a következő kódot írhatjuk:

```
<input type = "button"
       onClick= "alert('Ügyes vagy!')" />.
```

A böngésző a weboldalt fentről lefele olvassa be, és a
beágyazott, vagy csatolt szkript utasításokat a megjelenésük helyén elvégzi. Deklarációk, definíciók (például függvények, eseménykezelők) esetén az „elvégzés” ezek regisztrálását jelenti, az ezekhez tartozó utasítások tényleges futtatása a függvény meghívásakor, illetve az adott esemény adott objektumon történő kiváltásakor fog megtörténni.

3.2.2. Változók és konstansok

A JavaScript gyengén típusos nyelv és a változók típusát dinamikusan kezeli. Deklaráláskor nem adjuk meg a változó típusát, továbbá a változó típusa futás közben a hozzárendelt értékétől függően megváltozhat (például lehet egyszer egész, máskor string). A változók deklarálása a `var` kulcsszó használatával lehetséges, ennek hiányában első megjelenési helyükön automatikusan deklarálódnak, viszont a lokális változókat kötelező deklarálni. Minden olyan változó vagy tulajdonság, amelynek nem adtunk értéket `undefined` értékkel rendelkezik. Hasonlóképpen `undefined` értéket ad vissza egy függvény, ha nem rendelkezik visszatérési értékkel vagy az `undefined`. A `NaN` egy speciális érték, amire akkor értékelődik ki egy változó, ha előzőleg a `Number.NaN` értéket adjuk neki, vagy adott műveletben annak értéke nem szabályos szám (`undefined`-et is beleértve). A `NaN` érték tesztelése kizárólag az `isNAN()` függvénnyel lehetséges, egyenlőség operátorokkal nem. A `null` egy objektum, melyet egy változó csak programozó általi hozzárendeléssel kaphat értékül és azt jelzi, hogy a változó nem tartalmaz tényleges értéket. Az `undefined` és `null` is logikailag `false`-ra értékelődnek ki, numerikus kiértékelésnél azonban a `null` értéke 0.

A karaktersorozatok lehetnek `'aposztrófok'` vagy `"idézőjelek"` között, melyben idézőjelet `\"`-el írunk. A kétféle jelölés lehetővé teszi, hogy egy karaktersorozatban egy másik karaktersorozatot helyezzünk el.

Minden objektum, ami nem `undefined`, `null`, `NaN` vagy `0`, logikailag `true`-nak értékelődik ki, ezért üres karaktersorozat vagy `false` értékű `Boolean` objektum is `true`-nak fog kiértékelődni.

3.2.3. Operátorok

A JavaScript 2.0 óta létezik operáció kiterjesztés, ezért az egyes operátorok sok mindenre alkalmasak lehetnek, ebben az alfejezetben azonban a beépített funkciójukat fogjuk bemutatni.

JavaScript-ben a == csak az értékek egyenlőségét vizsgálja (esetleges típuskonverzióval). A típusok és értékek egyelőségét a === operátorral lehet vizsgálni, melynek negációja a `!==` operátor. Például `3 == "3"` teljesül, míg `3 === "3"` nem, ezért `3 !== "3"` igen. A Java-tól eltérően a `/` mindig lebegőpontos osztást jelent, tehát `1/2` = 0.5 lesz. A + karaktersorozatok összeillesztését is végzi.

A *tulajdonsag* `in` *objektum* igazat ad vissza, ha az objektumnak van olyan tulajdonsága (nem értéke!). Megjegyzendő, hogy tömb objektum esetén az index is tulajdonság.

Az *objektum* `instanceof` *tipus* akkor igaz, ha az objektum a meghatározott objektumtípusba (pl. tömb, dátum stb.) tartozik.

A `typeof` *operandus* vagy `typeof (`*operandus*`)` a nem kiértékelt operandus típusát jelző stringet adja vissza. Ezt a következőképpen használhatjuk annak lekérdezésére, hogy az objektumnak van-e bizonyos tulajdonsága:

```
if (typeof obj.tulajdonsag == "undefined"){
// az objektumnak nincs ilyen tulajdonsága
}.
```

Azért nem elegendő `if (obj.tulajdonsag)`-ot írni, mert ez akkor is hamisat adna, ha a tulajdonság létezne, de értéke `false`, `0`, vagy `null` lenne.

A `void kifejezes` vagy `void (kifejezes)` operátor használatakor a kifejezés kiértékelődik, de értéke nem adódik vissza. Például kiíratásnál ezzel meg lehet akadályozni, hogy egy művelet (például függvényhívás) eredménye kiíródjon.

3.2.4. Vezérlési szerkezetek

JavaScript-ben használhatjuk ugyanazokat az `if-else`, `switch`, `for`, `while`, `do-while`, `break`, `continue`, `label` vezérlési szerkezeteket, mint Java-ban, továbbá a `for (`*változó* `in` *objektum*`){` *utasítások* `}` vezérlési szerkezet esetén a *változó* az *objektum* összes tulajdonságának nevét rendre felveszi és minden esetben végrehajtódnak az utasítások.

Függvény bárhol (akár pl. egy `if`-ben vagy kifejezésen belül is) definiálható a következőképpen:

`function nev(`*paraméterLista*`) { ... }`.

Érték szerinti paraméterátadás van, de objektumok esetében a referencia (cím) másolódik. Egy függvény egyben `Function` típusú objektum is, így képes metódusokat és tulajdonságokat tárolni. Például:

```
function fuggv(){};
fuggv.adattag = "X";
alert(fuggv.adattag); // X
```

Függvény `Function` objektumként is létrehozható. A függvény argumentumait az `arguments` tömb is tárolja, az első paraméter az `argumetns[0]`, az argumentumok száma pedig `arguments.length`. Egy függvénynek a formális paraméterlistától eltérő számú paramétert is átadhatunk, melyeket a függvényben az `arguments` tömb segítségével kérdezhetünk le (például ha nem tudjuk előre, hogy hány paraméterrel fogjuk meghívni). A függvények adatként is használhatók, hozzárendelhetők változókhoz, objektumok tulajdonságaihoz, illetve tömbök elemeihez. Például:

```
function kob(n) { return n*n*n }
```

```
var a = kob;
var b = a(2); // b = 8 lesz
var c = new Array(2);
c[0] =kob;
c[1] = c[0](3); // c[1] = 27 lesz
```

Hogyha a függvény egy objektumhoz kapcsolódik, akkor *metódusnak* nevezzük.

3.2.5. Tömbök

JavaScript-ben a tömb változónevekkel társított értékek halmaza, melyet *asszociatív tömbnek* nevezünk. A tömb egy objektum, melynek indexei tulajdonságok. Fordítva, adott objektum tulajdonságai is hivatkozhatók `objektum["tulajdonság"]` formában.

3.2.6. Objektumok

A Java nyelv típusosságával szemben a JavaScript egy gyengén típusos nyelv, mely képes a dinamikus kötések nagyon magas fokára. Osztályokat nem tudunk létrehozni, azonban objektumokat igen, melyek között az öröklés is lehetséges ([10]).

Egy objektum létrehozására az egyik lehetőség az, hogy konkrétan megadjuk az objektum tulajdonságait és azok értékeit a következőképpen:

`objNev = {`*tul1*`:`*ert1*`, `*tul2*`:`*ert2*`, ...}`.

Objektumokat létrehozhatunk továbbá konstruktor függvények segítségével is, viszont JavaScript-ben nincsenek osztályok, ezért a létrehozandó objektumok tulajdonságait és metódusait a konstruktor függvényben kell megadnunk oly módon, hogy a létrehozandó objektumra `this`-el hivatkozunk. Ily módon definiáljuk a létrehozandó objektum *prototípusát*. Új objektum létrehozásához a konstruktort `new` operátorral kell meghívni. Például:

```
function nev(csaladi, kereszt){
   this.csn = csaladi;
   this.kn = kereszt;
}
function kiir(){
  var s = "";
  s += this.neve.csn + " ";
  s += this.neve.kn + " ";
  s += this.kora + "éves!"
  alert(s);
}
function ember(neve, kora){
  this.neve = neve;  this.kora = kora;  this.kiir = kiir;
}
nev1 = new nev("Kiss", "Árpád");
nev2 = new nev("Nagy", "Aladár");
e1 = new ember(nev1, 38);
e1.kiir();
```

A létrehozott objektumok további egyedi tulajdonságokkal és metódusokkal is bővíthetők. Például:

```
function ugral(s){
   document.write(s+" ugrálok!");
}
e1.jokedvu = "Igen";
e1.orvend = ugral; // hozzárendelés
```

```
e1.orvend("Vígan"); // metódushívás
```

A konstruktor függvény `prototype` tulajdonsága referencia egy úgynevezett *prototípus objektumra*, mely kezdetben üres objektum és bővíthető tulajdonság-érték párokkal. Például:

```
ember.prototype.magassag="150";
```

A hozzáadott tulajdonságok értékei is ugyanígy módosíthatók. Alternatív megoldásként a `prototype` referenciát egy már létező, de nem beépített objektumra is átállíthatjuk.

Amikor egy objektumban még nem létező tulajdonságnak értéket adunk, akkor az saját tulajdonságként létrejön az objektumban. Amikor az objektum adott tulajdonságát olvassuk, akkor először megnézi, hogy van-e az objektumnak olyan saját tulajdonsága, és ha nincs, akkor megpróbálja a konstruktor függvény prototípus objektumának azonos nevű tulajdonságát kiolvasni. Ily módon az objektum látszólag rendelkezni fog a prototípus objektumban megadott tulajdonságokkal is, azokat is beleértve, melyek az objektum létrehozása után lettek hozzáadva a prototípus objektumhoz. Ezt egyféle öröklésnek is tekinthetjük, melyben az ős a prototípus objektum. Az ős objektum metódusai hivatkozhatnak a gyermek objektumok tulajdonságaira is.

A prototípus objektum segítségével bővíthetjük a beépített JavaScript objektumokat is. Például, az `Array` objektum prototípus objektumához hozzáadhatunk egy `Osszeg` függvényt, mely kiszámolja az adott tömbben tárolt értékek összegét:

```
Array.prototype.Osszeg = function (){
  var ossz = 0; var i;
  for(i=0; i < this.length; i++) ossz += this[i];
  return ossz;
}
```

Ezután az `Osszeg` függvényt bármilyen numerikus értékeket tartalmazó tömb objektumra meghívhatjuk:

```
var tomb = new Array(1,2,3);
var o = tomb.Osszeg();
```

Az öröklés megvalósítására egy másik lehetőség az *Object Masquerading* technika alkalmazása, melynek az a lényege, hogy az ősnek szánt objektum konstruktor függvényét a gyermek objektum metódusaként deklaráljuk, majd meghívjuk. Ebben az esetben az ős konstruktorában levő `this`-ek a gyermek objektumra fognak vonatkozni. Ily módon többszörös öröklést is megvalósíthatunk, melyben ugyanazon adattag későbbi módosítása (például a később meghívott konstruktor által) felülírja az előző értéket.

3.2.7. Szabályos kifejezések

Számos más programozási nyelvhez hasonlóan JavaScriptben is használhatók szabályos kifejezések, melyek használatával kapcsolatosan jelen fejezetben a [10] alapján csak a legfontosabb tudnivalókat tárgyaljuk, de az olvasó a [10] és [22] irodalmakban további ismereteket és felhasználási lehetőségeket is talál.

Tipikusan űrlapoknál szoktuk használni annak ellenőrzésére, hogy a felhasználó által beírt egyes adatok (például telefonszám, e-mail cím stb.) megfelelnek-e a kívánt formai szabályoknak.

JavaScriptben szabályos kifejezéseket kétféleképpen definiálhatunk: / jelek között, illetve `RegExp` típusú objektumként. Utóbbi esetben a konstruktor meghívásakor annak paramétereként kell megadnunk a szabályos kifejezés felépítését, melyben a \ jeleket duplázni kell.

Lássunk néhány karaktert, melyeket szabályos kifejezésekben speciális jelentéssel használhatunk:

- \ közönséges karakterből speciálisat, speciális karakterből közönségest csinál
- ^ szöveg elejére való illeszkedés

- `$` szöveg végére való illeszkedés
- `*` az előző karakter 0 vagy több előfordulása
- `+` az előző karakter 1 vagy több előfordulása
- `.` újsor karakteren kívül bármilyen karakter
- *xx*`|`*yy* `xx`-re vagy `yy`-ra illeszkedik
- `?` az előző karakter 0 vagy 1 előfordulása
- `(`*xyz*`)` csoportosítja (egyetlen karakterként kezeli) *xyz*-t, illeszkedik *xyz*-re és megjegyzi az illeszkedést;
- `{`*n*`}` az előző karakter pontosan *n*-szeri előfordulására illeszkedik
- `{`*n*`,`*m*`}` az előző karakter legalább *n*, de legfeljebb *m* előfordulására illeszkedik
- `[`*xyz*`]` a felsorolt karakterek bármelyikére illeszkedik; `-` el tartomány is megadható (pl `[a-c]`)
- `[^`*xyz*`]`: bármilyen karakterre illeszkedik, ami nincs felsorolva; itt is használhatunk tartomány megadást (pl. `[^a-c]`)
- `\d` egy decimális számjegyre illeszkedik (ugyanaz, mint `[0-9]`)
- `\`*szám* hivatkozás a `szám`-adik kezdő `(`-el meghatározott, megjegyzett illeszkedésre.

Az `m`, `i`, `g` paraméterként megadható karakterekkel előírhatjuk, hogy a keresés több sorra is kiterjedjen, ne tegyen különbséget kis- és nagybetűk között, illetve az összes találatot adja vissza. A paraméter karaktereket a második `/` jel után, vagy a `RegExp` második paramétereként adjuk meg.

Szabályos kifejezések illesztésekor alapértelmezés szerint a lehető legtöbb karakter fog illeszkedni úgy, hogy a kifejezés hátralevő része is illeszkedhessen. A `?` használatával ez letiltható, de akkor is mindig megkeresi a lehető legelső illeszkedési pontot és onnan kezdve próbál a lehető legkevesebb karakterrel illeszteni. Például:
`/x*?y/` illesztése az `"xxxy"`-re nem a `"y"` lesz, hanem `"xxxy"`.

Ily módon például egy HTML tag a `/<(.*)>.*<\/\1>/`, míg egy IP cím a `/(\d{1,3}\.){3}\d{1,3}/` szabályos kifejezésre illeszkedik. Utolsó példaként egy karaktersorozatban az összes + karaktert a következőképpen tudjuk szóközre cserélni:
`szk = /\+/g; s = s.replace(szk," ");`.

3.2.8. Előre definiált objektumok

JavaScriptben a tömb egy `Array` objektum, ily módon egy dimenziós tömb létrehozása a következőképpen lehetséges:

```
a = new Array(e0, e1, ..., eN); vagy
a = new Array(elemszam);.
```

A tömb hosszát annak `length` tulajdonságával kérdezhetjük le. Tömbök kezelésére számos beépített metódus van. A többdimenziós tömbök nincsenek beépítve, de a tömb elemei lehetnek objektumok, azaz tömbök is, így a többdimenziós tömbök is létrehozhatók. Például:

```
var i, ah, bh;
var a = new Array(4);
for (i=0, ah = a.length; i < ah; ++i){
   a[i] = new Array(3);
   for (j=0, bh = a[i].length; j < bh; ++j){
       a[i][j]="Az"+i+". tömb "+j+" . eleme";
   }
}.
```

A dátum és idő kezelésére a `Date` objektum használható, melynek létrehozásakor paraméterként megadhatjuk az évet, hónapot, napot, órát, percet, másodpercet, melyben a hónapok számozása 0-tól kezdődik (az lesz január). Például:

```
dobj =new Date(2011, 2, 24, 13,38,33);.
```

Paraméter hiányában az objektum tartalma az aktuális dátum és időpont lesz. A `getTime()`, illetve `setTime()` metódusok lekérdezik, illetve beállítják a dátum objektum tartalmát 1970. január 1. 00:00:00-től eltelt milliszekundumban. Az év hónap, nap, óra, perc, másodperc lekérdezésére és beállítására a `getFullYear()`, `getMonth()`, `getDate()`, `getHours()`, `getMinutes()`, `getSeconds()` metódusok, illetve ezek `set`-el kezdődő párjai használhatók. Számos további metódus is rendelkezésre áll a dátumok kényelmesebb kezelésére.

A `Math` objektum a matematikai konstansok és függvények használatát teszi lehetővé. Ez esetben azonban nem saját objektumot hozunk létre, hanem a `Math` saját tulajdonságait (például `Math.PI`) és metódusait használjuk. Például `Math.floor()` az argumentumát lefele kerekíti, a `Math.random()` pedig egy [0,1) intervallumbeli véletlen számot ad vissza.

A `String` objektum a karakterlánc becsomagoló objektum típusa. A JavaScript a karakterláncot automatikusan String objektummá konvertálja, ezért a `String` objektum metódusai karakterláncokra is meghívhatók. A számos metódus közül itt most csak néhányat említünk meg.

- `indexOf()`: megadja a paramétereként átadott string első előfordulási helyét egy karakterláncon belül;
- `toLowerCase()`, `toUpperCase()`: a paraméter karakterlánc minden betűjét kisbetűsre, illetve nagybetűsre alakítja;
- `split()`: a karakterláncot a paraméterként átadott jel szerint részekre bontja és az egyes részeket elhelyezi egy tömbben;
- `replace()`: ráilleszti az első paraméterként átadott reguláris kifejezést a karakterláncra, és ha egyezést talál, akkor lecseréli az előfordulásokat a másodikként átadott paraméterre.

A `RegExp` reguláris kifejezések becsomagoló osztálya. Egy ilyen objektumra meghívott `exec()`, `test()` metódusok keresést hajtanak végre a paraméterként átadott karakterláncon.

3.3. A W3C dokumentumobjektum-modell (DOM) objektumai és eseményei

A DOM ([7], [10], [58]) a weboldalt alkotó objektumok, tulajdonságok és metódusok rendszere, melyben a weblap minden eleme objektumnak van tekintve. Ennek a rendszernek részei azok az események is, melyekben az egyes objektumok részt vehetnek. A DOM tulajdonképpen egy alkalmazásprogramozási felület (API), amelyet a böngészőbe építenek, így a szkriptnyelvek (például VBScript, JavaScript) mind használhatják. Jelen jegyzet keretei között a HTML DOM legfontosabb elemeit fogjuk bemutatni, kiegészítve azt a JavaScript fontosabb objektumaival.

3.3.1. Alapvető események

Ebben a fejezetben áttekintjük a legfontosabb eseményeket és azt, hogy azok mikor váltódnak ki:

- `select`: a felhasználó kijelölt egy szövegrészt,

- `focus`: egy objektum megkapta a fókuszt,
- `blur`: egy objektum elvesztette a fókuszt,
- `change`: egy űrlap bizonyos eleme megváltozott (más tartalma lett) és bizonyos esetekben elvesztette a fókuszt
- `reset`: a felhasználó törölte az űrlap tartalmát (a Reset gombra kattintott),
- `submit`: a felhasználó a Submit gombra kattintott,
- `mouseover`, `mouseout`: az egér egy objektum vagy link fölé került vagy elhagyta azt,
- `mousemove`: a felhasználó mozgatta az egeret,
- `mousedown`, `mouseup`: a felhasználó lenyomta vagy felengedte az egér gombját,
- `click`: a felhasználó kattintott az egér bal gombjával, fókuszban levő parancsgomb esetén Entert vagy szóközt ütött, vagy lenyomott egy objektumhoz kapcsolt billentyűparancsot,
- `dblclick`: a felhasználó duplán kattintott,
- `dragdrop`: a felhasználó letett egy objektumot (pl. egy fájlt belehúzott) a böngésző ablakába,
- `keydown`, `keypress`, `keyup`: a felhasználó lenyomott, lenyomva tartott, vagy felengedett egy billentyűt,
- `resize`: az ablak vagy keret átméreteződött,
- `move`: az ablak vagy keret elmozdult (felhasználó vagy szkript következtében),
- `abort`: a felhasználó megszakította egy kép betöltését,
- `error`: a dokumentum vagy kép letöltése közben hiba lépett fel,
- `load`: a böngésző befejezte a dokumentum betöltését egy ablakba vagy keretbe,
- `unload`: a felhasználó kilépett a dokumentumból.

Az eseménykezeléshez meg kell adnunk, hogy mely objektumon, milyen esemény esetén, milyen utasítások hajtódjanak végre, vagy melyik függvény hívódjon meg.

Ezt megadhatjuk az objektum nyitó tagjában `on` előtaggal, ezt nevezzük *inline* (sorba illesztett) módszernek. Például:

```
<input type = "button"
       onClick= "esemenykezelo()" />
```

Az eseménykezelést definiálhatjuk a szkriptben is, ebben az esetben a HTML kód tisztább marad, és az eseménykezelők is áttekinthetőbben lesznek felsorolva. Például, ha a szkriptben van egy gomb objektumreferenciánk a fenti gombra, akkor az eseménykezelőt így rendeljük hozzá:

```
gomb.onClick = kezel;
```

Arra, hogy a HTML oldalon levő objektumot a szkriptben hogyan tudjuk beazonosítani, később fogunk visszatérni.

Az esemény hozzárendeléseket a böngésző megjegyzi és az esemény bekövetkezésekor meghívja.

Az eseménykezelőben arra az objektumra, amelyen az esemény bekövetkezett szabványosan `this`-el, Internet Explorerben pedig `window.event.srcElement`-el tudunk hivatkozni.

Az objektumok egy részénél bizonyos események bekövetkezése alapértelmezett tevékenységeket is kivált a böngészőben. Ha saját magunk szeretnénk kezelni az eseményt, akkor az alapértelmezett eseménykezelést célszerű letiltani. Ezt megtehetjük úgy, hogy az esemény-

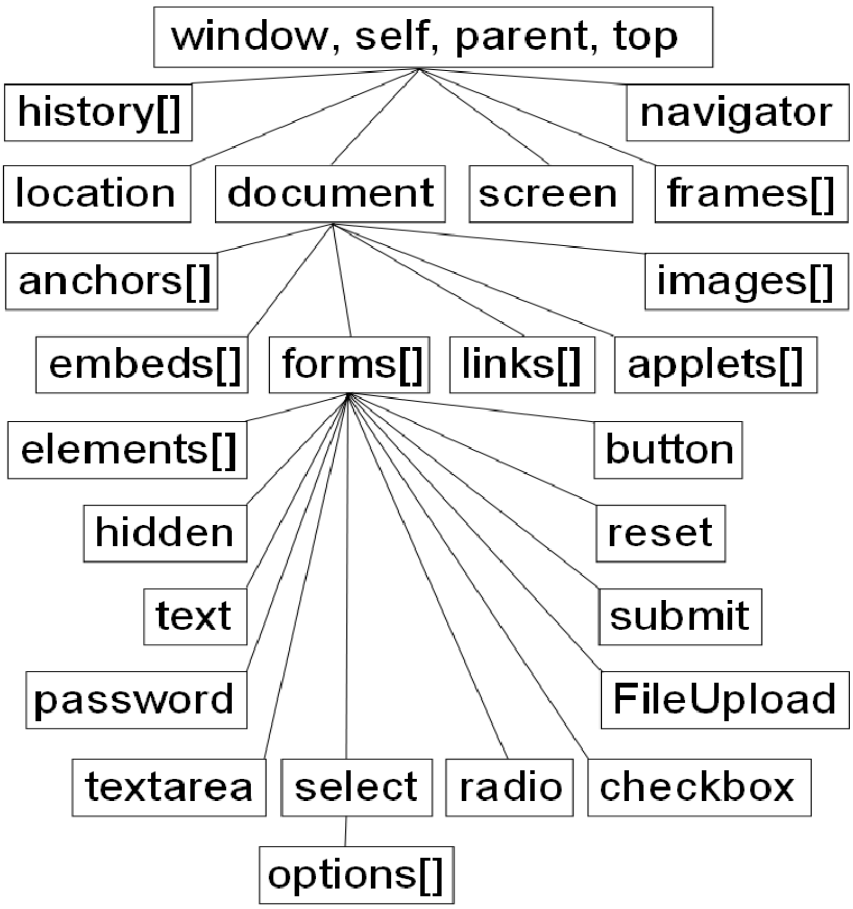
kezelőnek visszatérési értékként `false`-ot adunk meg, Internet Explorerben pedig a `window.event.returnValue = False;` utasítást adjuk.

A szabványos DOM tárgyalásánál az eseménykezelésre és az alapértelmezett események letiltására még további lehetőségeket is fogunk látni.

3.3.2. A böngésző objektumai

Ebben a fejezetben áttekintjük az ügyfél oldali webprogramozásban használható legfontosabb objektumokat. Az egyes objektumok után feltűntetett `[]` azt jelenti, hogy az objektum adott típusú elemek tömbje.

Az objektumokat hierarchikus faszerkezetbe rendezhetjük annak megfelelően, hogy egy adott objektumba milyen más típusú objektumokat ágyazhatunk be. Ezt a faszerkezetet mutatja be az 6. ábra.



6. ábra: DOM objektumok fája

Az ablak hivatkozása környezettől függően lehet `window`, `self`, `parent` vagy `top`. Az objektumok tulajdonságainak és metódusainak hivatkozásánál a gyökértől induló teljes útvonalat meg kell adni. Kivétel a `window`, mely egyetlen ablak használata esetén elhagyható.

Lássuk az objektumok jelentését, legfontosabb tulajdonságait és metódusait.

A `self` az aktuális ablakot jelenti, mely tartalmazza a JavaScript dokumentumot. A `parent` és `top` keretrendszer esetén az aktuális keretet, illetve a keretrendszer fődokumentumát tartalmazó ablakot jelölik. Az ablak `status` tulajdonsága tartalmazza a státuszsorban megjelenítendő karaktersorozatot. Az ablak `opener` tulajdonsága referencia arra a `window`

objektumra, amelyben levő szkript `open`-el létrehozta az ablakot; az új ablak ezzel hivatkozhat a létrehozójában levő változókra és függvényekre. Azt mondjuk, hogy egy dialógusablak *modális*, ha a felhasználó csak az ablak bezárása után tud visszatérni az előző ablakba. Az ablakokra vonatkozó legfontosabb metódusok:

- `alert()`: megjelenít egy (többnyire modális) dialógusablakot, melyen kiírja a paramétereként megadott (nem HTML) karaktersorozatot,
- `confirm()`: megjelenít egy modális OK és egy Mégse gombokat is tartalmazó dialógusablakot, benne a paraméter (nem HTML) karaktersorozattal; visszatérési értéke OK-ra való kattintás esetén `true`, Mégse-re való kattintás esetén `false`,
- `prompt()`: megjelenít egy OK, Mégse gombokat és egy beviteli szövegmezőt tartalmazó dialógusablakot, benne a paraméter (nem HTML) karaktersorozattal; visszatérési értéke OK esetén a szövegmező tartalma, Mégse esetén null,
- `setInterval(`*fgv*`, `*intervallum*`, `*par*`)`: az *intervallum*-ban megadott milliszekundumokként periodikusan végrehajtja az *fgv* függvényt vagy programkódot, melynek függvény esetén átadja a *par* paramétereket; egy referenciát ad vissza, melyet a végrehajtás leállítására szolgáló `clearInterval()` metódusnak lehet paraméterként átadni,
- `setTimeout(`*fgv*`, `*intervallum*`, `*par*`)`: az *intervallum*-ban megadott milliszekundumos késleltetéssel végrehajtja az *fgv* függvényt vagy programkódot, melynek függvény esetén átadja a *par* paramétereket; egy referenciát ad vissza, melyet a végrehajtás leállítására szolgáló `clearTimeout()` metódusnak lehet paraméterként átadni,
- `focus()`, `blur()`: ráviszi, illetve elveszi a fókuszt,
- `resizeTo()`: a megadott méretre átméretezi az ablakot,
- `moveTo()`: a megadott abszolút pozícióba helyezi az ablakot,
- `close()`: bezárja a megadott ablakot,
- `open(`*tartalom*`, `*név*`, `*hogyan*`)`: létrehoz egy új ablakot, melyben a *tartalom*-ban specifikált HTML dokumentumot nyitja meg; az ablak neve *név* lesz, ezzel lehet az ablakot a `target`-ben hivatkozni; *hogyan*-al a megnyitandó ablak tulajdonságait adhatjuk meg (méretek, pozíció, menüsor, státuszsor, eszközsor, gördítősáv megjelenítése stb.); visszaad egy objektumreferenciát, melynek segítségével az eredeti ablakból hivatkozható lesz.

Például:

```
var param="width=650,height=260";
param += ",location=no,toolbar=no";
param +=",menubar=no,titlebar=no, status=no";
ujAblak = window.open("ablak_uj.html", "ablak", param);
```

A létrehozott ablakra, betöltődése után, a következőképpen hivatkozunk:

```
var sz = ujAblak.document.getElementById("szoveg");
sz.innerHTML = "Az eredeti ablak üdvözöl!";
```

Az új ablakban a létrehozójára pedig a következőképpen hivatkozunk:

```
var mse = opener.document.getElementById("mese")
mse.innerHTML = "Új ablak jelenti, hogy megnyilt!";
```

A `navigator` objektum a böngésző tulajdonságait tartalmazza. A `userAgent` tulajdonsága egy karakterlánc, mely a böngésző által a kiszolgálónak küldött jellemzőket tartalmazza.

A `screen` objektum a valamennyi ablak számára közös képernyő legfontosabb tulajdonságait tartalmazza. Például a `height` és `width` a képernyő magassága, illetve szélessége pixelekben (azaz a felbontás).

© Holló Csaba, SzTE

A `history` az adott ablakban a felhasználó által meglátogatott URL-ekre vonatkozó információkat tartalmazó csak olvasható tömb. Biztonsági okokból a szkript nem kérdezheti le a meglátogatott oldalak URL-jét, de átirányíthatja az oldalt azokra a `back()`, `forward()`, illetve `go()` metódusok segítségével. Az első kettő a böngésző Vissza, Előre gombjainak szerepét tölti be, míg `go()` a paraméterében megadott számnak megfelelően előre vagy hátra lép a `history` tömbben és az ott található oldalt tölti be.

A `location` a böngészőablakban levő aktuális oldal URL-jével kapcsolatos adatokat tartalmazza. A teljes URL-t a `href` tulajdonsága tartalmazza, értékadás esetén betölti az adott oldalt, melyet a `history` tömbhöz hozzáad. Az URL egyes részei külön is lekérdezhetők. Például a `search` az URL lekérdezési részét adja meg. A `replace()` metódus a jelenlegi oldalt egy másikkal helyettesíti, de a `href` értékadástól eltérően, azzal felülírja a `history` tömb aktuális elemét, tehát a felülírt oldal a Vissza gombbal nem lesz elérhető.

A `document` objektum egy weboldalt jelképez és a dokumentum globális tulajdonságait tárolja. Például a `title` az oldal címe, a `cookie` pedig a dokumentumhoz tartozó süti. A `document` objektum `write()`, `writeln()` metódusai a HTML kódba írják be a paraméter értékét vagy szövegét, illetve egy újsor karaktert, de a weblap összeállítása után történő meghívásuk törli (felülírja) a forráskódot. Az `open()`, `close()` metódusok megnyitnak, illetve lezárnak egy folyamot, amelybe a `write`, `writeln` metódusokkal írhatunk.

Az `images[]` tömb az oldalon levő képeket tartalmazza. Képeket létrehozhatunk `Image` objektumként is `var kep = new Image(`*szélesség, magasság*`);` utasítással. A képobjektumhoz a képfájlt az `src` tulajdonsággal tudjuk hozzárendelni. A kép akkor fog megjelenni, amikor azt hozzáadjuk az `images` tömbhöz (például `document.images[3].src = kep.src;`) vagy az oldal DOM fájához (erre később visszatérünk).

A `forms[]` az űrlapok tömbje. Az `action`, `method`, `target`, `encoding` tulajdonságok a `form` tag `action`, `method`, `target`, `enctype` jellemzőinek felelnek meg. Minden űrlapnak van egy `elements[]` tulajdonsága, mely az űrlap elemeire mutató referenciákat tartalmazó tömb. Az űrlap alapállapotba állítását, illetve elküldését szkriptben a `reset()`, illetve `submit()` metódusokkal tudjuk előírni.

Űrlapelemek `name` és `type` jellemzőinek értékét az objektumok azonos nevű tulajdonságai tartalmazzák, a `form` pedig a befoglaló űrlapot adja meg. `Select` esetén a `type` az egyszeres vagy többszörös kiválaszthatóságot adja meg, a `selectedIndex` pedig megadja vagy beállítja az első kiválasztott opció indexét. Az `option` objektum `index` tulajdonsága az opció indexét adja meg az `options[]` tömbben, a `text`, illetve `value` tulajdonságai az opcióhoz rendelt szöveget, illetve az opció kiválasztása esetén az elküldendő értéket tartalmazzák. Szöveges űrlapmezők `defaultValue`, illetve `value` tulajdonságai tartalmazzák a mező eredeti, illetve aktuális értékét. Az objektumok eredeti, illetve aktuális kiválasztottsága `Option` objektum esetén a `defaultSelected`, `selected` tulajdonságokban, `radio` és `checkbox` objektumok esetén a `defaultChecked`, `checked` tulajdonságokban vannak eltárolva. Az aktuális értékek, illetve kiválasztottság módosíthatók. Minden űrlapmező esetén a `focus()`, illetve `blur()` ráviszi, illetve elveszi a fókuszt. Szöveges mezők esetén a `select()` kijelöli annak tartalmát (például annak érdekében, hogy a felhasználó egyetlen gombnyomással törölhesse). Gombok esetén a `click()` egy kattintást szimulál.

3.3.3. Sütik használata

A *süti* (pite, cookie, [8], [9]) egy fájlban tárolt karakterlánc, melynek egyetlen kötelező eleme a sütit azonosító *név=érték* páros, de tartalmazhat további értékeket is `;`-vel elválasztva

egymástól. A süti bizonyos karaktereket nem tartalmazhat, ezért az értékét íráskor `escape()`-el, olvasáskor `unescape()`-el át- illetve vissza kell alakítanunk. A süti tartalmát a `document.cookie`-nak kell értékül adni.

A süti alapesetben ideiglenes: amikor a felhasználó bezárja a böngészőt, akkor érvényét veszti, ezért ha azt akarjuk, hogy a böngésző bezárása után is megmaradjon, akkor meg kell adni a lejárati időt `expires=`*érték* szintaxissal, ahol az értéket a dátumból `Date.toGMTString()`-el kapjuk meg. Például ha azt akarjuk, hogy a süti adott számú nap után járjon le, akkor:

```
var most = new Date();
var lejar = new Date(most.getTime() + nap*24*60*60*1000);
var suti = nev + "=" + escape(ertek);
suti += "; expires=" + lejar.toGMTString();
document.cookie = suti;
```

Sütit úgy törölhetünk, hogy a lejárati időt egy múltbeli időpontra állítjuk. Több sütit egyszerre úgy állíthatunk be, hogy a `document.cookie`-nak többször adunk értéket, mely nem írja felül a korábbi sütit, hanem mellette újabbat is létrehoz. Olvasáskor a `document.cookie` az oldalra érvényes összes süti *név=érték* párját tartalmazni fogja `;`-vel elválasztva. A süti beállítása akkor volt sikeres, ha a

```
document.cookie.indexOf("név=érték") != -1
```

feltétel teljesül.

3.3.4. A szabványosított DOM

A böngészők közötti különbségek csökkentése, illetve további lehetőségek biztosítása érdekében a W3C szabványosította a DOM kezelését. Terjedelmi korlátok miatt most csak a HTML DOM legfontosabb jellemzőit fogjuk ismertetni.

3.3.4.1. A DOM fa

A DOM-ban a weboldal tatalma egy fával ábrázolható ([10], [8], [7]), melyben a csúcspontok a weboldalban elhelyezett objektumoknak felelnek meg. Egy csúcspont akkor gyermeke egy másiknak, ha a megfelelő objektuma közvetlenül be van ágyazva a szülőnek megfelelő objektumba. Egy HTML tagba beágyazott szöveg is külön csúcspontnak felel meg.

A `nodeValue`, illetve `className` tulajdonságok tárolják egy adott csúcspont értékét (például a benne található szöveget), illetve a `class` jellemzőjének értékét. A HTML tartalom kelezésére nem szabványos, de széles körben elterjedt megoldás az `innerHTML` tulajdonság használata is.

HTML tagot tartalmazó új csúcspontot a `document.createElement(tag)`, szöveget tartalmazó új csúcspontot pedig a `document.createTextNode(szöveg)` metódusokkal tudunk létrehozni. Adott egyedi azonosítóval rendelkező csúcspontot a `document.getElementById(azonosító)`-val tudunk megkeresni. A document.`getElementsByTagName(tag)` az adott taggal rendelkező csúcspontok tömbjét adja vissza, melyben *elemnév*`=*` esetén az összes csúcspontot megkapjuk. Magára a dokumentumra a `document.documentElement`-el hivatkozunk.

Egy csúcspont első, utolsó, illetve összes gyermekét a `firstChild`, `lastChild`, illetve `childNodes[]` tulajdonságokkal tudjuk hivatkozni. A csúcspont előző, következő testvérét, illetve szülő csúcspontját a `previousSibling`, `nextSibling`, illetve `parentNode` tulajdonságok adják meg. Adott csúcspont beszúrására és törlésére vonatkozó tagfüggvényeket a szülő objektumra kell meghívni a *szülő*`.`*metódus*`(`*csúcspont*`)` formában:

- `appendChild(`*csúcspont*`)`: új csúcspontot helyez el a meglévő gyermekek mögött,
- `insertBefore(`*új_csp*`, `*régi_csp*`)`: új csúcspontot helyez el a megadott régi csúcspont előtt,
- `replaceChild(`*régi_csp*`, `*új_csp*`)`: új csúcspontra cseréli le a régit,
- `removeChild(`*csúcspont*`)`: eltávolítja a meglévő csúcspontot.

A `hasChildNodes()` értéke `true`, ha a csúcspontnak vannak gyermekei.

A `hasAttributes()`, illetve `hasAttribute(`*jellemző*`)` `true`-t adnak vissza, ha a csúcspontnak vannak jellemzői, illetve van paraméterként specifikált jellemzője. Adott jellemző értékét beállítani, lekérdezni, illetve törölni a `setAttribute(`*jellemző*`, `*érték*`)`, `getAttribute(`*jellemző*`)`, `removeAttribute(`*jellemző*`)` metódusokkal lehet.

3.3.4.2. A szabványos DOM eseménykezelése

Ha azt akarjuk, hogy egy esemény valamely objektumra vonatkoztatva eseménykezelő futtatását váltsa ki, akkor az eseményt regisztrálnunk kell az adott objektumra ([10]). Az események tovaterjedésének három fázisa van. Az *elfogási fázisban* az esemény a gyökérelem `document` objektumtól a célelem ősein keresztül a célelemig terjed. A célelem *elérésének fázisában* a célelemen regisztrált eseménykezelő hívódik meg. A *visszaterjesztés fázisában* az esemény visszaterjesztődik az ősökön keresztül a gyökérelemig. Ha valamelyik őselem regisztrálta magát az elfogási fázishoz, akkor az eseménykezelője meghívódik ebben a fázisban. Nem minden eseménynek van három fázisa, ugyanis bizonyos (például űrlapra vonatkozó) eseményeknek csak adott környezetben van értelme. Egy objektumra bekövetkező adott eseményhez akár több eseménykezelőt is regisztrálhatunk, melyek mind végre fognak hajtódni. A szabványos DOM központosíthatóvá teszi az eseménykezelést oly módon, hogy akár minden eseményt le tudunk kezelni a `document` gyökérelemben.

Egy esemény regisztrálása az `addEventListener(`*es*`, `*fgv*`, `*elfog*`)` metódussal lehetséges, ahol *es* az eseménytípus, *fgv* az esemény bekövetkeztekor végrehajtandó függvény, az *elfog* pedig akkor `true`, ha azt akarjuk, hogy az eseménykezelő az elfogási fázisban hívódjon meg, egyébként `false`. Ezáltal megadhatjuk, hogy az egymásba ágyazott objektumokhoz regisztrált eseménykezelők milyen sorrendben hívódjanak meg. Egy regisztrált eseménykezelőt a `removeEventListener()` metódussal távolíthatunk el, amelynek pontosan ugyanazok a paraméterei kell legyenek, mint amikor regisztráltuk. Ez lehetővé teszi az eseménykezelők ideiglenes használatát.

A szabványos DOM moduláris felépítésű, így az eseménykezelést is négy modul tartalmazza, melyek részletes leírását az olvasó megtalálja a [10] irodalomban. Az eseménykezelő függvény paraméterként megkapja az eseményt. Az eseményobjektumnak sok tulajdonsága és metódusa lehet, ezek közül lássunk néhányat:

- `stopPropagation()`: bármelyik köztes elemen leállítja az esemény tovaterjedését,
- `preventDefault()`: letiltja az alapértelmezett eseménykezelést,
- `type`: az esemény típusa karakterlánc formában, ahogyan regisztráltuk (például `click`),
- `target`: azon objektum, melyen az esemény fellépett,
- `currentTarget`: azon objektum, melyen az esemény feldolgozása folyik,
- `button`: szám, mely `mousedown`, `mouseup`, `click` eseményeknél azt jelzi, hogy mely egérgomb állapota változott meg; értékei rendre 0, 1, 2 a bal, középső, jobb gomb esetén (balkezes egérnél fordítva),
- `screenX`, `screenY`: az egér abszolút pozíciója a képernyő koordinátarendszerében,

- `clientX`, `clientY`: az egér relatív koordinátái a böngészőablakhoz képest; a görgetést nem veszik figyelembe, tehát nem a dokumentum koordinátarendszerében értendők
- `altKey`, `ctrlKey`, `shiftKey` értéke `true`, ha az esemény közben az Alt, Ctrl, vagy Shift gomb lenyomott állapotban volt.

3.3.4.3. Eseménykezelés Internet Explorerben

Az események regisztrálása és a regisztrálás megszüntetése az `attachEvent(`*es*`, `*fgv*`)`, illetve `detachEvent(`*es*`, `*fgv*`)` metódusokkal lehetséges, melyben az esemény típusa `on` előtaggal kezdődik (például `onclick`). Harmadik paraméterre nincs szükség, mert az Internet Explorer nem támogatja az események elfogási fázisát.

Internet Explorerben az eseményobjektum egy `window.event` nevű globális objektum, melynek legfontosabb fentiektől eltérő tulajdonságai:

- `srcElement`: az eseményt elszenvedő célelem,
- `button`: értékei 1, 2, illetve 4, melyek több gomb lenyomása esetén összeadódnak,
- `cancelBubble`: `true` értéke letiltja az esemény visszaterjedését
- `returnValue`: `false` értéke letiltja a böngésző alapértelmezett eseménykezelését.

3.4. Alapvető feladatok megoldása

3.4.1. A böngésző megállapítása és képességérzékelés

Sok esetben ugyanazt a működést a különböző böngészőkben másképp kell megvalósítanunk, viszont azt szeretnénk, hogy a programunk minél több böngészőben jól működjön. Legalább az Internet Explorer és a többi (szabványosabb) böngésző megkülönböztetésére sok esetben szükség van. A megfelelő kód futtatására az egyik lehetőség az, hogy lekérdezzük a felhasználó által használt böngészőt és annak megfelelően választjuk ki a végrehajtandó kódot. A böngésző lekérdezésénél abból indulhatunk ki, hogy a `navigator.userAgent` részletes leírást ad a felhasználó által éppen használt böngésző tulajdonságairól, így ebben kell egy olyan jellemző karaktersorozatot keresnünk, amelyből egyértelműen meg tudjuk állapítani a böngészőt. Például tároljuk el egy `explorer` nevű változóban, hogy a felhasználó Internet Explorer-t használ-e:

```
var ua = navigator.userAgent;
var explorer = (ua.indexOf("MSIE")> -1) ? 1 : 0;
```

Sajnos nincs garancia arra vonatkozóan, hogy a böngésző azonosítására általunk használt karaktersorozat az adott böngésző következő verziójában is változatlanul szerepelni fog, ezért sok esetben ajánlott egy kevésbé átlátható, de megbízhatóbb és hatékonyabb módszer használata, amit képességérzékelésnek neveznek ([7]).

A *képességérzékelés* lényege, hogy nem a böngésző típusát vizsgáljuk, hanem azt, hogy az adott tulajdonság vagy metódus használható-e a felhasználó által éppen használt böngészőben (bármi is legyen az), ha igen, akkor futtatjuk, ha nem, akkor pedig megvizsgáljuk annak más böngészőkben használt változatát és így tovább. A későbbiekben fogunk látni példákat képességérzékelésre, bár annak bemutatása érdekében, hogy az egyes böngészőkben jelenleg melyek a működő megvalósítások, nem mindig ezt fogjuk használni.

3.4.2. Objektumok beazonosítása

Hogyha egy létező objektummal valamilyen műveletet szeretnénk végezni (formázás, mozgatás stb.), akkor azt mindenekelőtt be kell azonosítanunk, azaz egy változóban meg kell kapnunk annak referenciáját.

Hogyha az objektum rendelkezik egyedi azonosítóval, legyen ennek értéke például `urlap`, akkor annak legegyszerűbb beazonosítása a következőképpen lehetséges:

```
var obj = document.getElementById("urlap");
```

Egy másik lehetőség, hogy kigyűjtjük az objektum típusának megfelelő taggal rendelkező összes objektumot egy tömbbe, és abban azonosítjuk be az általunk keresett objektumot (például ha tudjuk, hogy azok között hányadik). Például:

```
var obj = document.getElementsByTagName ("form")[0];
```

Ha az objektum rendelkezik `class` jellemzővel, akkor hasonlóképpen tehetnénk meg azt is, hogy lekérdezzük az adott jellemzőhöz tartozó objektumok halmazát egy tömbbe, majd ezek között már könnyebb lesz megtalálni a keresett objektumot (például ha tudjuk, hogy hányadik lesz). Csakhogy nem létezik beépített `getElementsByClass`, ezért ezt nekünk kell megírnunk. Az [59] alapján a függvényt úgy írjuk meg, hogy a `classN` csoporthoz tartozó (`class` jellemzővel rendelkező), `tipus` típusú (taggal rendelkező) és `csomopont` gyökerű részfában található objektumokat fogunk keresni.

```
function getElementsByClass (classN,csomopont,tipus) {
  var classElemek = new Array();
  if (!csomopont) csomopont = document;
  if (!tipus) tipus = "*";
  var tagok = csomopont.getElementsByTagName(tipus);
  var minta = new RegExp('(^|\\s)'+classN+'(\\s|$)');
  var tl = tagok.length;
  for (i = 0, j = 0; i < tl; i++) {
    if (minta.test(tagok[i].className) ) {
       classElemek[j] = tagok[i];  j++;
    } // if
 } // for
  return classElemek;
} // getElementsByClass
```

Először létrehozunk egy `classElemek` tömböt, ebben fogjuk beletenni a keresésnek megfelelő objektumok referenciáit. Ha a függvény meghívásakor nem adunk meg `csomopont`-ot, akkor a teljes dokumentumban keresi, ha pedig nem adunk meg `tipus`-t, akkor az adott `class` jellemzővel rendelkező objektum típusát nem vizsgálja. Egy `tagok` tömbbe kigyűjtjük a `csomopont` gyökerű részfában levő összes adott típusú objektumot, majd azokat az elemeket, melyeknek a `class` jellemzőjében benne van a `classN` azonosító, hozzáadjuk a `classElemek` tömbhöz, melyet a függvény visszaad. Ez a lehetőség persze nem csak egyetlen csúcs beazonosítására használható, hanem akkor is, ha adott részfához tartozó és/vagy adott HTML taggal rendelkező és/vagy adott `class` jellemzővel rendelkező csúcspontokkal szeretnénk valamilyen közös műveletet végezni.

Egy további lehetőség a beazonosításra, hogy a DOM fában egy már beazonosított másik objektumból indulunk ki és kapcsolatleíró tulajdonságokkal jutunk el az általunk keresett objektumhoz. Tegyük fel, hogy ismerjük a keresett objektum közvetlen gyermekét, szülőjét vagy testvérét, legyen annak referenciája `viszobj`. Ha a keresett objektum a `viszobj` szülője, akkor azt `viszobj.parentNode`-al, ha annak első vagy utolsó gyermeke, akkor `viszobj.firstChild`-al, illetve `viszobj.lastChild`-al, ha pedig annak jobb- vagy baloldali közvetlen testvére, akkor
`viszobj. nextSibling`-al, illetve `viszobj. previousSibling`-al kaphatjuk meg. Ha a keresett objektum a `viszobj`-nak gyermeke, de nem is első és nem is utolsó, akkor

`viszobj.childNodes[i]`-vel tudunk annak `(i+1)`-dik gyermekére hivatkozni, de ebben az esetben figyelnünk kell arra, hogy az elemek indexelése különbözhet az egyes böngészőkben (például a szóköz, sorvég karakterek helyén a böngésző üres szöveges csúcspontokat szúrhat be), illetve esetleges további csúcspontok beszúrásánál vagy törlésénél változhatnak az indexek. Ha nincs ilyen beazonosítottnak feltételezett `viszobj` objektum, akkor egy ilyen egy ilyen objektum beazonosítása történhet rekurzív módon ugyanígy, vagy bármely más módszerrel.

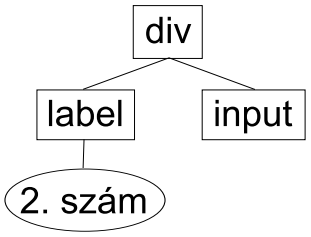
3.4.3. Csúcspontok létrehozása, beillesztése és törlése, tulajdonságok beállítása

A csúcspontok két legfontosabb fajtájával, a szöveges és HTML csúcspontokkal fogunk foglalkozni.

Tegyük fel, hogy létre szeretnénk hozni az alábbi kódnak megfelelő DOM részfát:

```
<div class="bevitel"
     style="margin-top: 0.25em; margin-bottom: 0.25em;">
  <label for="s11">2. szám:</label>
  <input id="s11" style="text-align: right;" size="5" />
</div>
```

A DOM részfa a 7. ábrán látható:



7. ábra: DOM részfa

Szöveges csúcspont szöveget tartalmaz, és a következőképpen tudjuk létrehozni:

```
var szoveg = document.createTextNode("2. szám:");
```

HTML csúcspont egy HTML objektumot tartalmaz és a `document.createElement()` függvénnyel tudjuk létrehozni, melynek paraméterként a létrehozandó objektum tagját kell megadni. Például egy `input` beviteli mező létrehozása:

```
var bemezo = document.createElement("input");
```

Ha az előző szöveget az input mező feliratának szánjuk, akkor létre kell hoznunk egy `label` objektumot is:

```
var labbeszoveg = document.createElement("label");.
```

A szöveges csúcspontot a `label` csúcspontba úgy fogjuk beletenni, hogy a DOM fában hozzáadjuk gyerekként:

```
labbeszoveg.appendChild(beszoveg);.
```

Végül létrehozzuk a `div` objektumot és hozzáadjuk gyerekként a `label` és input objektumokat:

```
var besor = document.createElement("div");
besor.appendChild(labbeszoveg);
besor.appendChild(bemezo);
```

 © Holló Csaba, SzTE

Ahhoz, hogy a `div` csúcspontú részfa megjelenjen, még hozzá kell adnunk az eredeti dokumentum már látható részfájához a kívánt helyre. Például, ha a `sor2` referenciájú objektum elé, tehát annak bal testvéreként szeretnénk beszúrni, akkor:

```
sor2.parentNode.insertBefore(besor,sor2);
```

mivel a beszúrás és törlés műveleteket mindig a szülő csúcspontra hívjuk meg.

A DOM részfával tehát elkészültünk, de hogyan lehet a jellemzőket és stílustulajdonságokat beállítani? Jellemzők esetén két lehetőség is van:

`bemezo.size = "5";` vagy `bemezo.setAttribute("size","5");`, illetve

`besor.className = "bevitel";` vagy

```
if (explorer){besor.setAttribute("className","bevitel");
} else {besor.setAttribute("class","bevitel");}.
```

Stílustulajdonságnál a

```
bemezo.style.textAlign = "right";
```

forma ajánlott.

Végül, hogyha a `div` objektumot törölni szeretnénk, akkor:

```
var valami = besor.parentNode; valami.removeChild(besor);.
```

3.4.4. Eseménykezelés

Ha egy `obj` referenciával rendelkező objektum kattintás eseményéhez a `szamTorles` függvényt szeretnénk hozzárendelni, akkor ezt megtehetjük az `obj.onclick = torolSzam;` értékadással, de ez nem a legjobb módszer, mert így nem lehetséges több eseménykezelő hozzárendelése, ezért ajánlottabb a szabványosított DOM eseménykezelés használata:

```
if (explorer){obj.attachEvent("onclick", torolSzam);}
else {obj.addEventListener("click", torolSzam, true);},
```

vagy ugyanez képességérzékeléssel:

```
if (obj.attachEvent){obj.attachEvent("onclick", torolSzam);}
else if (obj.addEventListener){
  obj.addEventListener("click", torolSzam, true);}
else { obj.onclick = torolSzam;}.
```

3.4.5. Űrlapok kezelése

Űrlapmezők létrehozása és tulajdonságainak beállítása ugyanúgy lehetséges, mint más objektumok esetén. Van azonban néhány olyan feladat, amit többnyire vagy kizárólag csak űrlapmezők esetén szoktunk elvégezni.

Gyakori eset, hogy azt szeretnénk, hogy bizonyos űrlapmezők tartalma az addig kitöltött más űrlapmezők tartalmától függően jelenjen meg. Például hasznos, ha egy dátum napjának kiválasztásakor a megjelenő napok listáját az évtől (szökőév-e) és a hónaptól függően jelenítjük meg. Ilyenkor az év vagy hónap módosításakor a napok listáját módosítani kell. Egy mező változtatását a `change` eseménnyel tudjuk ellenőrizni és az eseménykezeléséhez hozzá kell adni, hogy változtatás esetén hívja meg az összes tőle függő mező megfelelő módosítását megvalósító függvényt.

Egy másik tipikus feladat a szövegmezők elküldés előtti ellenőrzése. Ezt általában a mező értékének módosításakor (`change` esemény), a mező elhagyásakor (`blur` esemény), illetve az űrlap elküldésekor (`onsubmit` esemény) szoktuk megtenni. Ha ellenőrizni szeretnénk, hogy a mező ki lett-e töltve, akkor az utóbbira mindenképpen szükség van. Ha a mező elhagyásához és módosításához is írunk eseménykezelőt, és nem szeretnénk, hogy mindkettő esetén a közös tevékenységek kétszer hajtódjanak végre, akkor egy változó segítségével tesztelnünk kell, hogy a másik eseménykezelő lefutott-e. Ha rossz érték esetén nem akarjuk megen-

gedni a mező elhagyását, akkor a fókuszt a következőképpen visszatesszük a mezőbe: `setTimeout(function(){mezo.focus()}, 10);`. Célszerű a javítandó mezőt ki is jelölni: `mezo.select();`.

Egy csoportba tartozó rádiógombok elérhetők az `urlap.nev[]` tömb elemeiként, ahol a `nev` a rádiógombok közös name jellemzőjének értéke. Bejelölésük ellenőrzése a következőképpen lehetséges: `if(urlap.name[i].checked){ ... }`.

Ezzel szemben az egy csoportba tartozó jelölőnégyzeteknek nem ajánlatos azonos name értékeket adni, mert az elküldés után nehezebb lesz azonosítani a válaszokat. Elérhetjük őket egyenként az `id`-jük segítségével, de egyszerűbb lenne az azonos csoportba tartozókat egy tömbbe kigyűjteni. Egy lehetséges megoldás, hogy azonos osztályba tesszük őket, és az objektumok beazonosításáról szóló fejezetben megírt `getElementsByClass()` függvénnyel kigyűjtjük. A bejelölések ellenőrzése a választógombokhoz hasonlóan az `if(checkTomb[i].checked){ ... }` módon lehetséges.

Ha valamelyik mezőnél hibát észlelünk, akkor egyértelműen ki kell írni, hogy mi a probléma és célszerű a hibás mezőket el is színezni. Erre az egyik lehetséges megoldás: `mezo.style.color = "#f00";`, de létezik ennél elegánsabb megoldás is, melyben a mezőt hozzárendeljük egy hibás mezők formázását leíró osztályhoz (is). Például, ha a mező eredetileg egy „valami” osztályhoz tartozik, akkor most a „valami” és „hiba” osztályokhoz is fog tartozni (azaz class =„valami hiba” tulajdonságot adunk neki), amit megtehetünk a mezo.className = “valami hiba”; értékadással, ugyanakkor látható, hogy a „hiba” hozzáadása, illetve eltávolítása karakterlánc műveletekkel is megvalósítható.

Alapesetben űrlap elküldésének kezeléséhez annak `onsubmit`, az eredeti űrlapmező értékek visszaállításának kezeléséhez pedig az űrlap `onreset` eseményéhez rendelünk eseménykezelő függvényeket. Azonban, ha az eseménykezelő függvényben ezen események után még más tevékenységeket is végre akarunk hajtani, akkor az eseménykezelő függvényeket a megfelelő gombokra való kattintáshoz rendeljük. Például, ha a mezők visszaállítása után még valami más szöveget is törölni szeretnénk a képernyőről egy `torolSzoveg` függvénnyel, és be akarjuk állítani a fókuszt valamelyik mezőre, akkor a törlés gombnak adunk egy `torles` egyedi azonosítót és:

```
document.getElementById("torles").onclick = torol;
```

majd pedig

```
function torol(){
  var tor = confirm("Biztosan mindent törölni akarsz?");
  if (tor) {
    document.getElementById("urlap").reset();
    torolSzoveg();
  }
  var s1m = document.getElementById("s1");
  s1m.focus();
  return tor;
}.
```

Ahhoz, hogy az űrlap ne küldődjön el, vagy a visszaállítás ne történjen meg, az eseménykezelő függvénynek `false`-ot kell visszaadnia, vagy ha a böngésző lehetővé teszi, az eseményre meg kell hívni a `preventDefault()`-ot. Például:

```
function elkuld(e){
     if (e && e.preventDefault){e.preventDefault();}
     else {return false;}
```

```
}.
```

Az űrlapok tartalmát többnyire valamilyen kiszolgáló oldali programnak szoktuk elküldeni és ott további ellenőrzéseket kell végezni biztonsági okokból, illetve a szerveren tárolt adatok segítségével (például felhasználó hitelesítése érdekében). Gyakorlás végett azonban most azt nézzük meg, hogyan tudjuk kinyerni a mezők tartalmát akkor, ha az űrlapot egy másik HTML oldalnak küldjük tovább GET módszerrel. Használni fogunk szabályos kifejezést és látni fogjuk, hogy milyen átalakításokat kell végeznünk a kapott kereső karakterláncon, melynek formátumát a 2.8.1. fejezetben írtuk le. A mezők tartalmát most csak kiíratjuk egy `AdatKiir` azonosítóval rendelkező mezőbe, de természetesen más műveleteket is végezhetnénk.

```
if (window.attachEvent){
  window.attachEvent("onload", feldolgoz);}
else if (window.addEventListener){
  window.addEventListener("load", feldolgoz, true);}
else { window.onload = feldolgoz;}
function feldolgoz(){
 adatsor = "";
 kif = document.location.search;
 kif = kif.slice(1);
 szk = /\+/g;
 kif = kif.replace(szk," ");
 t = kif.split("&");
 for(var i=0; i<t.length;i++){
  t[i] = t[i].split("=");
  for(var j=0; j<t[i].length;j++){
   t[i][j] = unescape(t[i][j]);
   adatsor += "t["+i+","+j+"]:" +t[i][j] + "   ";
  }
  adatsor += "<br />";
 }
 document.getElementById("AdatKiir").innerHTML = adatsor;
}
```

GET módszer esetén az űrlap adatai az URL-beli lekérdező karakterláncban kerülnek elküldésre, melyet a `document.location.search`-el tudunk megkapni, de a szükségtelen első ? karaktert `slice(1)`-el törölnünk kell belőle. Utána az összes + jelt visszaalakítjuk szóközre, a kapott stringet a `split` függvénnyel feldaraboljuk az `&` jelek mentén, mely az egyes darabokat elhelyezi egy tömbben, ez lesz nálunk a `t`. Ily módon a `t` tömb elemei az elküldött mezők lesznek *mezőnév=érték* formában. Ezeket is feldaraboljuk az = jelek mentén és így már külön tömbelemekben lesznek a mezőnevek és a hozzá tartozó értékek. Annyi dolgunk van még, hogy ezekben az ékezetes és speciális karaktereket visszaalakítsuk, és mivel most ki szeretnénk íratni, ezért elhelyezzük a kiíratandó `adatsor` stringben. Végül az `adatsor` stringet az `AdatKiir` mezőben megjelenítjük.

Érdemes a fentieken egy tetszőleges űrlapon kipróbálni és megfigyelni, hogy a választómezők esetén csak a bejelölt mezők küldődnek el `on` értékkel, az azonos névvel rendelkező rádiógombok közül csak a kiválasztott küldődik el a neki megfelelő értékkel, míg választólista esetén, amennyiben engedélyezve volt több érték kiválasztása, akkor a lista neve minden értékkel együtt elküldődik, mintha külön választómezők lennének azonos névvel. Láthatjuk tehát, hogy a mezőnév-érték párok küldésénél a mezőnévnek nem kell egyedinek lennie, sőt a

szabvány így definiálja tömbök küldését. Ily módon lehetővé válik nem csak választómezők, hanem tetszőleges tömbök programozott küldése is, továbbá számos programozási nyelv képes a küldött adatok fogadásánál ezeket eleve tömbként szolgáltatni.

3.4.6. Objektumok generálása

Bizonyos feladatoknál sok, azonos típusú objektumot kell létrehozni. Ha ezek attribútumai és tartalmazott értékei valamilyen képlettel megadhatók, akkor lehetőség van a kódjaik egyenként, kézzel történő beírását automatikus generálással helyettesíteni.

Példaként szeretnénk kiíratni az 1-től 42-ig terjedő egész számokat és azok köbét. A táblázatunknak tehát két oszlopa lesz, az elsőben a szám, a másodikban annak köbe helyezkedik el. Ugyanakkor érdemes figyelembe vennünk azt is, hogy a felhasználók nem szeretnek sokat görgetni és esetleges nyomtatásnál papírt pazarolni, ezért egyetlen keskeny és nagyon magas táblázat helyett az értékek kiíratását felosztjuk hat táblázatba, melyek mindegyike hét értéket tartalmaz és ezeket egymás mellé helyezzük el. Végül, annak érdekében, hogy szemléltessük a tulajdonságok szkriptben történő beállítását is, minden táblázatban létrehozunk egy két oszlop szélességű fejléc cellát is.
A böngészőben tehát a 8. ábrán látható tartalmat szeretnénk megjeleníteni:

Számok köbe:

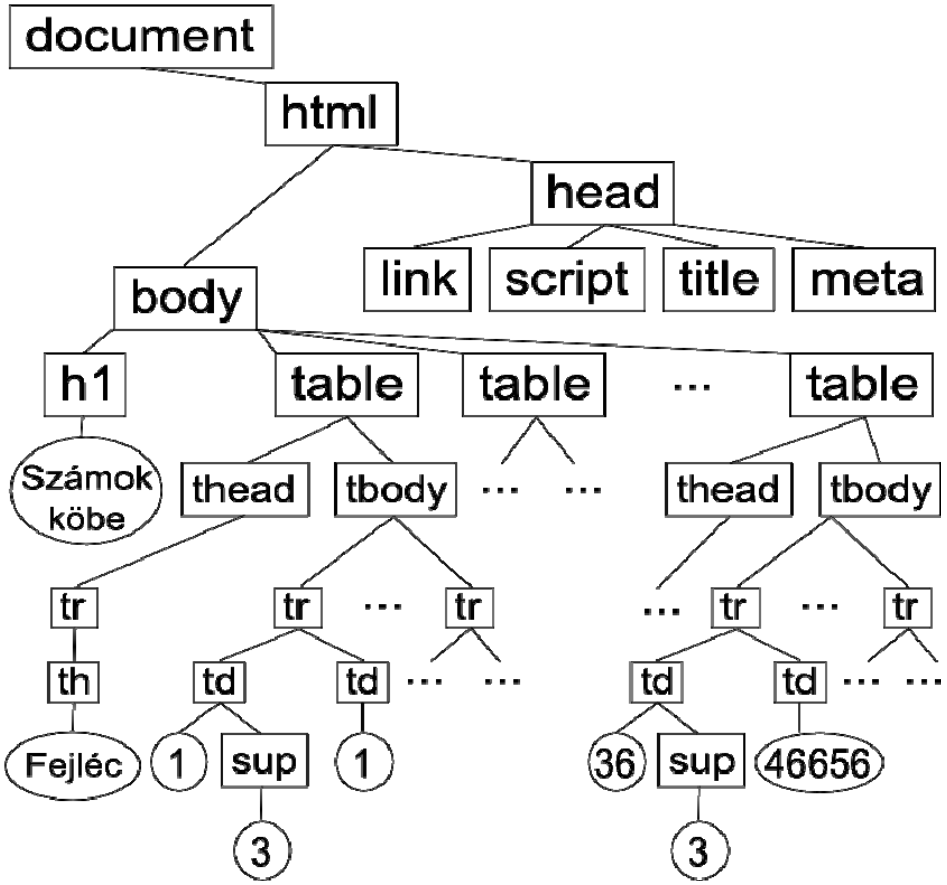
Fejléc		Fejléc		Fejléc		Fejléc		Fejléc		Fejléc	
1^3	1	8^3	512	15^3	3375	22^3	10648	29^3	24389	36^3	46656
2^3	8	9^3	729	16^3	4096	23^3	12167	30^3	27000	37^3	50653
3^3	27	10^3	1000	17^3	4913	24^3	13824	31^3	29791	38^3	54872
4^3	64	11^3	1331	18^3	5832	25^3	15625	32^3	32768	39^3	59319
5^3	125	12^3	1728	19^3	6859	26^3	17576	33^3	35937	40^3	64000
6^3	216	13^3	2197	20^3	8000	27^3	19683	34^3	39304	41^3	68921
7^3	343	14^3	2744	21^3	9261	28^3	21952	35^3	42875	42^3	74088

8. ábra: Generált táblázat

A generálás lényege, hogy az objektumokat ciklusokban hozzuk létre. Viszont felmerül a kérdés: hogyan hozzuk létre ezeket az objektumokat és hogyan jelenítjük meg őket?

Az egyik lehetséges megoldás, hogy a szükséges HTML kódot `document.write`-okkal beleíratjuk az oldal kódjába. Így létrejönnek az objektumok és azonnal meg is jelennek. Ennél valamivel barátságosabb megoldás az, amikor a kódot egy karakterláncban állítjuk összse, és azt egyetlen lépéssel íratjuk bele az oldal kódjába.

A legszebb megoldás azonban a szabványosított DOM fa kezelésével valósítható meg: létrehozzuk a kívánt HTML kódnak megfelelő csúcspontokat, felépítjük a megjeleníteni kívánt tartalom DOM fáját és azt hozzáadjuk a dokumentum DOM fájához. A fenti táblázathoz tartozó, felépítendő DOM fát a 9. ábrán láthatjuk.



9. ábra: A táblázathoz tartozó DOM fa

Táblázatok szkriptben történő létrehozásakor ajánlott a `thead` és `tbody` objektumokat is létrehozni. A fenti táblázatok generálása a következőképpen történhet:

```
var cim = document.createTextNode("Számok köbe:");
var cimsor = document.createElement("h1");
cimsor.appendChild(cim);
document.body.appendChild(cimsor);
var tabl, tablh, tablb, sor;
var cella1, tart1, cella2, tart2;
for(i=1; i<=6; i++){
  tabl = document.createElement("table");
  tablh = document.createElement("thead");
  sor = document.createElement("tr");
  cella1 = document.createElement("th");
  cella1.colSpan = "2";
  cella1.style.textAlign = "center";
  tart1 = document.createTextNode("Fejléc");
  cella1.appendChild(tart1);
  sor.appendChild(cella1);
  tablh.appendChild(sor);
  tabl.appendChild(tablh);
```

```
  tablb = document.createElement("tbody");
  for (j=7*(i-1)+1; j<=7*(i-1)+7; j++){
    sor = document.createElement("tr");
    cella1 = document.createElement("td");
    tart1 = document.createTextNode(j);
    cella1.appendChild(tart1);
    tart1s = document.createElement("sup")
    harom = document.createTextNode("3")
    tart1s.appendChild(harom)
    cella1.appendChild(tart1s);
    sor.appendChild(cella1);
    cella2 = document.createElement("td");
    cella2.className = "jobb";
    tart2 = document.createTextNode(j*j*j);
    cella2.appendChild(tart2);
    sor.appendChild(cella2);
    tablb.appendChild(sor);
  } // for j
  tabl.appendChild(tablb);
  document.body.appendChild(tabl);
} // for i
```

Azonban ahhoz, hogy ez szép táblázatos kinézetet kapjon és a táblázatok egymás mellé kerüljenek, az XHTML fájlhoz csatolt CSS-ben a táblázatokat formázni is kell:

```
td {border: 3px dotted #f0f;
    padding: 0.4em;}
table {border: 3px ridge #00f;
       margin-right: 10px; float: left;}
.jobb{text-align: right;}.
```

A táblázat blokk szintű elem, de mivel balra úsztattuk, ezért a következő táblázat mindig annak jobb oldalára kerül, amíg fér.

3.4.7. Tevékenységek végrehajtása az oldal betöltésekor

Az esetek többségében vannak olyan feladatok, melyeket közvetlenül az oldal betöltődése után kell elvégeznünk. Például még az oldal felhasználó általi birtokba vétele előtt el kell helyeznünk eseménykezelőket, vagy el kell indítanunk bizonyos tevékenységeket. Az AJAX tanulmányozásakor további példákat is fogunk látni ilyen helyzetekre. Az ilyen tevékenységek megvalósításában azonban a legtöbb esetben hivatkoznunk kell az XHTML fájl `body` részében létrehozott objektumokra. Mivel a szkriptet, illetve a szkriptfájl csatolását a legtöbb esetben a `head`-ben helyezzük el, ezért annak beolvasásakor az említett objektumok még nem léteznek, így hivatkozásaik hibajelzést adnának. A megoldás az, hogy ezeket a tevékenységeket a `window.onload` eseménykezelőjében helyezzük el, mely közvetlenül az oldal beolvasása után automatikusan meghívódik. A következő fejezetek példáiban használni fogjuk ezt a lehetőséget.

3.4.8. Objektumok mozgatása

JavaScript segítségével könnyen megtehetjük, hogy a képernyőn mozgó objektumokat jelenítsünk meg oly módon, hogy a programban változtatjuk az objektum pozícióját ([8]). A mozgó objektum bármilyen típusú lehet.

Az alábbi példában egy képet fogunk mozgatni, nevezetesen egy hattyú fog úszni a tavon.

Az XHTML `head` részében csatoljuk a CSS és szkript fájlokat, a body részben pedig elhelyezzük a mozgatni kívánt objektumot:

```
<head> …
<link rel="stylesheet" type="text/css" href="mozog.css" />
<script src = "mozog.js" type = "text/javascript"> </script>
</head>
<body id = "torzs">
  <div>
      <img src = "../Kepek/hattyu1k.jpg" id = "hattyu"
           width = "49" height = "106" alt = "hattyu" />
  </div>
</body>
```

A szkriptben az oldal betöltésekor szeretnénk elvégezni néhány inicializálást és elindítani az objektum mozgatását. Ezt úgy tudjuk megtenni, hogy ezeket a tevékenységeket elhelyezzük egy `indit` függvényben, és ezt csatoljuk egy eseménykezeléshez, melynek értelmében az `indit` függvény közvetlenül az oldal betöltődése után kerüljön meghívásra:

```
if (window.attachEvent){window.attachEvent("onload", indit);}
else if (window.addEventListener){
  window.addEventListener("load", indit, true);}
else { window.onload = indit;}
```

Az `indit` függvényben először is beállítjuk a hátteret. Jogos a kérdés: ezt miért nem a stílusfájlban tesszük meg? Azért, mert a háttérkép különböző felbontásokban különbözőképpen jelenik meg, és ha nem szeretnénk, hogy adott felbontásban a hattyú a tóparti fák tetején ússzon, akkor le kell kérdeznünk a felhasználó által használt felbontást és olyan háttérképet kell megjelenítenünk, mely az adott felbontásban megfelelő. A háttérkép beállításához egy `hatter` függvényt használunk, melyet az `indit`-ból hívunk meg.

```
function hatter(){
 var t = document.getElementById("torzs");
 if (screen.width < 1024 && screen.height < 768){
   t.style.backgroundImage = "url(../Kepek/tapolca2k.jpg)";
 } else{
   t.style.backgroundImage = "url(../Kepek/tapolca2n.jpg)";
 }
} // hatter()
```

A képernyő szélességét és magasságát (azaz a felbontást) a `screen.width` és `screen.height`-al tudjuk lekérdezni. Jelenleg csak két esetet különböztettünk meg, de szükség esetén többet is el lehet különíteni.

A továbbiakban a mozgatandó képet el kell helyeznünk a kezdeti pozíciójába. A kép pozícióját mindig százalékban adjuk meg, ugyanis ez lehetővé teszi az objektum felbontás független mozgatását. A mozgatás úgy fog megvalósulni, hogy többször meghívunk egy olyan `uszik` függvényt, mely az objektumot áthelyezi egy másik pozícióba. A többszöri meghívás történhet úgy, hogy az `uszik` függvény rekurzívan meghívja önmagát valamilyen feltétel bekövetkeztéig (például az objektum a képernyő széléhez ér), vagy már az `indit` függvényben annak többszöri, ciklikus meghívását írjuk elő. Az utóbbi esetben egy lehetséges megvalósítás a `window.setInterval` használata, mely az első paraméterében megadott

utasításokat (jelenleg az `uszik();` függvényhívást) a második paramétereként megadott milliszekundumok elteltével ismételi meg. Az `indit` függvényünk tehát így fog kinézni:

```
function indit(){
 hatter();
 h = document.getElementById("hattyu");
 hl = 0; ht = 82;
 h.style.position = "absolute";
 h.style.left = hl + "%";
 h.style.top = ht + "%";
 id = window.setInterval("uszik();",100);
} // indit()
```

Hátra van még az `uszik` függvény megvalósítása, melyben az objektumot a következő pozíciójára helyezzük, illetve ha az már nem szükséges, akkor az `uszik` ismételt meghívását leállítjuk. A leállításhoz a `window.clearInterval` függvényt használjuk, melynek paramétereként meg kell adnunk a leállítani kívánt `window.setInterval` által visszaadott egyedi azonosítót.

```
function uszik(){
 if (hl < 90){
   hl = hl + 1; ht = ht - 0.25;
   h.style.left = hl + "%";
   h.style.top = ht + "%";
 } else {
   window.clearInterval(id);
 }
} // uszik()
```

3.4.9. Az egérre és billentyűzetre vonatkozó események használata

Az egérre és billentyűzetre vonatkozó legfontosabb események használatát egy olyan példa segítségével mutatjuk be, melyben lehetővé tesszük a felhasználó számára, hogy egy objektumot egérrel, illetve billentyűzettel is tudjon mozgatni a képernyőn. Az egérrel történő mozgatás esetén a felhasználó a kép fölött lenyomja az egér bal oldali gombját, az egeret az objektummal együtt a kívánt pozícióba húzza, majd ott elengedi az egeret és az objektum ott marad. Jelen esetben egy olyan `div` objektumot fogunk mozgatni, mely tartalmaz egy képet és egy ahhoz tartozó szöveget, de bármi mást is tartalmazhatna. Az XHTML fájlunk tartalma a következő lesz:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title> Kép mozgatása </title>
<meta http-equiv="Content-Type"
          content="text/html;charset=ISO-8859-2" />
<link rel="stylesheet" type="text/css" href="mozgathato3.css"
/>
<script type = "text/javascript" src = "mozgathato3.js">
</script>
</head>
<body>
```

```
<h1> Kép mozgatása egérrel és az u, i, k, m, n, b, h, z, illetve kurzormozgató billentyűkkel (nyilakkal) </h1> <div id="mozgathato">
     <img src="../Kepek/kutya1mk.jpg" alt = "Mozgatható kutya"
             height="88" width="114"  />
     <span> Képhez tartozó szöveg </span>
 </div>
 </body>
 </html>
```

A társított JavaScript fájl elején deklaráljuk a globális változókat:

```
var egerX, egerY, mX, mY, mp, e, mozgat = false;
```

Az `egerX`, `egerY` változókban az egér, míg az `mX`, `mY` változókban a kép pozícióját tároljuk. Az `mp` változóval a mozgatandó `div` objektumot azonosítjuk, `e`-ben pedig az éppen aktuális eseményt tároljuk el. A `mozgat` változóban azt tároljuk, hogy adott pillanatban az egér mozgatásakor a `div` objektumunkat kell-e mozgatni: a `false` érték jelenti azt, hogy nem, a `true` pedig azt, hogy igen.

Közvetlenül az oldal beolvasása után, még mielőtt azt a felhasználó elkezdené használni, számos inicializálást kell végeznünk. Ennek érdekében az oldal betöltődésekor meghívunk egy `init` függvényt.

```
if (window.attachEvent){
  window.attachEvent("onload", init);
} else {
  window.addEventListener("load", init, true);
}
```

Az `init` függvényben először is elhelyezzük a mozgatandó objektumunkat a kívánt eredeti pozícióba. Ezután az egérgomb lenyomásához, felengedéséhez és az egér húzásához hozzárendeljük a `gombLe`, `gombFel` és `gMozog` függvényeket. Végül billentyűk lenyomásához és nyomva tartásához is hozzárendeljük a `bMozgat` függvényt. Gyakorlásképpen a `body` objektumot ezúttal a `getElementsByTagName` függvénnyel azonosítjuk. A megoldás sajnos böngészőfüggő.

```
function init(){
  mp = document.getElementById("mozgathato");
  mp.style.position = "absolute";
  mp.style.left = "200px";
  mp.style.top = "200px";
  if (mp.addEventListener){
    mp.addEventListener("mousedown", gombLe, false);
    mp.addEventListener("mouseup", gombFel, false);
    mp.addEventListener("mousemove", gMozog, false);
    document.onkeydown = bMozgat;
    document.onkeypress = bMozgat;
  } else {
    mp.attachEvent("onmousedown", gombLe);
    mp.attachEvent("onmouseup", gombFel);
    mp.attachEvent("onmousemove", gMozog);
    var bd = document.getElementsByTagName("body")[0];
    bd.attachEvent("onkeydown", bMozgat);
    bd.attachEvent("onkeypress", bMozgat);
```

```
  }
} // init()
```

A helyes működés érdekében bizonyos esetekbe le kell tiltani az általunk használni kívánt események alapértelmezett eseménykezelését. Jelenleg például az egér lenyomott gombbal történő húzása az objektum kijelölését eredményezné, viszont mi teljesen más működést szeretnénk. Az alapértelmezett eseménykezelés letiltását böngészőtől függően a `gombLe`, illetve `gMozog` függvényekben kell megtennünk, megvalósítása `e.preventDefault();`-al, illetve `e.returnValue = false;`-al lehetséges, ahol `e` az esemény.

Az egérgomb lenyomásakor a mozgat változó értékét `true`-ra állítjuk, ezáltal engedélyezzük, hogy az egér mozgatásakor az objektum mozgatható legyen.
Lekérdezzük és az `egerX`, `egerY`, `mX`, `mY` változókban eltároljuk az egér, illetve a mozgatandó objektum egérgomb lenyomásakor aktuális pozícióját. A fentiek alapján a `gombLe` függvény a következőképpen fog kinézni.

```
function gombLe(esemeny){
  mozgat = true;
  e = esemeny || window.event;
  egerX = e.clientX;
  egerY = e.clientY;
  mX = parseInt(mp.style.left);
  mY = parseInt(mp.style.top);
  if (e.preventDefault){
    e.preventDefault();
  }
} // gombLe()
```

Az egér gombjának lenyomásához tartozó pozíciójának lekérdezéséhez szükségünk van az eseményre, melyet szabványosan az eseménykezelő függvény paramétereként, Internet Explorerben azonban globális `window.event` eseményként kapunk meg.

A mozgatandó objektum beazonosítására itt most nincs szükség, mivel egyetlen ilyen objektumunk van, melyet az `init` függvényben való elhelyezés kapcsán már beazonosítottunk. Amennyiben több mozgatható objektumunk is lenne, akkor viszont be kellene azonosítanunk, hogy a felhasználó melyik objektumot akarja mozgatni, amire egy lehetőség lenne:

```
var t = e.target || e.srcElement;
mp = t.parentNode;
```

Böngészőtől függően `e.target`-el vagy `e.srcElement`-el megkapjuk azt az objektumot, amely fölött a felhasználó lenyomta az egér gombját. Viszont hogyha egy egész objektumcsoportot akarunk azzal együtt mozgatni (például jelen esetben azt akarnánk, hogy a kép együtt mozogjon a szöveggel), akkor a ténylegesen mozgatandó objektum a beágyazó objektum lenne, melyet jelen esetben a `t` objektum szülőjeként kapnánk meg.

Az egérgomb felengedésekor a `gombFel` függvényben visszaállítjuk a `mozgat` változó értékét `false`-ra, annak érdekében, hogy a továbbiakban az egér mozgatása esetén is a mozgatandó objektumunk a helyén maradjon.

```
function gombFel(){
  mozgat = false;
} // gombFel()
```

Az objektum mozgatását a `gMozog` függvényben valósítjuk meg. Figyelnünk kell arra, hogy csak akkor szabad mozgatnunk az objektumot, ha a felhasználó előzőleg a mozgatandó objektum fölött lenyomta az egérgombot, és azóta nem engedte fel, azaz a `mozgat` változó

értéke `true`. A mozgatás úgy történik, hogy lekérdezzük az egér új pozícióját és annak relatív elmozdulásával elmozdítjuk a mozgatandó objektumot. A `gMozog` függvény kódja:

```
function gMozog(esemeny){
  if (mozgat==true){
    var XMozdul, YMozdul;
    e = esemeny || window.event;
    xx = e.clientX;
    yy = e.clientY;
    XMozdul = xx - egerX;
    YMozdul = yy - egerY;
    mp.style.left = mX + XMozdul + "px";
    mp.style.top = mY + YMozdul + "px";
    e.returnValue = false;
  } // if (mozgat== true)
} // gMozog()
```

Billentyűkkel történő mozgatás esetén a `bMozgat` függvényben először böngészőtől függően az esemény `keyCode` vagy `which` tulajdonságával megállapítjuk a lenyomott billentyű kódját. Jelenleg a `wich` Firefoxban az alfanumerikus karakterekre, a `keyCode` a többi karakterre és számos más böngészőben működik, de a képességérzékeléses megoldás miatt nem kell ezt részletesen követni, elegendő, hogyha a két tulajdonság közül az egyik működik, és az helyesen adja vissza a kódot. A lenyomott karakter kódja alapján meghatározzuk, hogy vízszintesen és függőlegesen mennyit kell elmozdulnia a mozgatandó objektumnak. Ezután lekérdezzük az objektum aktuális pozícióját és azt a megfelelő mértékben elmozdítjuk.

```
function bMozgat(esemeny){
  var bill;
  e = esemeny || window.event;
  bill = e.keyCode ? e.keyCode : e.which;
  var deltaX = 0;
  var deltaY = 0;
  switch(bill){
    case 75:
    case   39: deltaX = 5; break;
    case 85:
    case   38: deltaY = -5; break;
    case 72:
    case 37: deltaX = -5; break;
    case 78:
    case 40: deltaY = 5; break;
    case 73:  deltaX = 5; deltaY = -5; break;
    case 90:  deltaX = -5; deltaY = -5; break;
    case 77:  deltaX = 5; deltaY = 5; break;
    case 66:  deltaX = -5; deltaY = 5; break;
  } // switch(bill)
  mX = parseInt(mp.style.left);
  mY = parseInt(mp.style.top);
  mp.style.left = mX + deltaX + "px";
  mp.style.top = mY + deltaY + "px";
} // bMozgat()
```

4. Java szervletek

4.1. Alapfogalmak

Ebben a fejezetben a kiszolgáló oldali web programozás alapjaival ismerkedünk meg, melyet olyan feladatok megoldására szoktak használni, melyek ügyfél oldalon nem megfelelően kezelhetők. Például szerver oldalon lehetőség van a felhasználók hitelesítésére és személyre szabott kezelésére, illetve az adatok szerveren történő, többnyire adatbázisban való tárolására. A jegyzet keretei között a szerver oldali programozásnak csak a legalapvetőbb fogalmait, problémáit és lehetőségeit tudjuk bemutatni, további tudnivalókat az olvasó a fejezet alapvető forrásaként is szolgáló [9], [10] és [13] irodalmakban talál.

A *webszerver* egy program, mely HTTP(S) kéréseket fogad, feldolgoz és válaszokat küld, melyhez kapcsolódóan számos szolgáltatást nyújt. Például a kapott URL-t lefordíthatja fájlnévre, vagy programnévre, melyet futtat és a program által előállított és elküldött eredményt küldi vissza. Bár webszervernek szokták nevezni azt a gépet is, amin ez a program fut, illetve léteznek speciális hardverek is, mi a továbbiakban webszerver alatt a webes szolgáltatásokat nyújtó alkalmazást fogjuk érteni. Példáinkban a legelterjedtebb ingyenesen használható Apache Tomcat webszervert fogjuk használni, mely letölthető a [23] címről és a [24]-nek megfelelően ingyenesen használható.

A *szervlet* a webkiszolgáló funkcionalitását bővítő Java nyelvű program ([10], [61], [62]). A szervlet lehet generikus vagy HTTP szervlet, mi az utóbbival fogunk foglalkozni.

A HTTP szervlet kommunikációja során a HTTP protokollt használja, HTTP kérést kaphat és HTTP választ generál. A HTTP szervlet alkalmas az ügyféloldalról elküldött információk feldolgozására, szerver oldali tárolására és dinamikus tartalom generálására.

A Java EE alkalmazásszerver azon komponensét, mely az URL-ek szervletekhez való hozzárendelését és a szervletek életciklusainak a kezelését végzi, *szervlet konténernek* nevezzük. Árban, funkcionalitásban, teljesítményben eltérő szervlet konténerek léteznek.
Míg régebben szerver oldalon kliensenként külön programok futottak és a szálkezelést is teljes egészében a programozónak kellett megvalósítania, ma már a webkonténereknek számos kényelmi és biztonsági szolgáltatásuk van, sőt bizonyos feladatok elvégzését meg is tilthatják a programozó számára. Megjegyzendő, hogy az Apache Tomcat jelenleg nem implementálja a Java EE minden részét, de az általa nyújtott szolgáltatáscsomag sok esetben elegendő lehet.

Az ügyféltől érkező szervletre vagy JSP oldalra vonatkozó kérést általában a webkiszolgáló fogadja és továbbítja a szervlet konténernek, mely azt lefuttatja, majd az eredményt visszaküldi a kiszolgálónak, ami továbbítja azt az ügyfélnek

A szervlet és a szervlet konténer kommunikációjához szükséges osztályokat és interfészeket tartalmazó Java Servlet API a `javax.servlet` és `javax.servlet.http` csomagokban van specifikálva, tehát a szervletekbe ezeket a csomagokat importálni fogjuk.

A szervlet kódja megírható Java nyelven, vagy amint a következő fejezetben látni fogjuk, generálható JavaServer Pages oldalból is. A webalkalmazás egyes részeit `.war` kiterjesztésű WAR fájlba szokták tömöríteni.

A kéréshez vagy válaszhoz hozzácsatolható egy speciális szoftverkomponens, melyet *szűrőnek* (filter) nevezünk, és melynek célja a kérés szervlet általi feldolgozás előtti, illetve a szervlet által generált válasz elküldés előtti feldolgozása. A szűrő használható például azonosításra, kódolásra, vagy tömörítésre.

4.2. GET és POST kérések és válaszok

Amikor egy ügyfél kapcsolódik egy kiszolgálóhoz és HTTP kérést küld, akkor azt jellemzően GET vagy POST módszerrel teszi meg.

A GET elvileg azt jelenti, hogy a küldő valamilyen információt kér, viszont a kérés részeként adatok is átadhatók, tehát információk küldésére is használható. Például, amikor a felhasználó beír egy URL-t és azt elküldi, akkor az GET módszerrel továbbítódik. De amikor az ügyfél egy űrlap adatait küldi el, akkor ezek elküldhetők a GET, illetve a POST módszerrel is. GET módszerrel történő küldés esetén a mezők adatai a kérés részeként, úgynevezett *lekérdező karakterláncban* (query string-ben) kerülnek elküldésre, melynek hosszát a böngészők különböző mértékben korlátozzák ([26]). Az elküldött lekérdező karakterlánc a felhasználó által a böngésző URL mezőjében megtekinthető.

A POST módszer elvileg valamilyen információ küldését jelenti. Ez esetben az adatok a HTTP kérés törzsében, közvetlenül a szoftverkapcsolaton keresztül küldődnek el, és így sokkal több (akár 2 gigabájtnyi) információt is küldhetünk. Például, ha egy űrlaphoz fájlt is csatolunk, akkor a fenti korlátozások miatt POST módszert kell használnunk. POST módszer alkalmazásakor az elküldött adatok nem láthatók az URL-ben, viszont bizonyos böngésző kiegészítőkkel megnézhetők. Titkos adatok küldéséhez mindenképpen HTTPS protokoll alkalmazása javasolt, hiszen az adatokat más felhasználóktól is védenünk kell.

4.3. Az első szervlet felépítése

Minden szervletnek meg kell valósítania a `javax.servlet.Servlet` interfészt, melyhez a HTTP szervlet a `javax.servlet.http.HTTPServlet` osztályt bővíti.

A szervleteknek nem kell tartalmazniuk `main` metódust. A kiszolgáló a kérés továbbításakor a Servlet interfész `service` metódusát hívja meg, melynek felülírása miatt HTTP szervlet esetén GET kéréskor annak `doGet`, POST esetén annak `doPost` metódusa hívódik meg, tehát a szervletben a várható kérés típusának megfelelőt kell megvalósítanunk. Hogyha a szervletben a kódot függetleníteni szeretnénk a kérés típusától, akkor megtehetjük, hogy a közös tevékenységet egy tetszőleges metódusban írjuk meg, melyet a `doGet` és `doPost` metódusokból is meghívunk.

A `service`, és ennek megfelelően a `doGet`, illetve `doPost` metódusok két paramétert vesznek át: egy `HttpServletRequest` típusú *kérési objektumot*, mely a kérésről tájékoztatja a szervletet és egy `HttpServletResponse` típusú *válaszobjektumot*, melyet a szervlet a válasz visszaküldéséhez használ.

HTML tartalom generálásánál a válaszobjektum `setContentType` metódusával a válasz típusaként a `text/html` típust adjuk meg, majd a `getWriter` metódusával létrehozott `PrintWriter` objektummal pedig kiíratjuk a HTML kódot.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class uw2 extends HttpServlet {
  protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
```

```
    out.println("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0
Strict//EN\"          \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd\">");
    out.println("<html
xmlns=\"http://www.w3.org/1999/xhtml\">");
    out.println("<head>");
    out.println("<link   rel=\"stylesheet\"   type=\"text/css\"
href=\"kozos.css\" />");
    out.println("<title>uw2</title>");
    out.println("<meta              http-equiv=\"Content-Type\"
content=\"text/html;charset=ISO-8859-2\" />");
    out.println("</head>");
    out.println("<body>");
    out.println("<p>Helló világ</p>");
    out.println("</body>");
    out.println("</html>");
    out.close();
  }
  protected     void     doGet(HttpServletRequest     request,
HttpServletResponse response)
    throws ServletException, IOException {
      processRequest(request, response);
  }
  protected     void     doPost(HttpServletRequest     request,
HttpServletResponse response)
    throws ServletException, IOException {
      processRequest(request, response);
  }
}
```

4.4. Az első szervlet futtatása

Mindenekelőtt telepítenünk kell egy webszervert, és ha adatbázist is szeretnénk használni, akkor egy adatbázisszervert is. A szervereket telepíthetjük külön, vagy telepíthetünk olyan programcsomagot, mely azokat más programmal együtt telepíti és konfigurálja. Apache Tomcat webszerver telepíthető például a NetBeans webfejlesztő környezettel együtt, vagy az XAMPP csomagban is, mely a webszerveren kívül MySQL adatbázisszervert és PHP-t is tartalmaz, azonban használatakor számos biztonsági probléma merül fel, ezért inkább kezdőknek, tanulás céljából ajánlott.

Ha telepítettük a webszervert, akkor futtatás céljából el kell helyeznünk a szervletet a megfelelő könyvtárban. Mivel egy webalkalmazás általában szervletek, JSP lapok, HTML dokumentumok, képek és egyéb erőforrások sokaságából áll, ezért a hordozhatóság érdekében a webalkalmazáson belüli fájlszerkezet szigorúan meghatározott ([25]).

Minden webalkalmazáson belül található egy `WEB-INF` könyvtár. A `WEB-INF/classes` alkönyvtárban a webalkalmazás szervletjeinek osztályfájljai és más osztályfájlok helyezhetők el. A `WEB-INF/lib` alkönyvtár a JAR fájlokba összecsomagolt osztályokat tartalmazza. A kiszolgáló osztálybetöltője ezeket a `classes` és `lib` alkönyvtárakat automatikusan átnézi az osztályok betöltésekor.

A Tomcat kiszolgáló webalkalmazásai alapértelmezésként a

kiszolgáló_gyökér/webapps/ könyvtárban helyezendők el, ahol a kiszolgáló_gyökér a Tomcat telepítési könyvtára. A Tomcat kiszolgáló alapértelmezés szerinti webalkalmazása a ROOT, mely tehát kiszolgáló_gyökér/webapps/ROOT/ könyvtárban található. Amíg kezdetben nem lesznek bonyolult webalkalmazásaink, addig egyszerűbb, ha tanulás céljára használandó szervletjeinket a
kiszolgáló_gyökér/webapps/ROOT/WEB-INF/classes
könyvtárba tesszük, ahol esetünkben a kiszolgáló_gyökér a Tomcat telepítési könyvtára.

A WEB-INF könyvtárban kell elhelyezni az alkalmazás *telepítésleíróját* (deployment descriptor). Ez egy web.xml nevű fájl, mely konfigurációs információkat tartalmaz arról a webalkalmazásról, amelyben található és melynek tartalmával fokozatosan fogunk megismerkedni. Első lépésként a szervletet ellátjuk egy névvel és megadjuk, hogy mely osztályfájl valósítja meg. Megjegyzendő, hogy a könnyebb átláthatóság érdekében most nem fogunk használni csomagokat, de nem default package használata esetén a csomagot is a szokásos módon meg kell jelölni (pl. [14]/3.1.3).

```
<servlet>
  <servlet-name> hello </servlet-name>
  <servlet-class> uw1 </servlet-class>
</servlet>
```

Ezek után az uw1.class fájlban levő szervletet kívülről hello néven lehet látni.

A futtatáshoz szükséges URL a kiszolgálótól függ.
A továbbiakban feltételezni fogjuk, hogy az olvasó tanulás céljából az ügyfélként használt saját gépére telepíti a szervert is. Ebben az esetben az URL lehet például:
http://localhost:8080/hello vagy
http://localhost:8080/servlet/hello.

Egy adott szervletre különböző URL mintákat is készíthetünk, melyekkel az meghívható lesz. Ez elrejtheti a szervletnek a webhelyen belüli kezelését és lehetővé teszi, hogy a szervlet az URL cím megváltoztatása nélkül lecserélje az oldalt. Például az előző szervletre legyen:

```
<servlet-mapping>
  <servlet-name> hello </servlet-name>
  <url-pattern> /hellocska </url-pattern>
</servlet-mapping>
```

Ha az uw1 az alapértelmezett (ROOT) webalkalmazáshoz tartozik, akkor az URL:
http://localhost:8080/hellocska.
H viszont az uw1 a kiszolgáló_gyökér/webapps/masik/
könyvtárban levő webalkalmazáshoz tartozna, akkor azt a
http://localhost:8080/masik/hellocska URL-el tudnánk elérni. De meg tudunk adni ennél bonyolultabb URL mintákat is.

```
<servlet-mapping>
  <servlet-name> hello </servlet-name>
  <url-pattern> /haliho/* </url-pattern>
</servlet-mapping>
```

vagy

```
<servlet-mapping>
  <servlet-name> hello </servlet-name>
  <url-pattern> *.haho </url-pattern>
```

```
</servlet-mapping>
```
Akkor a szervletet első esetben például a `http://localhost:8080/haliho/valami/barmi.html`, míg az utóbbi esetben a `http://localhost:8080/akarmi/barmi.haho` URL-ekkel is el tudjuk érni.

4.5. Szervletek működésének alapjai

4.5.1. A szervlet példányai

A szervlet konténerek többsége, a szervletek közötti hatékony adatmegosztás érdekében, az összes szervletet egyetlen Java virtuális gépen (JVM) futtatja. Kivételt képeznek azok csúcsteljesítményű a szervletek, amelyek támogatják a szervletek több háttérkiszolgálón történő megosztott végrehajtását.

Egy adott néven regisztrált szervlet betöltésekor létrejön annak egy példánya, mely kezelni fogja a további kéréseket is. Ha egy szervletet több néven is regisztrálunk, akkor minden regisztrált névhez létrejön a szervlet egy-egy példánya. Például

```
<servlet>
  <servlet-name> uw3d </servlet-name>
  <servlet-class> uw3d </servlet-class>
</servlet>
<servlet>
  <servlet-name> uw3dz </servlet-name>
  <servlet-class> uw3d </servlet-class>
</servlet>
```
A szervlet eléréséhez használt név határozza meg, hogy melyik példány dolgozza fel a kérést.

Az olyan információkat, melyeknek minden példányból elérhetőknek kell lenniük, statikus vagy osztályváltozóban kell tárolnunk. Például:

```
public class uw3d extends HttpServlet {
  static int osztalyValtozo = 10;
  public void doGet(){ ...
    osztalyValtozo--;
    out.println(osztalySzamlalo); ...
  }
}
```
Ebben az esetben az `osztalyValtozo` értéke a szervlet bármely névvel történő elérésekor eggyel csökkenni fog.

4.5.2. Szinkronizálási problémák

Amennyiben egy adott néven a szervletet több ügyfél is eléri, akkor a szervlet példány mindegyikhez létrehoz egy szálat, mely a kérések között is megmarad. A szálak egymás helyi (az-az metódusokban deklarált) változóihoz nem férnek hozza, de a metódusokon kívül deklarált nem helyi változókat közösen kezelik. Emiatt szinkronizálási problémák állhatnak elő. Például, ha a szervletre majdnem egyszerre két kérés fut be, akkor megtörténhet, hogy az első szál közös változóra vonatkozó két művelete között a közös változó értékét a második szál is módosítja.

A legegyszerűbb megoldás az lenne, ha csak helyi változókat használnánk, melyeket a többi szál nem érhet el. Mivel azonban a helyi változók nem maradnak meg a kérések között, ezért ez a megoldás ritkán lehetséges.

Egy meglehetősen durva megoldás az, hogyha előírjuk, hogy minden példányhoz csak egyetlen szál tartozzon. Ezt szervlet osztály definíciójában kell megadnunk a következőképpen:

```
public class uw3e extends HttpServlet
   implements SingleThreadModel {
      ...
}
```

Azonban a megosztott erőforrások használatával kapcsolatos szinkronizálási problémákat ez sem küszöböli ki, viszont nagyon lassíthatja a működést, ezért általában érdemes más megoldásokban gondolkodni.

Egy további lehetőség az úgynevezett *szinkronizált blokkok* létrehozása, mely azt jelenti, hogy elkülönítjük a kód egy részét és előírjuk, hogy ha azt egy szál elkezdte futtatni, akkor annak befejezéséig a többi szál nem futtathatja. Mivel ez idő alatt a többi szálnak várnia kell, ezért célszerű csak szükség esetén és minél kisebb méretű szinkronizált blokkokat használni.

Az egyik lehetőség az, hogy a kritikus kódrészletet tartalmazó teljes metódust szinkronizáltnak minősítjük. Például:

```
public void synchronized doGet(...){ ... }.
```

Ebben az esetben a szervlet egyidejűleg csak egyetlen GET kérést fogadhat.

Csökkenthetjük a szinkronizált blokk méretét az által, ha csak azokat a sorokat szinkronizáljuk, melyeknek az egy szál általi megszakíthatatlan végrehajtását biztosítani akarjuk ([9]). Például:

```
synchronized(this){
  osztalyValtozo--;
  out.println(osztalyValtozo);
}
```

Még tovább csökkenthetjük a szinkronizált blokk méretét, ha csak azt hagyjuk benne, aminek a párhuzamos végrehajtása valóban problémát okoz. Az előző példában a párhuzamos kiírás nem lenne probléma, ha meg tudnánk oldani, hogy az `osztalyValtozo` más szálak általi módosítása esetén is a kiírásnak jó érték adjon át. Ezt úgy tehetjük meg, hogy a közösen hozzáférhető változó saját szálon belüli módosításának eredményét a szinkronizált blokkon belül eltároljuk egy helyi (azaz metóduson belüli) változóban, melyet a másik szál nem módosíthat, majd azt íratjuk ki:

```
public void doGet(...){
  int helyi;
  synchronized(this){
    helyi = osztalyValtozo--;
  }
  out.println("A változó értéke: " + helyi);
}
```

A szinkronizálás azonban minden esetben terheli a szervert, ezért ha a szinkronizálás hiányának következményei nem súlyosak, akkor el is tekinthetünk a problémától.

Megjegyzendő, hogy számos keretrendszer létezik, melyek elfedik előlünk a szervleteket, és saját implementációban definiált osztályok létrehozását írják elő, melyek életciklusa eltér a szervletek életciklusától. Például a struts action-ök minden kéréshez külön osztálypéldányt hoznak létre, így a szervleteknél előforduló szinkronizálási probléma nem jön elő.

4.5.3. Szervletek betöltése és inicializálások

A `web.xml` fájlban, a `load-on-startup` taggal előírhatjuk, hogy a kiszolgáló indulásakor bizonyos szervletek automatikusan betöltődjenek és elkezdjék működésüket. Például:

```
<servlet>
  <servlet-name> uw2 </servlet-name>
  <servlet-class> uw2 </servlet-class>
  <load-on-startup>10</load-on-startup>
</servlet>
```

A `<load-on-startup>` és `</load-on-startup>` közötti értékekkel szabályozhatjuk, hogy a szervletek milyen sorrendben töltődjenek be. A kisebb pozitív egész számmal jelölt szervletek előbb töltődnek be, mint a nagyobb számmal jelöltek. A negatív vagy nem egész számmal, illetve üres `<load-on-startup/>` elemmel jelölt szervletek betöltési ideje és sorrendje a kiszolgálótól függ, azaz bármikor betöltődhetnek.

A szervletben elhelyezhetünk egy `init` metódust, mely közvetlenül a szervlet példányának elkészítése után, a kérés feldolgozása előtt hívódik meg és alkalmas inicializáló tevékenységek elvégzésére. Inicializáló értékeket a `web.xml` fájlban, az `init-param` elemben is elhelyezhetünk. Az inicializálandó változó neve a `param-name`, értéke a `param-value` tagban kell legyen.

```
<servlet>
  <servlet-name> uw3e </servlet-name>
  <servlet-class> uw3e </servlet-class>
  <init-param>
    <param-name>kezdetip</param-name>
    <param-value>10</param-value>
    <description> A kezdet. </description>
  </init-param>
</servlet>
```

A kezdeti paraméterek értékeit tetszőleges helyen (például az `init` metódusban) beolvashatjuk a `getInitParameter` metódus segítségével. Az inicializáló paraméterek értékeit a `getInitParameterNames` metódussal lehet lekérdezni, mely egy `Enumeration` típusú objektumban adja meg ezek felsorolását.

```
public void init() throws ServletException {
  Enumeration parameterek = getInitParameterNames();
  while (parameterek.hasMoreElements()) {
    String nev = (String) parameterek.nextElement();
    String kezdetis = getInitParameter("kezdetip");
    try {
      int kezdetie = Integer.parseInt(kezdetis);
      out.println(nev + ": " + kezdetie);
    } catch (NumberFormatException e) { ... }
  }
  ...
}
```

Ha ugyanazokat az inicializáló paramétereket több szervlet is használni szeretné, akkor ezeket nem a szervletek, hanem a környezet paramétereiként kell megadnunk a `web.xml` fájlban:

```
<context-param>
```

```
  <param-name> kparam </param-name>
  <param-value> 8888 </param-value>
</context-param>
```

Környezet alatt a szervletet tartalmazó webalkalmazást értjük. A szervletben ez egy `ServletContext` típusú környezeti objektum lesz, melyet a `getServletContext()` metódussal tudunk lekérdezni és melynek ismeretében a környezeti paramétereket a fentiekhez hasonlóan ezen objektumra meghívható `String ServletContext.getInitParameter(String name)` és `Enumeration ServletContext.getInitParameterNames()` metódusokkal tudunk beolvasni.

Ha a szervletben egy `destroy` metódust is elhelyezünk, akkor az a szervlettel kapcsolatos összes kérés feldolgozása után fog meghívódni, illetve akkor, ha a kérések túlléptek egy bizonyos időkorlátot. A `destroy` metódusban például fájlba menthetjük a szervlet állapotát, melyet a szervlet ismételt betöltése esetén vissza tudunk olvasni.

4.5.4. Környezeti attribútumok

A szervlet környezeti attribútumként tetszőleges objektumot el tud tárolni, melyhez más szervletek is hozzáférhetnek, tehát ez a módszer szervletek közötti információk megosztására is alkalmas. A környezeti attribútumokkal kapcsolatos metódusok a környezeti objektumra hívhatók meg. Környezeti attribútum és hozzá tartozó objektum felvétele a `setAttribute(String nev, Obejct o)` metódussal lehetséges. Ha már létezik adott nevű attribútum, akkor értéke felülíródik. A név a Java-ban megszokott csomagnevek szabályainak kell megfeleljen. Környezeti attribútumokat egyébként nem csak a szervlet helyezhet el, hanem léteznek meghatározott nevű kiszolgáló-specifikus attribútumok is. Az összes attribútum nevét az `Enumeration` típusú objektumot visszaadó `getAttributeNames()` metódussal tudjuk lekérdezni, egy adott nevű attribútum értékét pedig a `getAttribute(String nev)` metódus adja meg. Ha nem létezik adott nevű attribútum, akkor `null` értéket kapunk.

Például, ha információinkat ideiglenes fájlba akarjuk írni, akkor azt az ideiglenes fájlok könyvtárába kell elhelyeznünk, melyet a `javax.servlet.context.tempdir` környezeti attribútum tartalmaz.

```
File dir = (File) getServletContext().
               getAttribute("javax.servlet.context.tempdir");
```

Tomcat szerver esetén az ideiglenes fájlok többnyire a `Tomcat\work\catalina\localhost\_` könyvtárban találhatók.

Visszatérve a környezeti attribútumokra, amennyiben az elosztott környezet miatt vagy biztonsági okokból a szervlet konténer nem tiltja meg, akkor másik környezet attribútumai is hasonlóképpen elérhetők lehetnek, viszont azokat a másik környezet objektumára kell meghívni. A másik környezet objektumát a saját környezeti objektumra meghívható `getContext(String uri)` metódussal kaphatjuk meg, ahol az `uri` a `/`-el kezdődő, a kiszolgáló gyökeréből induló abszolút útvonal.

Ha valamely környezeti attribútumra már nincs szükség, akkor a memória foglaltság csökkentése érdekében célszerű azt törölni a `removeAttribute(String nev)` metódussal.

Megjegyzendő, hogy a környezeti attribútumokat többnyire menetkövetésre szokták használni. A menetkövetéssel a 4.10. alfejezetben fogunk megismerkedni.

4.5.5. Kiszolgáló oldali adatok lekérdezése

A kiszolgáló nevét és a kérés portszámát a kérési objektum `getServerName()` és `getServerPort()` metódusaival tudjuk lekérdezni. A környezeti objektumra meghívható `getServerInfo()` metódus a kiszolgáló szoftverének nevét és verzióját adja meg egy `/` jellel elválasztva. A verzióra akkor lehet szükségünk, ha olyan feladatunk van, melyet régebbi verziókban másképp kellett megvalósítani. Ebben az esetben a környezet objektumra meghívható `getMajorVersion()` és `getMinorVersion()` metódusok megadják a szervlet API verziójának . előtti, illetve utáni értékét. A különböző verziók kezelésének másik lehetősége, hogy a képességérzékeléshez hasonlóan a verzió helyett bizonyos osztályok meglétét teszteljük.

4.5.6. Az ügyfél által küldött adatok lekérdezése

Az ügyfél által küldött adatok lekérdezésére alkalmas metódusok többsége a kérési objektumra hívható meg, ezért ebben a fejezetben ezt tekintjük alapértelmezettnek és a metódus osztályát csak akkor fogjuk jelezni, ha ettől eltérő.

Az ügyfél gépének nevét és IP címét a `getRemoteHost()`, illetve `getRemoteAddr()` metódusok adják meg. Azonban ezek egy proxy kiszolgálóhoz is tartozhatnak, így nem minden esetben alkalmasak a tényleges ügyfél azonosítására.

A böngésző a kéréssel együtt név-érték párokat is elküldhet. Erre láttunk példát az űrlapok elküldésénél. Az ilyen paraméterek neveit a `getParameterNames()`, egyetlen értékkel érkező adott nevű paraméter értékét a `getParameter(String nev)`, több értékkel érkező adott nevű paraméter értékét pedig a `getParameterValues(String nev)` metódussal kaphatjuk meg. Ha nincs adott nevű paraméter, akkor `null`-t, egyébként az utóbbi esetben az értékeket `String` objektumokként tartalmazó tömböt kapunk. A teljes lekérdező karakterláncot a `getQueryString()` metódussal kapjuk meg. Például, ha egy elküldött űrlap adatait szeretnénk feldolgozni, akkor a fenti metódusokat használhatjuk.

A kérés elküldéséhez használt módszert (például GET vagy POST) a `getMethod()` metódussal kaphatjuk meg. A `HttpServlet service()` metódusa is ezt használja a kérések megfelelő helyre való irányításához. Megtudhatjuk továbbá a kérés sémáját (például `http`, `https`, `ftp`) illetve protokolljának nevét és verzióját a `getScheme()` illetve `getProtocol()` metódusokkal.

Egy HTTP kérés a paraméterei mellett még elérési *útinformációkat* (vagy virtuális elérési utakat, [9]) is tartalmazhat, melyekkel valamely kiszolgálón található fájl elérési útját lehet megadni. Például az űrlap elküldésénél az `action` jellemző értékeként a szervlet specifikálása után `/` jelekkel elválasztva megadható egy fájl elérési útja: `action = "/szervlet/konyvtar/fajl"`. Ebben az esetben az URL a következőképpen nézhet ki: `http://kiszolgáló:port/szervlet/konyvtar/fajl`. Az útinformáció a `getPathInfo()`, míg az útinformációként megkapott fájl tényleges, fájlrendszeren belüli helye a `getPathTranslated()` metódussal kapható meg. Például, ha `action="../uw9/valami/barmi.txt"`, akkor a `getPathInfo()` visszatérési értéke `/valami/barmi.txt`, míg a `getPathTranslated()` metódusé `C:\Program Files\Apache Group\Tomcat 7.0\webapps\ROOT\valami\barmi.txt` lehet. Ha a virtuális elérési utat a szervlet nem útinformációként kapja meg, hanem valahonnan tudja, akkor a fájlrendszeren belüli tényleges helyét a környezeti objektumra meghívható `getRealPath(String` *virtualisUt*`)` metódus adja meg, mely tehát ugyanazt adja vissza, mint amit elérési útinformáció esetén a

`getPathTranslated()` adott volna.

Bizonyos esetekben szükség lehet fájlok tartalomtípusának lekérdezésére. Ezt a környezeti objektumra meghívható `getMimeType(String file)` metódussal tudjuk megkapni. Például a virtuális elérési útban megadott fájl tartalomtípusának lekérdezése `String tipus = getServletContext().getMimeType(req.getPathTranslated());` lesz. Bár a kiszolgálók többnyire ismerik a fájlkiterjesztések és MIME típusok megfeleltetéseit, ezeket a `web.xml`-ben bővíthetjük vagy módosíthatjuk a következőképpen:

```
<mime-mapping>
   <extension> java </extension>
   <mime-type> text/plain <mime-type>
</mime-mapping>.
```

Az ügyfél által használt URL a `getRequestURL()` metódussal kérdezhető le. A kérés URI-jét, melyet normál HTTP szervleteknél úgy kapunk, hogy az URL-ből töröljük az URL sémát, a gépnevet, portszámot és a lekérdező karakterláncot, a `getRequestURI()` metódus adja vissza. Az URI-nek a szervletre vonatkozó részét is külön megkaphatjuk a `getServletPath()` segítségével.

A szervletünket tartalmazó webalkalmazás URI előtagja (a kiinduló könyvtárának elérési útja) a `getContextPath()` metódussal kérdezhető le, mely segítségével a hordozhatóság érdekében elkerülhetjük a környezeti út explicit beírását. Ily módon a `getRequestURI()` által visszaadott (és szükség esetén URL dekódolt) érték megkapható `getContextPath()`, `getServletPath()` és `getPathInfo()` visszatérési értékeinek (szükség esetén dekódolás utáni) összefűzéseként.

4.5.7. Kérési fejlécek

Az ügyfél a kérés részeként úgynevezett *fejléc információkat* is küld. Jelen fejezetben csak a legfontosabbakat fogjuk megvizsgálni azok közül, melyeket a szervlet felhasználhat.

Az ügyfél az általa elfogadott médiatípusokat típus/altípus formában, vesszővel elválasztva, az `Accept` fejléc értékeként küldi el. A szervlet az ügyfél által elfogadott nyelveket fontossági sorrendben, ISO-639 szabványos rövidítésekkel, vagy ISO-3166-os országkódokkal, az `Accept-Language` fejléc értékeként kapja meg. A `User-Agent` fejléc az ügyfél szoftveréről szolgáltat információkat, így általában tartalmazza többek között az operációs rendszert és a böngésző adatait (nevét, verzióját). A `Referer` fejléc értéke annak a dokumentumnak az URL-je, amelyik a jelenlegire mutató hivatkozást tartalmazta. Az `IF-Modified-Since` fejléc megadja az ismételten betölteni kívánt weboldal utolsó módosítási időpontját (amit utoljára az oldallal együtt a kiszolgálótól kapott).

A fejlécek értékének olvasása a kérési objektumra meghívható metódusokkal lehetséges. Tetszőleges fejléc értékét a `getHeader(String fejlecnev)` metódussal kapjuk meg, mely a `fejlecnev`-ben nem tesz különbséget kis- és nagybetűk között, a megfelelő fejléc hiányában pedig `null`-t ad vissza. Ha a fejlécnek több értéke is lehet (például az ügyfél több nyelvet is elfogad), akkor a `getHeaders(String fejlecnev)` metódust kell használni. A szervlet által elérhető összes fejléc nevét `getHeaderNames()` metódus adja meg egy `Enumeration` objektumban.

Dátumot tartalmazó fejlécek lekérdezésére használható a `getDateHeader(String name)` metódus, mely a dátumot a `Date` típusnak megfelelően 1970. jan. 1. éjfél GMT óta. eltelt napok számában, `long` értékként adja vissza. Segítségével lekérdezhető például az előbbiekben ismertetett `IF-Modified-Since` fejléc értéke.

Fejléc hiányában -1-et ad vissza, továbbá ha a fejléc értéke nem alakítható dátummá, akkor `IllegalArgumentException` típusú kivételt dob.

Egész értéket tartalmazó fejlécek olvasására alkalmas a `getIntHeader(String name)` metódus, mely a fejléc értékét `int` értékként adja vissza, annak hiányában -1-et kapunk, ha pedig nem alakítható egész értékre, akkor `NumberFormatException` kivételt dob.

4.6. Információk küldése

Egy HTTP szervlet küldhet az ügyfélnek egy állapotkódot, tetszőleges http fejléceket, illetve a válasz törzsét.

4.6.1. Állapotkódok

Az *állapotkód* egy szám, ami jelezheti azt, hogy a kérést sikerült-e teljesíteni, sikertelenség esetén annak okát, illetve azt, hogy az ügyfélnek további lépéseket kell-e tennie a kérés befejezéséhez. A számot kísérheti egy szöveges magyarázat is.

Hogyha egy szervlet nem állítja be az állapotkódot, akkor a kiszolgáló a kérést teljesítettnek minősíti és a 200 `OK` kódot és üzenetet állítja be.

A szervletben az állapotkód a válaszobjektum `setStatus(int` *allapotKod*`)` metódusával állítható be, ahol *allapotKod* a HttpServletResponse interfészben definiált `static final int` típusú konstans lehet ([27]). A válasz elküldése után az állapotkód beállítása hatástalan.

Az alábbiakban felsorolunk néhány gyakrabban használt állapotkódot.
Mindegyik esetén megadjuk annak numerikus értékét, a konstans nevét, jelentésének angol nevét és az értelmezését.

- 200, `SC_OK`, OK: alapértelmezett állapotkód, a kérés teljesítése sikeres volt;
- 204, `SC_NO_CONTENT`, No Content: a kérés teljesítése sikeres volt, de nincs visszaküldendő tartalom;
- 301, `SC_MOVED_PERMANENTLY`, Moved Permanently: a kért erőforrás tartósan át lett helyezve a `Location` fejléc által megadott helyre, ezért a későbbi hivatkozásoknak az új címet kell használniuk; a böngészők többsége automatikusan elvégzi az átirányítást;
- 302, `C_MOVED_TEMPORARILY`, Moved Temporarily: a kért erőforrás ideiglenes új helyre lett áthelyezve, de a későbbi hivatkozásoknak az eredeti helyet kell használniuk; az új helyet ez esetben is a `Location` fejléc tartalmazza, ahová a böngészők többsége automatikusan átirányítja a felhasználót;
- 401, `SC_UNAUTHORIZED`, Unauthorized: megfelelő hitelesítés hiánya miatt a kliensnek nincs joga hozzáférni a kért erőforráshoz; a `WWW-Authenticate` és `Authorization` fejlécekkel együtt szokták használni;
- 404, `SC_NOT_FOUND`, Not Found: a kért erőforrás nem található vagy nem áll rendelkezésre;
- 500, `SC_INTERNAL_SERVER_ERROR`, Internal Server Error: váratlan hiba (pl. kezeletlen kivétel) lépett fel a kiszolgálón, ami megakadályozza a kérés teljesítését; például a szervlet a karakterkódolásnak nem megfelelő karaktereket használ;
- 501, `SC_NOT_IMPLEMENTED`, Not Implemented: a kiszolgáló nem támogatja a kérés teljesítéséhez szükséges funkciót;

- 503, `SC_SERVICE_UNAVAILABLE`, Service Unavailable: a szolgáltatás (kiszolgáló) *ideiglenesen* nem érhető el; a kiszolgáló a `Retry-After` fejlécben tájékoztatást adhat arról, hogy mikor lesz elérhető.

4.6.2. HTTP Fejlécek

Az ügyfél által küldött fejlécekkel már foglalkoztunk, most azt fogjuk áttekinteni, hogy a kiszolgáló hogyan küldhet fejléceket.

Először lássuk a szervlet által küldhető legfontosabb fejléceket ([63]).

- `Cache-Control`: azt határozza meg, hogy a dokumentum mennyi ideig legyen gyorsítótárolva; `no-cache` érték esetén nem kell, `no-store` érték esetén kifejezetten tilos gyorsítótárolni (például a tartalom titkossága miatt); `max-age-seconds` értékeként megadható, hogy mennyi idő múlva legyen elavultnak tekintve a tartalom;
- `Connection`: azt adja meg, hogy kész-e kiszolgáló tartós kapcsolat létrehozására az ügyféllel; `keep-alive` érték esetén igen, `close` érték esetén nem; a tartós kapcsolatokat később fogjuk tárgyalni;
- `Retry-After`: a kiszolgáló ideiglenes elérhetetlensége esetén másodpercek számában vagy dátumot tartalmazó karaktersorozatban megadja azt az időt, amikor a kiszolgáló kezelni fogja tudni a kéréseket; az `SC_SERVICE_UNAVAILABLE` állapotkóddal együtt használjuk;
- `Expires`: megadja azt az időt, ameddig a dokumentum valószínűleg nem változik meg, vagy amikor a dokumentumban levő információk érvénytelenné válnak;
- `Last-Modified`: az oldal utolsó módosításának ideje
- `Location`: a dokumentum áthelyezése esetén annak új URL-je;
- `Refresh`: használata előírja az aktuális vagy specifikált oldal bizonyos idő elteltével történő (ismételt) betöltését;
- `WWW-Authenticate`: megadja azt a jogosultságvizsgálati sémát és jogosultsági kört, amelybe az ügyfélnek tartoznia kell ahhoz, hozzáférhessen az elkért URL-hez; az `SC_UNAUTHORIZED` állapotkóddal együtt használjuk;
- `Content-Type`: a tartalom típusa;
- `Content-Encoding`: megadja a választörzs kódolását; többféle kódolást ,-vel elválasztva lehet megadni, abban a sorrendben, amelyben az adatokat kódoltuk;
- `Content-Length`: a tartalom hossza.

A HTTP protokoll szerint az állapotkódot és a fejlécet a válasz törzse előtt kell elküldeni.

A fejléceket statikus XHTML kódban meta objektumokként lehet megadni

```
<meta http-equiv="fejlec" content = "ertek" />
```

formában. Szervletek esetében azonban a fejlécek beállítására a válaszobjektumra meghívható metódusok használhatók. A `setHeader(String fejlec, String ertek)` metódus beállítja a `fejlec` fejléc értékét `ertek`-re, ha pedig már volt értéke az adott fejlécnek, akkor azt felülírja. Többértékű fejlécek beállítására az `addHeader(String fejlec, String ertek)` metódust használjuk, mely nem írja felül az adott nevű fejléc régi értékét, hanem az új értéket is hozzáadja. Dátum értékkel rendelkező fejlécek beállítására a `setDateHeader(String fejlec, long datum)` és `addDateHeader(String fejlec, long datum)` metódusokat használjuk, ahol létező fejléc esetén az előbbi az értéket felülírja, míg az utóbbi az új értéket hozzáadja. Egész értékű fejlécek esetén hasonlóan használhatók a `setIntHeader(String fejlec, int ertek)` és

`addIntHeader(String fejlec, int ertek)` metódusok. Annak lekérdezése, hogy adott nevű fejléc be lett-e állítva a `containHeader(String fejlec)` fejléccel lehetséges, mely beállított fejléc esetén `true`-t, egyébként `false`-ot ad vissza.

A fejlécek megfelelő használatával gyorsíthatjuk is az oldalak megjelenítését. Például, ha a felhasználó ismételten ugyanazt a szervletet hívja meg, akkor kiszolgáló oldalon ellenőrizhető, hogy szükséges-e az oldal ismételt legenerálása, vagy a böngésző megjelenítheti azt a gyorsítótárából. Ehhez a szervletnek gondoskodnia kell a generálás idejének a `Last-Modified` fejlécben való beállításáról oly módon, hogy felüldefiniálja a `HttpServlet` osztály `public long getLastModified(HttpServletRequest req)` metódusát, és az időt megadja annak visszatérési értékeként. A böngészők többsége visszaküldi ezt a weboldal újbóli betöltésekor az `IF-Modified-Since` fejléc értékeként, melyet a kiszolgáló összehasonlít a `getLastModified()` ismételt meghívásának visszatérési értékével. Ha az utóbbi értéke nagyobb, akkor meghívja a `doGet()` metódust és elküldi az új tartalmat, ha viszont nem, akkor nem hívja meg a `doGet()` metódust, hanem a 304-es `Not Modified` állapotkódot küldi el, amiből a böngésző tudni fogja, hogy a gyorsítótárából kell megjelenítenie az oldalt.

Az is előfordulhat, hogy az ügyfél nem is kéri tartalom legenerálását, hanem csak fejléc információkra kíváncsi. Ebben az esetben nem GET vagy POST, hanem HEAD típusú kérést küld. Viszont nincs `doHead()` metódus, ezért a `HttpServlet` osztály `service()` metódusa most is a `doGet()`-et fogja meghívni, de egy módosított `HttpServletResponse` objektummal, melynek a törzsében levő kimenetét figyelmen kívül hagyja. Vannak olyan fejléc információk, melyek meghatározásához szükséges a tartalom legenerálása (például `Content-Length`), de a fejlécek többségében erre nincs szükség. A felesleges tartalom legenerálását elkerülendő, a `doGet()` metódusban ellenőrizni lehet a kérés típusát a kérési objektumra meghívható `getMethod()`-al, és megfelelő HEAD kérés esetén elegendő csak a fejlécet beállítani.

4.6.3. Választörzs küldése

Mindenekelőtt be kell állítanunk a küldendő tartalom típusát, azaz a `Content-Type` fejlécet a

```
public void ServletResponse.setContentType(String type)
```

metódus segítségével.

Karakteralapú kiírást a

```
public PrintWriter ServletResponse.getWriter() throws
IOException
```

metódussal tudunk megvalósítani, mely a karaktereket a tartalomtípusban megadott karakterkészlet szerint kódolja. Nem támogatott kódolás esetén `UnsupportedEncodingException` kivételt kapunk.

Adatok bájtonkénti, kódolás nélküli kiírását a `public ServletOutputStream ServletResponse.getOutputStream() throws IOException` metódussal tudjuk megvalósítani. A `ServletOutputStream` a `javax.servlet` csomagban van definiálva, és a `java.io.OutputStream` közvetlen alosztályaként rendelkezésünkre bocsátja az `OutputStream` osztály `write()`, `flush()`, `close()`, továbbá a saját `print()` és `println()` metódusait, melyeknek `boolean`, `char`, `douuble`, `float`, `int`, `long` és `String` típusú paramétereket adhatunk meg. A `getWriter()` illetve `getOutputStream()` használata esetén `IllegalStateException` kivételt kapunk, ha ugyanarra a válaszra előzőleg már meghívódott a másik metódus.

Hogyha a választörzs bármely része elküldésre került, akkor a válasz *végrehajtottnak* minősül és az állapotkód és a fejlécek nem változtathatók meg.

A válaszok elküldése *tartós kapcsolatok* (életben tartott kapcsolatok, persistent connections, keep-alive connections, [64]) létrehozásával optimalizálható, feltéve, hogy a kiszolgáló támogatja ezt a lehetőséget. Amikor egy ügyfél (pl. böngésző) elkér valamilyen tartalmat a kiszolgálótól, akkor létrehoz egy *szoftvercsatornának* nevezett kapcsolatot a kiszolgálóval, melyen keresztül elküldi az adatait, és megkapja a választ. Ha az elküldött oldal olyan objektumokat tartalmaz, amelyek miatt a felhasználó további elemeket (például képeket) kell elkérjen a kiszolgálótól, akkor minden ilyen elemre újabb szoftvercsatorna épül ki, ami időbe kerül, tehát lassítja a kapcsolatot. Azt szeretnénk, hogy ha egyszer már kiépült a kapcsolat, akkor azon mindent elküldjünk, és ezt nevezzük *tartós kapcsolatnak*. Tartós kapcsolat esetén az ügyfél és a kiszolgáló megegyeznek, hogy hogyan közlik egyértelműen egymással a válaszuk végét. Erre egy lehetőség, hogy a tartalom előtt elküldik annak hosszát. Statikus fájlok esetén a kiszolgáló beírja a válasz `Content-Length` fejlécébe a tartalom hosszát, szervletek esetén viszont ezt a szervletnek kell beállítani a válaszobjektumra meghívható `setContentLength(int` *hossz*`)` metódus segítségével, ahol *hossz* az elküldendő bájtok száma. Mivel fejléc információról van szó, ezért ezt a törzs elküldése előtt kell meghívni. Azonban a törzs legenerálása előtt általában nem ismerjük annak hosszát. Ezt a látszólagos ellentmondást úgy oldjuk fel, hogy a legenerált tartalmat pufferelјük.

A válasz *pufferelése* lehetővé teszi, hogy a szervlet adatokat írjon a kimenetébe úgy, hogy azok ne kerüljenek azonnal elküldésre. Így az elküldés előtt a szervlet még változtathat az állapotkódon és fejlécen. A szervlet előírhatja, hogy a kiszolgáló pufferelje-e a választ és mekkora legyen a puffer mérete. Ha a teljes választörzs belefér a pufferbe és a kiszolgáló támogatja a tartós kapcsolatokat, akkor a kiszolgáló beállítja a válasz `Content-Length` fejlécét. Azonban az adatok pufferelése egyrészt több memóriát igényel, másrészt, amíg azt nem küldjük el, addig az ügyfélnek várakoznia kell, ezért túlságosan nagy mennyiségű tartalom pufferelése nem ajánlott. A szervletben a puffer kezelésére használható metódusokat a válaszobjektumra kell meghívni. A puffer bájtokban számolt méretét a `setBufferSize(int meret)` metódussal tudjuk előírni, azonban a kiszolgáló a kértnél nagyobb puffert is beállíthat. Ezután a szervletben ugyanúgy generáljuk le és küldjük el a tartalmat, mintha nem is lenne pufferelés, csak a kiszolgáló mindaddig nem küldi tovább, amíg a puffer nem telik meg, vagy a szervlet kifejezetten nem kéri a tartalom továbbítását. A `setBufferSize` metódust bármilyen tartalom elküldése előtt kell meghívni, ha ezt mégse így tennénk, akkor `IllegalStateException` kivételt kapunk. Arról, hogy elküldésre került-e a válasz bármely része, az `isComitted()` metódussal lehet megbizonyosodni, `true` válasz esetén már nem lehet megváltoztatni az állapotkódot és a fejléceket. A puffer méretét a `getBufferSize()` metódussal lehet megtudni, a `reset()` metódussal pedig törölni lehet a puffert, az állapotkódot és a fejléceket, természetesen a tartalom elküldése előtt. A válasz azonnali elküldését az állapotkóddal és fejlécekkel együtt a `flushBuffer()` metódussal írhatjuk elő.

4.6.4. Sütik használata

A kiszolgáló ideiglenes tárolás céljából sütiket is küldhet a böngészőnek. Amikor a böngésző újra eléri a kiszolgáló adott oldalát, akkor az annak megfelelő sütit visszaküldi ([8], [9]).

Minden sütinek van egy neve, hozzátartozó értéke, és további opcionális adatai. Sütik kezeléséhez a Servlet API `javax.servlet.http.Cookie` osztályára van szükség. Először létre kell hoznunk egy `Cookie` objektumot a publikus `Cookie(String nev,`

`String ertek)` metódussal, majd hozzá kell adjuk a válaszobjektumhoz annak `addCookie(Cookie suti)` metódusával. Például:

```
Cookie suti = new Cookie("elso", "123");
res.addCookie(suti);
```

A válaszobjektumhoz több süti is hozzáadható, de van néhány korlátozás, amire figyelnünk kell. Általában oldalanként legfeljebb 20, felhasználónként legfeljebb 300 sütit helyezhetünk el, és egy süti mérete nem haladhatja meg a 4096 bájtot. A sütik a HTTP fejlécben kerülnek elküldésre, ezért még a válasz végrehajtása előtt el kell helyezni őket.

A böngésző által visszaküldött sütiket a kérési objektum `getCookies()` metódusával kapjuk meg egy tömbben. Az egyes `Cookie` objektumoknak a `getName()`, illetve `getValue()` metódusaival tudjuk lekérdezni a süti nevét, illetve értékét. A `setValue(int ertek)` metódussal a süti értéke módosítható.

A sütire vonatkozóan további tulajdonságokat is beállíthatunk. Ezekhez `Cookie` típusú objektumokra meghívható metódusokat használunk, így a metódusok osztályának minden esetben történő feltűntetésétől eltekintünk.

Alapértelmezés szerint a böngésző a sütit csak annak a szervernek küldi vissza, amelyik azt beállította. Ha azt szeretnénk, hogy egy címtartományon belül a sütit bármelyik szerver megkapja (például azért, mert a szerverek megosztják a feladatokat és nem lehet előre tudni, hogy ténylegesen melyik szerver kell feldolgozza az oldalt), akkor a címtartományt a már elkészített süti objektumra meghívható `setDomain(String minta)` metódus segítségével állíthatjuk be. Például: `suti.setDomain(„.u-szeged.hu”)`. Megjegyzendő, hogy egyes böngészők speciális szabályokat alkalmazhatnak a sütik kezelésére vonatkozóan. Például az Internet Explorer biztonsági okokból a két betűs domain nevekhez nem enged sütiket elhelyezni ([28, 29]).

Beállíthatjuk a süti élettartalmát is a `setMaxAge(int meddig)` metódussal, melynek paramétereként azt kell megadni, hogy a süti hány másodperc múlva váljon törölhetővé. A 0 érték azt jelenti, hogy azonnal, a negatív érték pedig azt, hogy a böngészőből való kilépéskor törlődjön, ez utóbbi az alapértelmezés is.

Beállítható továbbá a süti verziója is a `setVersion(int v)` metódussal.

Alapértelmezés szerint a süti elküldése lehetséges nem biztonságos protokollokkal, de a `setSecure(boolean bizt)` metódus paraméterének `true` értékével előírható, hogy a süti csak biztonságos protokollon (például `https`) keresztül legyen elküldhető.

A fenti metódusoknak van lekérdező (`set` helyett `get`) alakja is, de ezeket ritkán szokták használni, mert a kiszolgálóra visszaérkező süti általában csak a nevét, értékét és verzióját tartalmazza. Ugyanakkor, ha a süti bármely tulajdonságát módosítjuk, akkor ezen három tulajdonságon kívül a többi tulajdonságot ismételten be kell állítani.

4.7. Szervletek közötti együttműködés

Az előző fejezetekben láttuk, hogy a szervletek információikat megoszthatják környezeti attribútumok vagy külső információtárak (pl. adatbázis) használatával. Az együttműködés azonban úgy is megvalósulhat, hogy a vezérlést átadják egymásnak ([9]). Ebben az esetben az információ átadás lekérdező karakterlánc segítségével is lehetséges.

4.7.1. Átirányítás

A vezérlés átadásának legegyszerűbb megvalósítása a kérés átirányítása egy másik szervlethez. Ezt használhatjuk például akkor, ha a dokumentum áthelyezésre került vagy terheléskiegyenlítés céljából, ha az URL több gép között osztja szét a terhelését. Lényegében azt

kell tennünk, hogy beállítjuk az állapotkódot `SC_MOVED_TEMPORARILY`-ra, a `Location` fejlécet pedig az új helyre. Alkalmazható viszont a válaszobjektum `sendRedirect(hely)` metódusa is, mely mindezt megteszi, de a helyet relatív URL-ként is elfogadja, továbbá elküld egy rövid választörzset is arra az esetre, ha az ügyfél nem észlelné az `SC_MOVED_TEMPORARILY` állapotkódot (tehát mi ne generáljunk választörzset) és kiüríti a puffert, de megtartja az előzőleg beállított fejléceket. Ily módon a vezérlést egy másik kiszolgálón levő szervletnek is átadhatjuk. Az átirányítást megvalósíthatjuk a `Refresh` fejléc beállításával is, amikor azt is megadhatjuk, hogy a másik oldal mennyi időn belül töltődjön be (és az alatt a felhasználó el tudja olvasni az általunk kiírt üzenetet). Például: `setHeader("Refresh","4; http:// www.valami.hu");`.
Abban az esetben is átirányítást kell használnunk, ha számlálni szeretnénk az oldalainkon levő linkek látogatottságát. Ehhez a linkeket úgy alakítjuk ki, hogy valójában egy saját szervletünkre mutassanak, és lekérdező karakterláncként tartalmazzák a másik oldal címét.
Például `<a href="/szervlet?hely=http://www.masikoldal.hu"> http://www.masikoldal.hu </a>`. A meghívott szervletben `getParameter("hely");`-el le tudjuk kérdezni a hivatkozott oldal címét, azt eltároljuk, majd átirányítjuk a felhasználót a kért oldalra. Természetesen, ha a céloldal URL-je eleve tartalmaz lekérdező karakterláncot, akkor az átirányítás előtt a saját részünket töröljük belőle.

4.7.2. Kéréselosztó objektum igénylése

Szervletek közötti vezérlés átadására a kiszolgálón belül további lehetőségek is vannak. Ehhez mindenekelőtt igényelni kell egy kéréselosztó objektumot a kérési objektumra meghívható `getRequestDispatcher(String eleresi_ut)` metódussal, melyben az `eleresi_ut` relatív is lehet, de a szervlet környezetéből kívülre nem mutathat. Ha az `eleresi_ut` `/`-el kezdődik, akkor az aktuális környezet gyökeréhez képest értendő. Kéréselosztó objektumot a `ServletContext` osztály `getNamedDispatcher(String nev)` metódusával is kérhetünk, ebben az esetben elérési út helyett az erőforrást a telepítésleíróban megadott névvel specifikáljuk, így nyilvánosan nem elérhető erőforrásokhoz is továbbítható a kérés. Mindkét metódus `RequestDispatcher` típusú kéréselosztó objektumot ad vissza, melyről a következő alfejezetekben látni fogjuk, hogy hogyan használható a kérések továbbadására.

4.7.3. Továbbítás

Amikor a kérést végleg át akarjuk adni azonos kiszolgáló másik erőforrásának, akkor ugyanazon a kezelő szálon belül a kéréselosztó objektumra meghívhatjuk a `forward(ServletRequest request, ServletResponse response)` metódust, ahol `request` és `response` a saját szervletünk által kapott kérési- és válaszobjektumok kell legyenek. A továbbító szervlet beállíthatja a fejléceket és az állapotkódot, de választ nem küldhet az ügyfélnek, különben `IllegalStateException` kivételt kapunk. Továbbításkor a válasz puffere kiürül. Ha a kéréselosztó elérési úttal van megadva, akkor az útinformáció átíródik a címzettnek megfelelően, illetve lekérdező karakterlánc is fűzhető a kéréselosztó útvonalhoz. Név alapján történő továbbításnál nincs ilyen lehetőség, viszont biztonságosabb, mert kevésbé nyilvánvaló a cél elérési útja, ugyanakkor figyelni kell arra, hogy a relatív linkek rosszul működhetnek. Másik környezetben levő erőforráshoz annak `getContext()`-el történő lekérdezése segítségével továbbíthatjuk a kérést. A `sendRedirect()`-től eltérően a `forward()` teljes egészében a kiszolgálón belül működik, ezért a továbbítás az ügyfél számára láthatatlan, ugyanakkor a fentiek miatt a

`sendRedirect()` használata sokkal egyszerűbb.

4.7.4. Beillesztés

Amennyiben a vezérlést nem akarjuk teljes egészében átadni egy másik erőforrásnak, csak annak kimenetét akarjuk beilleszteni az általunk generált tartalomba, akkor a kérési objektumra ugyanazon a kezelő szálon belül meghívható `include(ServletRequest request, ServletResponse response)` metódust használjuk, ahol a `forward()`-hoz hasonlóan a `request` és `response` a saját szervletünk által kapott kérési- és válaszobjektumok kell legyenek. A beillesztett szervlet a tartalmat az oldal bármely részébe beszúrhatja, továbbá nem tudja megváltoztathatni a válaszban elküldendő állapotkódot illetve fejléceket. Attól függően, hogy a hívó szervlet a kiíráshoz `PrintWriter`-t vagy `OutputStream`-et használ, a beillesztett erőforrásnak is azt kell használni, különben `IllegalStateException`-t kapunk, ezért a szöveges kiírásokra általában ajánlott egységesen a `PrintWriter` használata. Az `include()` használatakor, a `forward()`-tól eltérően, az útinformációk nem íródnak át a címzettnek megfelelően, tehát ha azokat a beillesztett szervletben a szokásos módon kérdezzük le, akkor is a beágyazó szervlet útinformációit kapjuk. A beillesztett szervlet a saját útinformációi az alábbi kiszolgálóhoz rendelt kérési attribútumokban vannak eltárolva:
`javax.servlet.include.request_uri`,
`javax.servlet.include.context_path`,
`javax.servlet.include.servlet_path`,
`javax.servlet.include.path_info`,
`javax.servlet.include.query_string`,
Ezek lekérdezése a kérési objektumra meghívható `getAttribute(String attributum_neve)` metódussal lehetséges. Az összes kiszolgálóhoz rendelt attribútum nevét a kérési objektumra meghívható `getAttributeNames()` metódussal kaphatjuk meg.

Például legyen egy `uw22a.java` szervlet, mely `include()`-al beágyazza az `uw22b.java` szervletet:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class uw22a extends HttpServlet {
  public void doGet(HttpServletRequest req,
HttpServletResponse res) throws ServletException, IOException
  {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    out.println("Az include() elotti kiiras");
    req.setAttribute("elso", "uw22aAttr");
    String ut = "/uw22b?a=3&b=4";
    RequestDispatcher eloszto = req.getRequestDispatcher(ut);
    eloszto.include(req, res);
    out.println("Az include() utani kiiras");
  }
}
```

Legyen az `uw22b.java` szervlet a következő:

```
import java.io.*;
```

```
import javax.servlet.*;
import javax.servlet.http.*;
public class uw22b extends HttpServlet {
  public void doGet(HttpServletRequest req,
HttpServletResponse res) throws ServletException, IOException
  {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    String attributum = (String) req.getAttribute("elso");
    out.println();
    out.println("Kezdodik az uw22b.");
    out.println("elso= " + attributum);
    out.println("a= " + req.getParameter("a"));
    out.println();
    out.println("Request URI: req.getRequestURI(): " +
req.getRequestURI());
    out.println("javax.servlet.include.request_uri: " +
req.getAttribute("javax.servlet.include.request_uri"));
    out.println("Context Path: req.getContextPath(): " +
req.getContextPath());
    out.println("javax.servlet.include.context_path: " +
req.getAttribute("javax.servlet.include.context_path"));
    out.println("Servlet Path: req.getServletPath(): " +
req.getServletPath());
    out.println("javax.servlet.include.servlet_path: " +
req.getAttribute("javax.servlet.include.servlet_path"));
    out.println("Path Info: req.getPathInfo(): " +
req.getPathInfo());
    out.println("javax.servlet.include.path_info: " +
req.getAttribute("javax.servlet.include.path_info"));
    out.println("Query String: req.getQueryString(): " +
req.getQueryString());
    out.println("javax.servlet.include.query_string: " +
req.getAttribute("javax.servlet.include.query_string"));
    out.println("Vege az uw22b-nek.");
    out.println();
  }
}
```

Az `uw22a` szervletet futtatva a következő kimenetet kapjuk:

```
Az include() elotti kiiras

Kezdodik az uw22b.
elso= uw22aAttr
a= 3

Request URI: req.getRequestURI(): /uw22a
javax.servlet.include.request_uri: /uw22b
Context Path: req.getContextPath():
javax.servlet.include.context_path:
```

```
Servlet Path: req.getServletPath(): /uw22a
javax.servlet.include.servlet_path: /uw22b
Path Info: req.getPathInfo(): null
javax.servlet.include.path_info: null
Query String: req.getQueryString(): null
javax.servlet.include.query_string: a=3&b=4
Vege az uw22b-nek.

Az include() utani kiiras
```

Az `uw22b`-ben levő `req.getRequestURI()`, `req.getContextPath()`, `req.getServletPath()`, `req.getPathInfo()`, `req.getQueryString()` hívások az `uw22a`-ra vonatkozó eredményt adják vissza. Hogyha az `uw22b` `include()` helyett `forward()`-al kapta volna meg a vezérlést, akkor ugyanezek a metódusok az `uw22b`-re vonatkozó értékeket adták volna, melyeket így a
`javax.servlet.include.request_uri`,
`javax.servlet.include.context_path`,
`javax.servlet.include.servlet_path`,
`javax.servlet.include.path_info`,
`javax.servlet.include.query_string` kiszolgálóhoz rendelt kérési attribútumokban kaptunk meg.

4.8. Hibakezelés

Ha a szervlet hibát akar jelezni, akkor a válasz végrehajtása előtt meghívhatja a válaszobjektum `sendError(int` *`allapotKod`*`)` vagy `sendError(int` *`allapotKod`*`, String` *`s`*`)` metódusát, melynek átadja a hibára jellemző állapotkódot és állapotüzenetet. Ebben az esetben a kiszolgáló legenerál és elküld egy hibát leíró oldalt. Az előzőleg beállított fejlécek változatlanok maradnak, azonban a hibalap generálása előtt a puffert kiüríti.

Ha a kiírandó tartalmat a szervletben szeretnénk legenerálni, akkor az állapotkódot a `setStatus()`-al kell beállítani ([9]). Ennek érdekében ajánlott idejében elvégezni a hibavizsgálatot és elkapni a kivételt. A hibaüzenet helyben történő kiíratásánál sokkal hatékonyabb a hibalapot külön szervletben legenerálni. Ebben az esetben a `web.xml` fájlban meg lehet adni, hogy a `sendError()`-al beállított hibakód esetén a vezérlés mely másik erőforrásnak adódjon át, mely csak környezeten belüli oldal lehet, de szükség esetén átirányíthat egy másik környezetbeli oldalra:

```
<error-page>
  <error-code> hibakód </error-code>
  <location> erőforrás </location>
</error-page>
```

Több `error-page` tagot is elhelyezhetünk. A erőforrásnak `/`-el kell kezdődnie, és szervlet vagy JSP esetén a `sendError()`-ban elhelyezett állapotkódot és üzenetet a
`javax.servlet.error.status_code`, illetve
`javax.servlet.error.message` kiszolgálóhoz rendelt kérési attribútumokból tudja lekérdezni.

Hiba esetén különösen hasznos lehet a történtek naplózása, mely a `GenericServlet` osztály `log(String` *`uzenet`*`)` vagy `log(String` *`uzenet`*`, Throwable` *`t`*`)` metódusá-

val végezhető el. A naplófájl formátuma és helye a kiszolgálótól függ, de többnyire tartalmazza a szervlet regisztrált nevét és az időpontot.

Természetesen nem elegendő kiíratni és naplózni a keletkezett kivételt, hanem azt kezelni is kell. `RuntimeException`, `IOException` és `ServletException` típusú kivételek kezelése átháríthető a kiszolgálóra, bár a kiszolgálók különbözősége miatt az se mindig ajánlott, minden más kivételt pedig egyértelműen nekünk kell kezelnünk.

A szervletekben előforduló hibák alaposztálya a `ServletException`, melynek `javax.servlet.servletException()` konstruktora meghívható opcionális üzenettel, illetve egy `Throwable` típusú objektummal, melynek segítségével bármilyen `k` kivétel továbbadható a `throw new servletException(k)` kivételdobással, így a gyökérhiba eljuthat a kiszolgálóig. Hogyha a szervlet nem érhető el (ideiglenesen vagy tartósan), akkor `UnavailableException` típusú hibát dobunk. A szervlet tartósan nem érhető el, ha a kivételt dobó példány nem tud kikeveredni a hibából. Ilyen jellegű kivétel dobása a `javax.servlet.UnavailableException(String uzenet)` konstruktorhívással lehetséges, melynek paraméterében le lehet írni a kivétel okát és más információkat. A szervlet ideiglenesen nem érhető el, ha az a teljes rendszert érintő valamilyen hiba miatt bizonyos ideig nem képes a kérések kezelésére. Ez alatt a kéréseket a kiszolgáló kezeli az `SC_SERVICE_UNAVAILABLE` (503-as) állapotkód és `Retry-After` fejléc elküldésével. Ebben az esetben az `UnavailableException` konstruktornak át kell adnunk az elérhetetlenség becsült időtartamát is másodpercekben, ha nem tudjuk, akkor pedig egy negatív számot. Az `isPermanent()` metódussal lekérdezhető, hogy tartós-e a kivétel, ha pedig ne, akkor `getUnavailableSeconds()`-al az is, hogy a szervlet mennyi idő múlva lesz elérhető. A `web.xml`-ben a kivételekre vonatkozóan is megadható, hogy adott kivételdobás esetén a vezérlés melyik másik erőforrásnak adódjon át.

```
<error-page>
  <exception-type> javax.servlet.ServletException
  </exception-type>
  <location> /servlet/HibaLap </location>
</error-page>
```

A hibát kezelő szervletben a `javax.servlet.error.exception_type`, `javax.servlet.error.exception`, `javax.servlet.error.message` kérési attribútumokból tudjuk kiolvasni a kivétel típusát, a kivételt tartalmazó `Throwable` típusú objektumot és a kivételhez tartozó szöveges üzenetet.

Hosszabb tartalom generálásakor megtörténhet, hogy a felhasználó nem várja meg, amíg azt elküldjük, hanem sokkal korábban már más oldalra távozik. Ez akkor is így van, ha több kisebb tartalmat generálunk, de azokat puffereljük. Ilyenkor sajnos nem keletkezik kivétel, ezért a szervlet csak akkor értesülhet erről, amikor elküldi a választ. A felesleges munka elkerülése érdekében érdemes a generálás közben időnként kiíratni egy szóközt és ellenőrizni, hogy sikerült-e a kiírás. Ha a kiírás nem sikerült, akkor `ServletOutputStream` használata esetén `IOException` kivétel keletkezik, melyet kezelhetünk vagy továbbadhatunk, de mindenképpen gondoskodnunk kell az erőforrások felszabadításáról és lezárásáról. `PrintWriter` használata esetén nem keletkezik kivétel, hanem a `PrintWriter` objektum `checkError()` metódusával a puffer tartalmának kiírásával együtt tudjuk ellenőrizni, hogy keletkezett-e hiba a kiírásnál (`true` esetén igen).

4.9. Hitelesítés

4.9.1. Alapszintű hitelesítés

A HTTP protokollba be van építve egy alapszintű hitelesítés ([9], [66]). A webkiszolgáló egy adatbázist tart fenn a felhasználókról, jelszavaikról, illetve a védendő erőforrásokról és minden védett erőforráshoz való hozzáférési kísérletkor bekéri a felhasználó nevét és jelszavát. A böngésző a felhasználónév és jelszó bekérése után másodszor hitelesítettként is elküldi a kérést és a kiszolgáló a hozzáférést csak akkor engedélyezi, ha megfelelő értékeket kapott. Tehát az alapszintű hitelesítést teljes egészében a kiszolgáló végzi. Megjegyzendő azonban, hogy ez csak egy alapfokú biztonságot ad, mert például az adatok továbbítása nem biztonságos Base64 kódolással történik.

Az alkalmazásszerver előre elkészített tartományokat (úgynevezett realm elemeket) tartalmaz, melyekhez csoportokat és felhasználókat az adminisztrációs felületén keresztül adhatunk hozzá. Az alábbi példában, mivel állomány alapú adatbázist fogunk használni, feltételezzük, hogy a felhasználókat a `file` nevű tartományhoz adtuk hozzá.

Meg kell adnunk a kiszolgálónak azt is, hogy mely alkalmazásokat kell védeni, így ezekből gyűjteményt képezünk, melyet egy névvel fogunk ellátni. A hozzáférések vezérléséhez szerepköröket (role) hozunk létre. Egy szerepkört több felhasználóhoz, illetve több gyűjteményhez is hozzárendelhetünk. A felhasználó azokhoz az alkalmazásokhoz férhet hozzá, amelyekhez valamely szerepköre tartozik. A felhasználók neveinek, jelszavainak és szerepköreinek nyilvántartása kiszolgálófüggő. A hitelesítést a `web.xml` fájlban konfiguráljuk.

```
<servlet> ... </servlet>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>
       GyujtemenyNeve
    </web-resource-name>
    <url-pattern> URLMinta_1 </url-pattern> ...
    <url-pattern> URLMinta_n </url-pattern>
    <http-method> GET </http-method> ...
    <http-method> POST </http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name> Szerepkor_1 </role-name> ...
    <role-name> Szerepkor_k </role-name>
</auth-constraint>
</security-constraint>
```

A tetszőleges számú URL mintát és URL metódust magába foglaló gyűjteményhez csak a megadott szerepkörökkel rendelkező felhasználók férhetnek hozzá. Ha egyetlen `<http-method>` tagot sem adunk meg, akkor az összes HTTP metódus védett.

A bejelentkezéssel kapcsolatosan meg kell adnunk a hitelesítési eljárást (melynek értéke alapszintű hitelesítésnél BASIC) és a bejelentkezéshez használandó tartományt:

```
<login-config>
  <auth-method> BASIC  </auth-method>
  <realm-name> file </realm-name>
</login-config>
```

Végül meg kell adnunk az alkalmazásban használható szerepköröket:

```
<security-role>
```

```
  <role-name> Szerepkör_1 </role-name> ...
 <role-name> Szerepkör_n </role-name>
</security-role>
```

A bejelentkezés után a szervletben a kérési objektumra meghívható `getUserPrincipal()` metódus egy olyan `java.security.principal` típusú objektumot ad vissza, mely egységbe zárja a felhasználó azonosságához tartozó adatokat és melyre meghívható `getName()` metódussal kaphatjuk meg a felhasználónevet. Azt, hogy a felhasználó adott szerepkörbe tartozik-e a kérési objektumra meghívható public boolean `isUserInRole(String` *szerep_neve*`)` metódus adja meg.

4.9.2. Űrlap alapú hitelesítés

Ez a módszer az alapszintű hitelesítéshez képest annyi kiegészítést nyújt, hogy a felhasználó adatait általunk elkészített oldalon kérjük be és hiba esetén egy saját hibaoldalt jelenítünk meg ([9]). Ehhez a `web.xml` fájlt a következőképpen konfiguráljuk:

```
<login-config>
  <auth-method> FORM </auth-method>
  <form-login-config>
    <form-login-page>
         /bejelentkezo.html
    </form-login-page>
    <form-error-page>
        /hibalap.html
    </form-error-page>
  </form-login-config>
<login-config>
```

Az oldalak a környezet gyökeréhez viszonyított abszolút hivatkozásokkal kell legyenek megadva. Az általunk készített bejelentkező oldal értelmezéséhez szükséges, hogy:

- a felhasználónév mezőjének neve `j_username` legyen,
- a jelszó mezőneve `j_password` legyen,
- a `method` értéke `post` legyen,
- az `action` értéke `j_security_check` legyen.

4.9.3. Egyedi hitelesítés

Megtehetjük azt is, hogy az alkalmazást a kiszolgáló szempontjából mindenki számára elérhetővé tesszük, és a szervletben gondoskodunk a felhasználók hitelesítéséről.

Ehhez készítünk egy bejelentkeztető oldalt, melyen bekérjük az azonosításhoz szükséges adatokat. A szervletben megvizsgáljuk, hogy a felhasználó megfelelően töltötte-e ki ezeket, és ha igen, akkor a felhasználóhoz rendelt menetobjektumban (melyről a menetkövetésnél lesz szó) létrehozunk egy bejegyzést, melyből később tudni fogjuk, hogy a felhasználó bejelentkezett. Végül megvizsgáljuk, hogy a menetobjektumban van-e olyan bejegyzés hogy a felhasználó eredetileg az alkalmazás egy másik védett oldalát szerette volna elérni, és (például `sendRedirect()`-el) átirányítjuk a felhasználót erre az oldalra vagy az alkalmazás főoldalára.

Hogyha a felhasználó egy bejelentkeztetőtől eltérő oldalhoz próbál hozzáférni, akkor megvizsgáljuk, hogy a menetobjektumban van-e előzetes bejelentkezést igazoló bejegyzés. Ha van, akkor megjelenítjük számára az oldal tartalmát, ha nincs, akkor átirányítjuk a bejelentkeztető oldalra.

4.9.4. HTTPS hitelesítés

Az eddigi bemutatott hitelesítések legnagyobb hátránya az információtovábbítás alacsony biztonsága. HTTPS (=HTTP+SSL) protokoll használatával lehetőség van sokkal biztonságosabb kommunikáció megvalósítására ([9], [66]). Ehhez a felhasználónak rendelkeznie kell egy nyilvános kulcsú bizonyítvánnyal.

A HTTPS ügyfélhitelesítés beállításához a hitelesítési módszert `CLIENT-CERT`-re kell állítani:

```
<login-config>
  <auth-method> CLIENT-CERT </auth-method>
  ...
<login-config>
```

Továbbá, a `<web-resource-collection>` elembe elhelyezendő `<user-data-constraint>` elembe beágyazott `<transport-guarantee>` elemmel be kell állítanunk a biztonsági szintet, melynek leggyakoribb és legbiztonságosabb értéke a `CONFIDENTIAL`:

```
<security-constraint>
   …
   <user-data-constraint>
     <transport-guarantee>
       CONFIDENTIAL
     </transport-guarantee>
   </user-data-constraint>
</security-constraint>
```

Hitelesítés után a felhasználó azonosságához tartozó adatokat a `getUserPrincipal()` metódus által visszaadott objektum tartalmazza.

4.10. Menetkövetés

A HTTP állapot nélküli protokoll, ezért nincs olyan beépített eljárás, amiből megállapítható lenne, hogy az egymást követő kérések ugyanattól a felhasználótól származnak-e. A webalkalmazásoknak viszont folyamatos kapcsolatban kell lenniük a felhasználókkal és a kérések között emlékezniük kell azok adataira és az addigi műveletekre. Az ügyfél azonosítása IP cím alapján nem lehetséges, mert sok esetben a visszaadott IP cím egy proxy kiszolgálóé, amely sok különböző ügyfelet kiszolgálhat. A megoldás az, hogy minden kérés előtt az ügyfél elküld magáról valamilyen információt, ami alapján azonosítani tudjuk. Jelen fejezetben feltételezni fogjuk, hogy a kéréseket egyetlen kiszolgáló elégíti ki. A menetkövetésre több módszer létezik, melyeket a [9] irodalom osztályozása szerint fogunk tárgyalni, az olvasó részletesebb információkat a [9], [67], [68], [69] irodalmakban talál.

4.10.1. Hitelesítés használata

A menetkövetés egyik lehetősége, hogy hitelesítjük a felhasználót, azt a böngésző megjegyzi, és mindig elküldi a felhasználó nevét, melyet a szervletben le tudunk kérdezni és annak megfelelő tartalmat tudunk generálni. A felhasználó különböző gépekről is küldheti a kéréseit, ha mindig bejelentkezik. Viszont ehhez regisztrálnia kell, és kijelentkeztetése is körülményes.

A felhasználó kényelme szempontjából (ami többnyire az oldal tulajdonosának üzleti érdeke is) fontos, hogy a látogatótól mindig csak a feltétlen szükséges erőfeszítéseket kérjük. Jelen esetben csak akkor indokolt a felhasználó hitelesítését kérni, amikor tényleg tudnunk kell a kilétét, például azért, hogy adott webáruházban kiválasztott termékeket postázhassuk

számára. Viszont olyan esetben, amikor egy generálásnál csak azt akarjuk tudni, hogy a szervletet hívó ügyfél járt-e már az webhelyünkön és milyen eltárolandó tevékenységeket végzett, de nem vagyunk kíváncsi a tényleges kilétére, akkor barátságosabb megoldásokat is használhatunk.

4.10.2. Rejtett űrlapmezők használata

Ez a módszer dinamikusan generált oldalak sorozatánál alkalmazható és az a lényege, hogy egy rejtett űrlapmezőben tetszőleges információt helyezünk el, melyet a következő szervlethíváskor kiolvasunk, és szükség szerint módosíthatunk.

Az egyik lehetőség, hogy a rejtett űrlapmezőbe az adott felhasználóval kapcsolatos addig felgyűlt összes adatot beírjuk, és ezt folyamatosan bővítjük. Azonban, ha lehetőség van az adatok szerveren történő eltárolására (például adatbázisban), akkor ennél hatékonyabb, ha minden felhasználóhoz rendelünk egy egyedi azonosítót, a rejtett űrlapmezőbe csak azt írjuk be, az összes többi hozzátartozó adatot pedig kiszolgáló oldalon tárolunk. Az ilyen azonosító csak arra az időre szól, amíg a felhasználót követni akarjuk és egyedi *menetazonosítónak* (session ID) nevezzük.

A menetazonosító eléggé bonyolult kell legyen ahhoz, hogy ha valaki hozzáfér a rejtett űrlapmezőhöz, akkor a menetazonosítóból ne tudja kitalálni egy másik ügyfél menetazonosítóját, mert akkor annak adhatja ki magát. Továbbá ne tegyünk nyilvánosan elérhetővé semmi olyan tartalmat (például naplófájlokat), melyek tartalmazhatják az aktív menetazonosítókat.

A kijelentkeztetés agyszerűen megoldható úgy, hogy a menetazonosítót nem generáljuk bele a további oldalakba, vagy töröljük a menetazonosító kapcsolatát a kiszolgálón eltárolt adatokkal.

4.10.3. URL újraírás

Azt feltételezve, hogy a felhasználó a webhelyünk oldalai között a linkek használatával közlekedik, a menetazonosítót továbbíthatjuk a hivatkozások segítségével is. A módszer lényege, hogy az oldal generálásakor minden hivatkozásba elhelyezzük az egyedi menetazonosítót vagy a szükséges információkat, a hivatkozás céljaként megadott szervlet pedig azt az URL-ből azokat kiolvassa.

Megvalósítás szempontjából az URL-ben elhelyezendő adatok megadhatók:

- a lekérdező karakterláncban paraméterként, például `http://valami?menetID = menetazonosito`,
- útinformációként, például `http://valami/menetazonosito`, vagy
- egyedi módon, melyet a kiszolgálók eltérő módon támogathatnak, például `http://valami;menetID = menetazonosito`.

Vigyázni kell arra, hogy paraméterek esetén névütközések léphetnek fel, az útinformáció pedig csak akkor használható, ha nincs szükség tényleges útinformációra.

A kijelentkeztetést itt is úgy tudjuk megvalósítani, hogy a további generálásokból kihagyjuk a menetazonosítót, illetve töröljük a menetazonosító kapcsolatát a kiszolgálón eltárolt adatokkal.

Az URL újraírás előnye a rejtett űrlapmezőkkel szemben, hogy nincs szükség űrlapok használatára.

4.10.4. Sütik használata

A menetazonosítót, esetleg más adatokkal együtt (például a felhasználó bizonyos beállításai), sütikben is tárolhatjuk. Figyelni kell azonban arra, hogy az egyes böngészők külön tárolják a

sütiket. A kijelentkeztetést itt meg tudjuk oldani oly módon, hogy töröljük a menetazonosítót tároló sütit és a menetazonosító kapcsolatát a kiszolgálón eltárolt adatokkal.

A módszer hátránya, hogy nem minden ügyfél engedélyezi a sütik fogadását, ezért ez nem mindig alkalmazható.

4.10.5. A menetkövetési API

A Servlet API tartalmaz kifejezetten a menetkövetés megvalósítására szolgáló metódusokat és osztályokat is ([70]). A Servlet API ezen részét fogjuk menetkövetési API-nak nevezni. A menetkövetési API-t a kiszolgálók támogatják, de eltérő mértékben. Például egyes kiszolgálók engedélyezik a menetobjektumoknak a kiszolgáló lemezére vagy adatbázisba írását, ha nincs elég memória, vagy a kiszolgáló leáll. Ehhez a menetobjektumoknak meg kell valósítaniuk a `Serializable` interfészt.

Most azt kérdezheti az olvasó, hogy ha vannak kifejezetten menetkövetésre tervezett osztályok, akkor miért tárgyaltunk az előzőekben számos más módszert? Nos, látni fogjuk a továbbiakban, hogy a menetkövetési API működése is az előző módszerekre épül, ezért annak megfelelő megértéséhez a fentiek ismerete is szükséges.

4.10.5.1. Menetek létrehozása és működése

Amikor egy felhasználó először hozzáfér egy webalkalmazáshoz, hozzárendelünk egy új `javax.servlet.http.HttpSession` típusú menetobjektumot és egy `String` típusú menetazonosítót, amely segítségével a későbbiekben is a felhasználóhoz tudjuk kötni a menetobjektumot.

Menetobjektum létrehozását illetve lekérdezését a kérési objektumra meghívható
`public HttpSession HttpServletRequest.getSession(boolean letrehoz)`
metódussal kezdeményezhetjük, mely hogyha a felhasználóhoz már tartozik menetobjektum, akkor azt adja vissza, ha nem, akkor a `letrehoz` paraméter `true` értéke esetén létrehoz egy újat, `false` értéke esetén `null`-t ad vissza. A metódusnak létezik paraméter nélküli változata is, ez úgy működik, mintha a `letrehoz`-nak `true`-t adtunk volna meg. A menetobjektum megfelelő karbantartása érdekében a `getSession()`-t a válasz végrehajtása előtt legalább egyszer meg kell hívni.

A menetobjektumhoz további objektumokat köthetünk, melyek a menet élettartalma alatt megjegyzendő információk eltárolására szolgálnak (például egy bevásárlókocsi tartalma vagy adatbázis kapcsolat). Ezeket az objektumokat a menetobjektumra meghívható `setAttribute(String nev, Object objektum)` metódus segítségével adott névvel kötjük a menetobjektumhoz, később pedig a menetobjektumra meghívható `getAttribute(String nev)` metódussal tudjuk megkapni. A menetobjektumhoz társított objektumokhoz tartozó neveket később a
`public Enumeration HttpSession.getAttributeNames()`
metódussal tudhatjuk meg. Az menetobjektumhoz rendelt összes objektum kinyerése a következőképpen lehetséges:

```
Enumeration enum = menet.getAttributeNames();
while (enum.hasMoreElements()) {
   String nev = (String) enum.nextElement();
   obj = menet.getAttribute(nev); ...
}
```

Ha egy hozzárendelt objektumot törölni szeretnénk, akkor ezt a menetobjektumra meghívható `removeAttribute(String nev)` metódussal tehetjük meg. Ha a törlendő objektum nem létezik, akkor semmi sem történik.

Bizonyos esetekben szükség lehet arra, hogy a menethez kapcsolt (például adatbázissal kapcsolatos feladatokat ellátó) objektum bizonyos tevékenységeket végezzen akkor, amikor létrejön vagy megszűnik a kapcsolat közte és a menetobjektum között. Ennek érdekében az objektum osztályának meg kell valósítania a
`javax.servlet.http.HttpSessionBindingListener`
interfészt, ezért meg kell valósítani annak
`public void HttpSessionBindingListener.valueBound(HttpSessionBindingEvent esemeny)`, illetve
`public void HttpSessionBindingListener.valueUnBound(HttpSessionBindingEvent esemeny)`
metódusait. Az előbbi akkor fog meghívódni, amikor az objektum hozzákapcsolódik egy menetobjektumhoz, az utóbbi akkor, amikor megszűnik ez a kapcsolat. Mivel a fenti metódusok minden menetkötéskor meghívódnak, ezért azokban a
`public HttpSession HttpSessionBindingEvent.getSession()`
metódussal lekérdezhető a menetobjektum, a
`public String HttpSessionBindingEvent.getName()`
metódussal pedig az, hogy az objektumunk milyen névvel kapcsolódott a menetobjektumhoz.

A menetobjektum létrehozásával generálódik egy egyedi menetazonosító is. Egyes kiszolgálók a felhasználóhoz az összes alkalmazásra vonatkozóan egyetlen menetazonosítót rendelnek, mások biztonsági okokból alkalmazásonként különböző menetazonosítót használnak. Az adott menethez rendelt menetazonosítót a menetobjektumra meghívható `getID()` metódus adja vissza. Ha a menetobjektum érvénytelen, akkor minden menetobjektumra meghívható előző metódus `IllegalStateException` kivételt dob.

Hogyan történik ténylegesen a menetkövetés? Amennyiben a felhasználó engedélyezi a sütiket, akkor a menetazonosítót egy ügyfélnél levő `JSESSIONID` nevű sütiben tárolja, ha viszont nem, akkor az újraírt URL részeként, a `jsessionid` egyedi elérési út paraméterrel tárolható, például a következő formában:
`http://www.valami.hu/barmi/akarmi;jsessionid=246?param=4`,
de léteznek más (például biztonságos SSL) megvalósítások is. Az újraírás szükségességét és módját a kiszolgáló határozza meg. Az URL-ek újraírásának tényleges megvalósítása viszont a szervlet dolga, a menetkövetési API ebben csak segítséget tud nyújtani a válaszobjektumra meghívható következő metódusokkal. Az `encodeURL(String url)`, amennyiben az újraírás szükséges és támogatott, akkor a paraméterében megadott URL-hez hozzáfűzi a menetazonosítót, egyébként azt nem módosítja. Akkor egy link generálása a következőképpen nézhet ki:

```
out.println("Ez egy <a href=\"" + res.encodeURL("/szervletB")
+ "\">hivatkozás</a>");
```

Ha a szervlet a saját magára mutató hivatkozást írja át, akkor `"/szervletB"` helyett `req.getRequestURL` kell. Átirányítás esetén azonban a `sendRedirect()` metódusban megadott URL-t az `encodeRedirectURL(String url)` metódussal kell újraírni, ugyanis ez esetben az URL újraírásra vonatkozóan a kiszolgáló más szabályokat alkalmazhat. Például: `res.sendRedirect(res.encodeURL("/szervletB"));`.
Az `encodeURL()`-t ne tévesszük össze a hasonló nevű

`java.net.URLEncoder.encode()` metódussal, mely a speciális karakterek kódolását végzi. Az `isRequestedSessionIdValid()` metódussal le lehet kérdezni, hogy a menetazonosító érvényes-e, továbbá az `isRequestedSessionIdFromCookie()`, illetve `isRequestedSessionIdFromURL()` metódusok akkor adnak true értéket, ha a menetazonosító sütiből illetve újraírt URL-ból származik.

4.10.5.2. Menetek érvényessége

A menet bármikor megszüntethető vagy beállítható, hogy adott idő múlva érvénytelenné váljon. Továbbá megszűnhet automatikusan is bizonyos tevékenységek következtében (például a kiszolgáló leállása esetén). A megszüntetett menetobjektum és a hozzá tartozó további adatok ilyenkor törlődnek, ezért a menet érvényességén túl megtartandó vagy fontos adatokat a menettől függetlenül is a felhasználóhoz köthetően el kell tárolnunk (például adatbázisban).

A kiszolgálónak van egy alapbeállítása az összes menet érvényességi idejére vonatkozóan, de beállíthatunk más értéket az alkalmazás összes menetére, vagy egy adott menetre vonatkozóan is. Az alkalmazás összes menetére vonatkozó érvényességi időt a `web.xml` telepítésleíróban lehet meghatározni egész számokkal megadott *percekben*, a `session-timeout` tag tartalmaként:

```
<session-config>
      <session-timeout> 45 </session-timeout>
</session-config>
```

Egy adott menet esetén ezt az értéket felülírhatjuk a menetobjektumra meghívható `setMaxInactiveInterval(int mp)` metódussal, melynek paramétereként az érvényességi időt viszont *másodpercekben* kell megadnunk. Negatív értékkel az adható meg, hogy a menet sohase járjon le. Egy adott menet érvényességi idejét, annak meghatározásától függetlenül, a menetobjektum `getMaxInactiveInterval()` metódusával tudjuk lekérdezni. Az érvényességi idő számolása az utolsó kérés elküldésekor indul, és újabb kérés küldése esetén elölről kezdődik.

Az érvényességi idő meghatározásánál figyelembe kell vennünk a felhasználó kényelmét és biztonságát, továbbá a szerver terheltségét.

A felhasználó kényelme érdekében minél hosszabb időt kellene megadnunk ahhoz, hogy közben más tevékenységeket is végezhessen. Például megtörténhet, hogy egy internetes vásárlás közben a felhasználó más oldalakon tájékozódni szeretne valamely termék paramétereiről, és hogyha eközben „kidobjuk” (azaz a menet lezárásával együtt kijelentkeztetjük), akkor örökre elveszíthetjük.

A szerver skálázhatóságának érdekében a lefoglalt objektumokat annál hamarabb fel kell szabadítanunk, minél nagyobb a szerver terheltsége. Viszont ha kiszolgáló oldalon olyan műveletek vannak, melyek növelik az ügyfél várakozási idejét (például lassú a keresés az adatbázisban), akkor az érvényességi idő megállapításánál ezt is figyelembe kell vennünk.

A felhasználó biztonsága azt kívánja, hogy ha elfelejtett kijelentkezni, akkor minél hamarabb érvénytelenítsük a menetet és jelentkeztessük ki azért, hogy ha otthagyja a gépet, akkor ezzel mások minél nehezebben tudjanak visszaélni. Például olyan biztonságos webalkalmazásoknál, mint az elektronikus banki ügyintézés, a szokásosnál rövidebb lejárati időt kell beállítanunk, melynek elteltével megkérdezzük a felhasználót, hogy akarja-e folytatni a tevékenységét, és hogyha nem, akkor kijelentkeztetjük. Ha be tudjuk tartatni az ügyfelekkel, hogy munkájuk végeztével jelentkezzenek ki, akkor biztonsági szempontból hosszabb érvényességi időt is engedélyezhetünk. A `setMaxInactiveInterval()` metódus segítségével azt is megtehetjük, hogy az egyes felhasználóknak különböző érvényességi időt

adunk, illetve szükség szerint azt menet közben is módosíthatjuk. Használhatjuk a menetobjektum `getCreationTime()` és `getLastAccessedTime()` metódusait is, melyek a menet létrehozásának, illetve az utolsó kérés elküldésének időpontját adják meg 1970. január 1. 0:00 GMT óta eltelt ezredmásodpercek számában. Szükség esetén a menetobjektum `invalidate()` metódusával a menetet azonnal érvényteleníthetjük és ez által a felhasználót kijelentkeztethetjük.

5. AJAX

5.1. Mi az AJAX?

Az AJAX (Asynchronous JavaScript and XML) ([71]) egy olyan kommunikációs módszer, mely a felhasználói élmény fokozása érdekében lehetővé teszi az ügyfél és kiszolgáló között adatok háttérben történő küldését és fogadását anélkül, hogy egy új oldalt töltene be. Ez a kommunikáció többféleképpen is megvalósulhat, a továbbiakban ilyen lehetőségeket fogunk áttekinteni. Látni fogjuk, hogy az AJAX számos megvalósítása már régi technológiákkal is lehetséges volt, azonban elterjedése az ügyfél oldali JavaScript motorok teljesítményének növekedésével vált lehetségessé ([72]). Jelen fejezetben természetesen csak az AJAX legfontosabb technikáit és használati lehetőségeit tudjuk bemutatni, további információkat az olvasó a fejezet alapvető forrásaként is szolgáló [11], [73], [74], [75] irodalmakban talál.

5.2. Rejtett keretek használata

Hagyományos módon, amikor egy kérést küldünk a kiszolgálónak és az visszaküld egy választ, akkor az oldalunk lecserélődik a választ tartalmazó oldallal. Hogyha azt szeretnénk, hogy a kérést küldő oldal tartalma továbbra is megmaradjon (sőt, esetleg ki is egészíthessük további tartalommal), akkor meg kell oldanunk, hogy a válasz máshova érkezzen, és annak ismeretében jelenítsük meg a látható oldalon a kívánt tartalmat.

A rejtett kereteket használó módszerek alapvető lépései következők:

1. rejtett keret létrehozása,
2. a kérés és a hozzá tartozó információk elküldése oly módon, hogy a válasz majd a láthatatlan keretbe érkezzen,
3. a kérés szerver oldali feldolgozása és a válasz elküldése,
4. a válasz rejtett keretbe történő megérkezésekor, annak esetleges feldolgozása után, a látható oldal tartalmának tetszőleges módosítása.

5.2.1. Rejtett keret létrehozása

Rejtett kereteket használó módszerek esetében a válasz egy másik keretbe vagy belső keretbe fog megérkezni. Természetesen a felhasználónak továbbra is csak egy oldalt kell látnia, ezért a választ tartalmazó keretet el fogjuk rejteni.

Megvalósítás szempontjából erre az egyik lehetőség, hogy definiálunk egy olyan keretrendszert, melyben az oldal két keretből áll, az egyik elfoglalja felületet, a másik pedig 0 méretű lesz:

```
<frameset rows="100%,0" style="border: 0px">
  <frame name="lathatoKeret" src="lathato.html"
         noresize="noresize" />
  <frame name="rejtettKeret" src="about:blank"
         noresize="noresize"/>
</frameset>
```

A rejtett keretbe eredetileg semmit sem töltünk be, ebbe fog majd érkezni a szerver válasza. Azért, hogy a felhasználó ehhez ne férjen hozzá, a keretek átméretezését is letiltjuk.

Egy másik lehetőség, hogy a felhasználó által látható oldalban egy rejtett belső keretet helyezünk el:

```
<iframe src="about:blank" name="rejtettKeret"
        style="display: none">
```

```
</iframe>
```

Az belső keretet dinamikusan JavaScripttel is létrehozhatjuk:

```
function iFrameKeszito() {
  var oIFE = document.createElement("iframe");
  oIFE.style.display = "none";
  oIFE.name = "rejtettKeret";
  oIFE.id = "rejtettKeret";
  document.body.appendChild(oIFE);
  frame = frames["rejtettKeret"];
}
```

A látható és a rejtett oldalakhoz is fog tartozni egy-egy szkriptfájl, melyek AJAX szempontjából az információk küldéséért, fogadásáért és megjelenítéséért fognak felelni.

Az érthetőség kedvéért a rejtett keretek működését egy olyan példán fogjuk bemutatni, melyben:

- a felhasználó által látható oldalon megjelenített fájl `lathato.html`, a hozzá tartozó szkriptfájl pedig `lathato.js` lesz,
- a `lathato.html` fog tartalmazni egy űrlapot, melyben lesz egy `eAdat` nevű mező, melynek tartalmát el fogjuk küldeni a kiszolgáló oldali alkalmazásnak,
- a látható oldalon lesz továbbá egy `eredmenyMezo` egyedi azonosítóval ellátott, eredetileg üres `div` objektum is, melyben a válasz megérkezése után azt meg fogjuk jeleníteni,
- a rejtett keretben levő tartalomhoz tartozó szkriptfájl `rejtett.js` lesz,
- a kiszolgáló oldali alkalmazás pedig `szervlet` néven lesz elérhető.

Általában persze nem csak egyetlen adatot küldünk el, és nem is feltétlen a kiszolgáló által visszaküldött választ akarjuk megjeleníteni, de a módszer megértése után a fentiek általánosítása egyszerűen megoldható.

5.2.2. A kérés és a hozzá tartozó információk elküldése

Külön kell választanunk azt a két esetet, amikor GET vagy POST módszerrel küldjük el az adatokat.

5.2.2.1. Információk küldése GET módszerrel

Mindenekelőtt azt kell megoldanunk, hogy a válasz majd a rejtett keretbe érkezzen. Ezért a `lathato.html`-ben levő űrlap elküldés gombjához a `lathato.js`-ben hozzárendelünk egy `elkuld()` eseménykezelőt, mely gondoskodni fog a kérés megfelelő elküldéséről.

Mit kell tennie az `elkuld()` függvénynek? Először is be kell azonosítania azt a keretet, ahová az eredményt kérni fogja. Statikus oldal esetében ez így fog kinézni: `frame = top.frames["rejtettKeret"]`, míg dinamikusan létrehozott belső keretnél már megtette ezt az `iFrameKeszito()` függvényben. Egyes böngészőknek szükségük van egy kis időre ahhoz, hogy a dinamikusan létrehozott belső keret megjelenjen, ezért az utóbbi esetben az `elkuld()` bizonyos késleltetéssel mindaddig újra meghívja önmagát, amíg a `frame` változó tényleges keretre nem fog mutatni:

```
if (!frame) {
   iFrameKeszito();
   setTimeout(elkuld, 10);
   return;
}.
```

Az `elkuld()` a továbbiakban kinyeri az elküldendő mező tartalmát:
```
var sAdat = document.getElementById("eAdat").value;
```
és elküldi a kérést, melyhez hozzáfűzi az elküldendő adatot:
```
frame.location =
    "http://localhost:8080/szervlet?adat=" + sAdat;
```

5.2.2.2. Információk küldése POST módszerrel

Ebben az esetben az URL-hez nem tudjuk hozzáfűzni az elküldendő adatokat, hanem az oldalt a szokásos módon kell elküldenünk, De akkor hogyan tudjuk megoldani, hogy a válasz a rejtett keretbe érkezzen?

Az egyik lehetőség, hogy az `form` tag `target` jellemző értékének megadjuk a rejtett keret nevét:
```
<form method="post"
      target="rejtettKeret"
      action="http://localhost:8080/szervlet">.
```
Ebben az esetben nem lesz szükség elküldés gombhoz társított eseménykezelőre sem.

Hogyha nem akarunk, vagy nem tudunk `target` jellemzőt használni, például azért, mert az űrlap `target` attribútumát nem minden böngészőben lehet dinamikusan létrehozott belső keretre állítani, akkor a megoldás bonyolultabb.

A `lathato.js`-ben legyen most is `elkuld()` a látható űrlap elküldéséhez társított eseménykezelő. Ebben szükség esetén létrehozzuk a dinamikus belső keretet, egyéb esetekben a láthatatlan keretet csak beazonosítjuk egy `frame` változóban. A láthatatlan keretbe most betöltünk egy olyan oldalt, mely kizárólag egy üres űrlapot tartalmaz:
```
setTimeout(function () {
  frame.location = "uresUrlap.html";}, 10);
}
```
Az `uresUrlap.html`-hez hozzá van rendelve egy `ures.js` szkript is. A cél az, hogy a rajtett keretben levő űrlapba átmásoljuk a látható oldalon levő elküldendő mezőket, és az eredeti űrlap helyett a rejtett keretben levő űrlapot küldjük el, így a válasz is majd oda fog érkezni. Ehhez az `ures.js` egyetlen eseménykezelőt fog tartalmazni, mely az `uresUrlap.html` betöltődésekor meghívja a `lathato.js`-ben levő másolást megvalósító `feltolt()` függvényt. A `feltolt()` azonban nem fog tényleges másolást végezni, hanem végigjárja a látható űrlap mezőit:
```
for (var i=0 ; i < urlap.elements.length; i++) {
  var mezo = urlap.elements[i];
 …
}
```
és annak minden elküldendő mezőjéhez létrehoz egy rejtett mezőt a láthatatlan keretben, az eredeti mezőnek megfelelő névvel és értékkel, majd elküldi a rejtett űrlapot a látható űrlap címzettjének:
```
rejtettUrlap.action = urlap.action;
rejtettUrlap.submit();.
```
Megjegyzendő, hogy a `feltolt()` függvény a látható űrlap `button`, `submit`, `reset` típusú mezőit, illetve a be nem jelölt `radio` és `checkbox` típusú mezőket figyelmen kívül hagyhatja, hiszen ezek nem tartalmaznak elküldendő értékeket.

5.2.3. A kérés szerver oldali feldolgozása és a válasz elküldése

A kiszolgáló oldali alkalmazás feldolgozza az átadott adatokat, legenerálja az eredményt tartalmazó oldalt, melyben elhelyez egy szkriptfájl csatolást a `rejtett.js` fájlra, és az eredményt visszaküldi, mely a kérésnek megfelelően a rejtett keretbe fog megérkezni.

5.2.4. Az eredmény megjelenítése

A `rejtett.js` fájlban a rejtett keretben megjelenő oldal betöltéséhez hozzárendelünk egy eseménykezelőt, mely a rejtett keretbe betöltött oldal előre eltervezett mezőiből kiolvassa az eredményt, esetlegesen azt feldolgozza, majd átadja a látható oldal megjelenítést megvalósító függvényének. Felosztó keretek esetén a látható oldalra
`top.frames["lathatoKeret"]`-el, belső keret esetén pedig `parent`-el hivatkozunk, tehát az átadás lehet

```
top.frames["lathatoKeret"].megjelenit(eredmeny);
```

vagy

```
parent.megjelenit(eredmeny);
```

Ehhez természetesen a `lathato.js` -ben kell lennie egy `megjelenit` függvénynek, mely megfelelő módon megjeleníti a kapott adatokat.

5.3. XMLHttp kérések használata

Ebben a módszerben a háttérbeli kommunikáció egy `XMLHttpRequest` objektum segítségével fog megvalósulni, melyet röviden `XHR` objektumnak fogunk nevezni.

Az `XMLHttp` kérést használó módszer alapvető lépései következők:

1. az `XHR` objektum létrehozása és inicializálása ügyfél oldalon,
2. a kérés és az ahhoz tartozó információk elküldése,
3. a kérés szerver oldali feldolgozása és a válasz elküldése,
4. a válasz megérkezésének észlelése és annak esetleges feldolgozása után a látható oldal tartalmának tetszőleges módosítása.

Ezeket a lépéseket természetesen akkor kell végrehajtanunk, amikor a kommunikációt meg szeretnénk valósítani, ezért annak megvalósítását megfelelő eseménykezelőbe helyezhetjük.

5.3.1. Az XHR objektum létrehozása és inicializálása

Az `XHR` objektum létrehozása a következőképpen lehetséges:

```
var oXHR = new XMLHttpRequest();
```

Az Internet Explorer 7-esnél korábbi verzióiban az `XHR` objektum ActiveX objektumként létezett, tehát ha azt akarjuk, hogy azokban is működjön, akkor
a verziónak megfelelő ActiveX objektumként kell létrehozni.

Lehetőségünk lesz a kérést a háttérben úgy elküldeni, hogy a program további végrehajtása megvárja, amíg a szervertől megérkezik a válasz, vagy oly módon, hogy az ügyfél oldali JavaScript kód végrehajtása a kérés feldolgozása alatt is folytatódjon. Az előbbi esetben a kérést *szinkronkérésnek*, utóbbi esetben azt *aszinkron kérésnek* nevezzük. Szinkronkérés esetén a válasz megérkezéséig a felhasználó nem fogja tudni folytatni a munkáját, ezért azt csak akkor használjuk, ha olyan gyors kommunikációt lehetővé tevő kis adatmennyiséget várunk a válaszban, amivel nem tesszük próbára a felhasználó türelmét, vagy kifejezetten azt szeretnénk, hogy a felhasználó a munka folytatása előtt megvárja a kiszolgáló oldali választ.

Az `XHR` objektumot létrehozás után inicializálni kell `oXHR.open(`*modszer*`,` *URL*`,` *async*`)` metódushívással, ahol:

- *modszer*: `"get"` vagy `"post"`,

- *URL*: string, ide küldjük a kérést,
- *async*: `true` vagy `false`; `true` esetén a kérés aszinkron.

Például:

```
oXHR.open("get", " http://localhost:8080/ XHRGetPl?adat=" +
sAdat, true);.
```

Az `open` metódus nem nyit meg semmilyen kapcsolatot, csak előkészíti az objektumot.

5.3.2. A kérés és az ahhoz tartozó információk elküldése

Amint a fenti példában is látható, `GET` kérés esetén a küldendő adatokat egyszerűen hozzáfűzhetjük az `URL`-hez, `POST` módszer használata esetén azonban erre nincs lehetőség, így nekünk kell összeállítanunk az elküldendő adatok neveit és értékeit tartalmazó stringet. Például egy űrlap esetén végignézzük a mezőket, kihagyjuk azokat, melyek nem tartalmaznak elküldendő értékeket, a többit kódoljuk az `encodeURIComponent()` metódussal, mezőnév=érték párokat képezünk belőlük, majd ezeket `&` jelekkel elválasztva összefűzzük egy `torzs` karaktersorozatba.

`POST` módszer használatakor a `Content-Type` fejlécet is beállítjuk a `setRequestHeader("`*fejléc_neve*`", "`*fejléc_értéke*`")` metódussal, mely hozzáadja az új fejlécet a többiekhez. Például:

```
oXHR.setRequestHeader("Content-Type",
   "application/x-www-form-urlencoded");.
```

A kérést ténylegesen a `send(torzs)` metódussal küldjük el. `GET` módszer esetén a `torzs` értéke `null`.

5.3.3. A kérés szerver oldali feldolgozása és a válasz elküldése

A kiszolgáló oldali programban nem feltétlen egy teljes (X)HTML oldalt állítunk elő, hanem megtehetjük, hogy csak a kérés válaszának szánt HTML töredéket "íratjuk ki", ezért a tartalom formátumát `text/plain`-re állítjuk: `res.setContentType("text/plain");`. A tartalom előállítása és kiíratása egyébként a szokásos módon történik.

5.3.4. A válasz megérkezésének észlelése és a kívánt tartalom megjelenítése

Az XHR objektum a `readyState` tulajdonságban egész számként tárolja a kérés feldolgozásának állapotát. Számunkra a 4-es állapot lesz fontos, mely a teljes válasz megérkezését és a kapcsolat lezárását jelenti. Aszinkron kérés esetén ahhoz, hogy a válasz megérkezéséről értesüljünk, ügyfél oldalon gyakran le kell kérdezni a feldolgozás állapotát, eseménykezelővel figyelni kell a `readyState` tulajdonság értékének minden változásakor kiváltódó `readystatechange` eseményt, és tesztelni kell a `readyState` tulajdonság értékét. Megjegyzendő, hogy ezért volt fontos a JavaScript motorok és a kliens számítógépek gyorsasága az AJAX elterjedéséhez.

Azonban a válasz akkor is megérkezettnek minősül, ha közben valamilyen hiba lépett fel (például a kért URL nem elérhető), tehát ez nem alkalmas a hiba tesztelésére. Viszont az XHR objektum `status` tulajdonsága tartalmazza a HTTP állapotkódot, így annak értékéből meg tudjuk állapítani, hogy történt-e valami hiba a kérés végrehajtása során. Az állapotkód szöveges leírását az XHR objektum a `statusText` tulajdonsága tartalmazza.

A válasz törzsét, illetve hiba esetén a hibaüzenetet az XHR objektum `responseText` tulajdonsága tartalmazza. Ha a visszaadott adatok típusa `text/xml`, akkor az XML dokumentum-objektumot a `responseXML` tulajdonság értékeként kapjuk meg.

A válasz megérkezésének tesztelése és annak megjelenítése tehát a következőképpen történhet ([11]):

```
oXHR.onreadystatechange = function () {
  if (oXHR.readyState == 4) {
    if (oXHR.status == 200 || oXHR.status == 304) {
      megjelenit(oXHR.responseText);
    } else {
      megjelenit("Hiba történt: " + oXHR.statusText);
    }
  }
};
```

Ily módon a rejtett keretekhez képest sokkal átláthatóbb kódunk lesz, hiszen a válasz nem feltétlen egy egész oldal és nincs hozzákapcsolt szkript kód sem.

Bizonyos esetekben, például az érkezett dokumentum típusának megállapítása érdekében, szükségünk lehet a válaszban kapott fejléc információkra is. A kapott fejlécek neveit az XHR objektum `getAllResponseHeaders()` metódusával tudjuk lekérdezni, mely egy olyan stringet ad vissza, melyben a fejlécek `\n` vagy `\r\n` karakterekkel vannak elválasztva egymástól, ezért a fejlécek neveit a következőképpen tudjuk kinyerni egy dinamikus tömbbe:

```
var sFejlecek = oXHR.getAllResponseHeaders();
var tFejlecek = sFejlecek.split(/\r?\n/);.
```

Adott nevű fejléc értékét a `getResponseHeader("fejlec_neve")` metódussal tudjuk megkapni.

Rejtett űrlapok, illetve XHR objektum használata esetén kizárólag ugyanazon tartomány erőforrásaihoz férünk hozzá, ezért más kiszolgálókhoz való hozzáféréshez a kérést és választ továbbító szerveroldali proxyra van szükség. A továbbiakban két olyan módszerrel ismerkedünk meg, melyek esetében más szervereken tárolt fájlokhoz is hozzáférhetünk.

5.4. Képek és sütik használata

A HTML oldalhoz tartozó JavaScript kódban `new Image()` vagy `document.createElement("img")` parancsok valamelyikével létrehozunk egy kép objektumot. A kép `src` tulajdonságának értékeként azt a szervletet adjuk meg, amelytől adatokat kérünk. Ez azt jelenti, hogy a képet a szervlettől várjuk. Bár maga a kép számunkra haszontalan, segítségével kommunikálni tudunk a szervlettel. Egyik irányban a szervlet URL-jében (például lekérdező karakterlánccal) adhatunk át értékeket. Másik irányban a szervlet válaszként információkat közölhet a szolgáltatott kép méretei által (`width` és `height`), illetve a képpel együtt küldött sütiben. A gyors válasz érdekében ajánlott kisméretű képeket használni. A válasz megérkezését a kép letöltődését vizsgáló `onload` eseménykezelővel tudjuk tesztelni. A módszer hátránya, hogy úgy az URL-ben küldött, mint a sütiben kapott információk mennyisége meglehetősen korlátos, továbbá az ügyfél gépén a képek, illetve a sütik használata le lehet tiltva.

5.5. Dinamikus szkriptbetöltés

A HTML oldalhoz csatolt szkriptben a kérés elindításakor `document.createElement("script")`-el létrehozunk egy szkript objektumot, melynek `src` tulajdonságának egy szervletet feleltetünk meg. Ily módon a létrehozott szkript kódja a szervlet kimenete lesz. Az `src` értékadásnál a szervlet URL-jében paramétereket is átad-

hatunk, persze a 4.2. fejezetben leírtak szerint korlátozott mennyiségben. Tipikusan az egyik átadott paraméter egy olyan függvény neve lesz (nevezzük a továbbiakban ezt *feldolgoz()*-nak), mely meg kell hívódjon akkor, amikor a válasz megérkezik és tetszőlegesen feldolgozhatja, megjelenítheti a kapott adatokat.

A szervlet meghívása és a szkript betöltése ténylegesen akkor kezdődik el, amikor a létrehozott szkriptet `document.body.appendChild(`*szkript*`);`-el hozzáadjuk az oldalhoz. A szervlet `setContentType("text/javascript");`-el beállítja a fejlécet és egy olyan JavaScript kódot küld vissza, mely meghívja a *feldolgoz()* függvényt, és ennek paraméterként átadja a visszaküldendő adatokat:

`out.println(`*feldolgoz*`+"(\""+adatok+"\")");`

A szervletben figyelni kell arra, hogy ha a visszaküldött kódban valamit karakterláncként kell értelmezni, akkor abban az `\` helyett `\\`, `"` helyett pedig `\"` kell legyen, de ehhez a kimenetre íratott karakterláncban `\\\\`, illetve `\\\"` kell legyen.

5.6. Az AJAX használata

Ebben a fejezetben néhány tipikus példát fogunk látni az AJAX használatára.

Olyan esetekben, amikor valószínűsíthető, hogy mik lesznek azok az információk, amelyeket a felhasználó az elkövetkezendőkben a kiszolgálótól kérni fog, ezeket a háttérben még az alatt letölthetjük, mialatt a felhasználó az aktuális tevékenységét végzi. Például egy több oldalas cikk olvasását a felhasználó nagy valószínűséggel a következő oldallal fogja folytatni, így azt a háttérben letöltjük. Amikor a felhasználó kérni fogja az adott információkat, akkor azokat a saját gépéről gyorsabban meg tudjuk majd jeleníteni.

Hasonlóképpen, az információ egyes részeit a háttérben már azelőtt elküldhetjük, mielőtt a felhasználó az egész elküldését kérné. Például, egy űrlap mezőinek kitöltése közben az egyes mezők tartalmának változásakor (`change` esemény) azokat azonnal elküldhetjük aszinkron módon, miközben a felhasználó folytathatja a többi mező kitöltését, és ha megérkezett az ellenőrzés eredménye és hibát mutat, akkor azt megjelenítjük. Hasonlóképpen megtehetjük azt is, hogy mezőbe történő gépelés közben is időközönként elküldjük a mező tartalmát (például a jelszó erősségének ellenőrzése érdekében). Természetesen sok esetben a mezők ellenőrzése kizárólag ügyfél oldalon maradva is megvalósítható. Továbbá, ha a felhasználó letiltja a JavaScriptet, akkor az ezt használó AJAX módszerek nem is fognak működni. Ezért, továbbá biztonsági okokból is, az űrlap végleges elküldésekor a mezőket szerver oldalon újból ellenőrizni kell.

Bizonyos alkalmazások esetén szükséges az oldal egyes részeit meghatározott időközönként frissíteni. Például ha egy levelezőprogramban új levél érkezett, akkor célszerű azt mihamarabb megjeleníteni. Hasonlóképpen, egy közösségi kommunikációs oldalon (chat, twitter stb.) folyamatosan meg kell jeleníteni az újabb üzeneteket. Az AJAX használata ez esetben azért hasznos, mert a háttérben meg tudja vizsgálni, hogy van-e megjelenítendő újabb információ, továbbá ha van ilyen, akkor sem kell az egész oldalt újratöltenünk, csak azt a mezőt, melyet az újabb információk megjelenítése érdekében frissíteni kell.

Utolsó példaként AJAX-al szabályozhatjuk az oldalon levő információk megjelenítési sorrendjét is. Ha a felhasználónak akarunk kedvezni, akkor először a felhasználó számára fontos és gyorsan letölthető tartalmat jelenítjük meg. Megtehetjük azonban azt is, hogy először reklámokat jelenítünk meg, ha úgy gondoljuk, hogy néhány felhasználó bosszúsága és esetleges elvesztése ellenére is ez nekünk megéri.

Másfelől az AJAX-ot népszerűsége ellenére sem mindig érdemes használni. Például, ha az oldal jelentős részét szeretnénk újratölteni, akkor az AJAX használata lényegesen több adat

mozgatását jelentheti a háttérben, mintha csak egyszerűen újratöltjük a teljes oldalt. A modern böngészők a teljes oldal újratöltésekor az azonos részeket vagy nem töltik le újra, vagy ha le is töltik, a felhasználó akkor sem fogja ezt észrevenni (nem fog villanni a kép).

Végül fontos megjegyezni, hogy számos olyan osztálykönyvtár létezik (például jQuery, Prototype), melyek egyéb szolgáltatásaik mellett az AJAX használatát is kényelmesebbé teszik. Viszont ezek használata előtt is mérlegelnünk kell, hogy megéri-e nagy mennyiségű JavaScript kód beemelése, mely lassítja az oldal (első) betöltését és konfliktusba kerülhet a saját kódunkkal.

6. JavaServer Pages

6.1. Alapfogalmak

Szervletek esetében a dinamikusan előállított kimenetet Java kódba ágyazott jelölőelemekkel lehet előállítani, ami nehezen átlátható kódot eredményezett.

A JavaServer Pages (JSP) ([76]) egy olyan szerver oldali technológia, mely lehetővé teszi, hogy Java nyelvű utasításokat statikus HTML oldalon el tudjunk helyezni, melyeket a kiszolgáló *az oldal letöltése* előtt végrehajt. A JSP kód beágyazása a statikus HTML kódba speciális jelölőelemek segítségével történik. Általában a JSP fájlok kiterjesztése `.jsp` és ugyanoda helyezzük el, ahova a szokásos HTML fájlokat tehetjük.

Amikor egy JSP oldalra először hivatkozunk, akkor a háttérben a kiszolgáló azt egy szervletté alakítja, illetve ha az már létezik, de régebben volt módosítva, mint a JSP oldal, akkor azt újrafordítja. Azonban a szervletté alakítás és fordítás időigényes feladat. Azért, hogy az első felhasználónak ne kelljen sokat várni, ajánlott már fejlesztőként egyszer meghívni az oldalt, vagy a webalkalmazás telepítésekor utasítani a konténert, hogy előre fordítsa le a JSP oldalakat. Ha a szervlet utolsó módosítási időpontja nem régebbi, mint a JSP oldalé, akkor a kérések a már létrehozott és lefordított szervletnek továbbítódnak.

A JSP elemek a HTML kódban a `<%` és `%>` jelek között vannak elhatárolva, ezért a HTML kódban, illetve a JSP elemek attribútumaiban ezek helyett a `<\%` és `%\>` karaktersorozatokat kell írni. A JSP oldalon kétféle megjegyzést helyezhetünk el. A `<%--` és `--%>` között megadható JSP megjegyzést a fordító figyelmen kívül hagyja és a kimenetbe sem kerül bele, míg a `<!--` és `-->` közötti HTML megjegyzés belekerül a kimenetbe, és ha JSP elemet tartalmaz, akkor az végrehajtódik.

JSP-ben a statikus HTML tartalmat ugyanúgy helyezzük el, mint a szokásos HTML fájlokban. A statikus HTML tartalom mellett azonban még háromféle elemet helyezhetünk el: szkriptelemeket, direktívákat és akcióelemeket.

6.2. Szkriptelemek

A szkriptelemek lehetővé teszik Java nyelvű programkód beillesztését a JSP oldalból generált szervletbe és háromfélék lehetnek: deklarációk, kifejezések, vagy szkriptletek.

Deklarációként `<%!` és `%>` jelek között megadhatunk változót vagy metódust, mely a generált szervletosztály törzsében (a `doXXX()`-en kívül) deklarálódik.

A `<%=` és `%>` között elhelyezhető kifejezés egy Java kifejezés vagy metódushívás lehet, melynek eredménye automatikus konverzióval vagy `toString()`-el karakterlánccá konvertálódva kerül be a válaszba. A kifejezés végén nem lehet pontosvessző (`;`) és kiértékelése lekéréskor történik, ezért hozzáfér a lekéréshez kapcsolódó információkhoz.

A szkriptlet `<%` és `%>` jelek között elhelyezett programkód, mely a generált szervlet doXXX() metódusába kerül bele, ezért nem tartalmazhat osztály vagy metódusdefiníciót. A szkriptletek keverhetők a statikus tartalommal. Például:

```
<% int x=1; int y = 2;
if(x*x < 2*y){ %> <p>kisebb</p>
<% } else { %> <p>nem kisebb</p>
<% } %>
```

amiből a szervlet doXXX() metódusában lényegében ez lesz:

```
int x=1; int y = 2;
```

```
if(x*x < 2*y){
  out.println("<p>kisebb</p>");
} else {
  out.println("<p>nem kisebb</p>");
}
```

6.3. Implicit objektumok

Vannak olyan objektumok, melyek minden JSP oldalon automatikusan definiálódnak és ezek között olyanok is, melyek kiváltanak bizonyos szervletekben használt objektumokat. Lássunk néhány ilyen fontosabb objektumot!

A `request` és `response` objektumokat úgy használhatjuk, mint a szervletekben definiált kérési illetve válaszobjektumokat. A kimeneti csatorna pufferelt, ezért a szervletektől eltérően a HTTP státuszkódok és fejlécek beállítása akkor is megengedett, ha már elküldtünk valamilyen tartalmat. A `session` objektum a kérelemhez kapcsolódó `HttpSession` objektum, mely automatikusan létrejön, hacsak a `page` direktíva `session` attribútumával ezt le nem tiltjuk (direktívákról később lesz szó). Végül, az `application` egy `ServletContext` típusú objektum, mely a szervleteknél használt `getServletContext()` által visszaadott referenciának felel meg, és melyben több webkomponens által elérendő adatokat tárolunk. Az adatokat írni a `setAttribute()`, olvasni pedig a `getAttribute()` metódusokkal lehet.

6.3. Direktívák

A direktívák konténernek küldött üzenetek, melyekkel a szervlet osztály struktúráját módosíthatjuk. Például a

`<%@ include file = "relatív URL" %>`

direktíva lehetővé teszi a JSP oldal szervletté alakításakor adott fájl tartalmának bemásolását a JSP oldalba, mely ezzel együtt fog szervletté alakulni. A

`<%@ page` *jellemző* `= "érték" %>`

direktívával a JSP oldalra érvényes jellemzők értékeit állíthatjuk be. Például az `import` jellemző értékeként megadhatjuk az oldalon használni kívánt csomagokat. A `contentType`, illetve `pageEncoding` jellemzők értékeként megadhatjuk a kimenet MIME típusát (alapértelmezett a `text/html`), illetve karakterkódolását. A `session` jellemző `false` értékével tilthatjuk le az alapértelmezett menetkövetést.

6.5. Akcióelemek

Az akcióelemek a szervlet motor viselkedését szabályozzák. A `jsp:include` akcióelem úgy működik, mint a szervletek `RequestDispatcher include()` metódusa, szintaxisa pedig:

```
<jsp:include page="relatív_URL | <%= kifejezés %>"
              flush ="true | false">
  <jsp:param name ="név_1" value ="érték_1 | <%= kifejezés_1 %>" />
  ...
  <jsp:param name ="név_n" value ="érték_n | <%= kifejezés_n %>" />
</jsp:include>
```

ahol *név_1*, …, *név_n* az URL-nek átadandó paraméterek, `flush=true` esetén pedig a konténer a másik alkalmazás kimenetének beszúrása előtt kiüríti a puffert. Hasonlóképpen, a `jsp:forward` akcióelem a szervletek `forward()` metódusának megfelelően működik, szintaxisa pedig annyiban tér el az előzőtől, hogy `jsp:include` helyett `jsp:forward`-ot írunk és nincs `flush` jellemző.

Akcióelemek JavaBean komponensek kezelésére is használhatók. A JavaBean komponens egy speciális feltételeket kielégítő osztály. Például, az osztályra meghívható kell legyen paraméter nélküli konstruktor, illetve ha van az osztálynak egy `x` nevű példányváltozója, akkor annak írására és olvasására léteznie kell `setX()`, illetve `getX()` nevű metódusoknak. Ezek a feltételek lehetővé teszik, hogy JSP jelölőelemek segítségével JavaBean objektumok kezelését előírhassuk. Egy ilyen akcióelem a `jsp:useBean`, melynek szintaxisa:

```
<jsp:useBean id = "nev" class = "csomag.osztaly"
             type = "tipus" scope = "hatókör" />
```

A *hatókör* lehet `page` (alapértelmezett), `request`, `session`, vagy `application`. Ezen értékeknek megfelelően olyan adott `id`-vel specifikált nevű JavaBean objektumot keres, vagy hiánya esetén létrehoz, mely csak az adott oldalra vonatkozó `PageContext` objektumban, a kérési objektumban, a menetobjektumban, illetve az alkalmazás többi oldala által is hozzáférhető `ServletContext` objektumban van tárolva. A talált objektumot `type` hiányában *csomag.osztaly* típusúként azonosítja, vagy hozza létre. A *tipus* csak a `class`-ban megadott osztály, annak ősosztálya vagy általa implementált interfész lehet. Az objektum attribútumainak a `jsp:setproperty` akcióelemmel adhatunk értéket, melynek szintaxisa:

```
<jsp:setProperty name = "név"
                 property = "attribútum_neve"
                 value = "érték | <%= kifejezés %>" />
```

vagy

```
<jsp:setProperty name = "név"
                 property = "attribútum_neve"
                 param = "paraméter_neve" />
```

Az utóbbi esetben az attribútum értéke a kérési objektum *paraméter_neve* nevű paraméter értékére lesz beállítva. Ha se `value`-t, se `param`-ot nem adunk meg, akkor az attribútumot annak nevével megegyező kérési paraméter értékére állítja be. Ha `property` értékének `"*"`-ot adunk meg, akkor minden paramétert beállít a kérési objektum azonos nevű paraméterének értékére, amennyiben az létezik. A `jsp:useBean` elem esetén záró tagot is használhatunk, ebben az esetben a `<jsp:useBean ...>` és `</jsp:useBean>` közöti utasítások csak a bean objektum létrehozásakor hajtódnak végre, ezért a csak inicializálásként elvégzendő beállításokat ily módon tudjuk beírni. A

```
<jsp:getProperty name = "név"
                 property = "attribútum_neve" />
```

elemmel lekérdezhetjük és beszúrhatjuk az oldalba az objektum adott attribútumának értékét.

A JSP-k használatát elemkönyvtárak segítik. Például a JavaServer Pages Standard Tag Library standard, gyakran használt elemek gyűjteményét tartalmazza. A felhasználók saját elemkönyvtárakat is képezhetnek a hasonló funkciót megvalósító, újrahasznosítható komponensekből.

6.6. Irodalom

Jelen fejezetben terjedelmi korlátok miatt csak a JSP alapjainak bemutatására volt lehetőség, részletesebb információkat az olvasó a fejezet forrásaként is szolgáló [10], [9], [14], [77], [78] irodalmakban talál. A fejezet tartalmán messze túlmutató, első sorban szerver oldali technológiák mélyebb elsajátítása a ([4]) jegyzet segítségével, illetve a hozzá tartozó kurzus keretében lehetséges.

7. Keretrendszerek

Webes alkalmazások fejlesztéséhez számos keretrendszert kidolgoztak. Akár egyetlen keretrendszer részletes tárgyalása is túlmutatna jelen jegyzet terjedelmi korlátain, ezért annak érdekében, hogy legalább a keretrendszerek legfontosabb jellegzetességeit szemléltetni tudjuk, ebben a fejezetben két különböző megközelítést használó keretrendszer alapjait fogjuk bemutatni.

7.1. JavaServer Faces

A JavaSever Faces (JSF) ([79]) egy elterjedt webprogramozási keretrendszer, melynek célja a programozókat segíteni abban, hogy csapatmunkában, minél kevesebb kódírással, a lényegre koncentrálva, biztonságos és ügyfélbarát programot tudjanak írni.

A JSF a csapatmunkát oly módon támogatja, hogy megvalósítja a Modell-Nézet-Vezérlő (Model-View-Controller, MVC, [82]) szerkezeti mintát, melynek lényege, hogy szétválasztja az adatok (modell) és a felhasználói felület (nézet) kezelését és ehhez egy harmadik összetevőt (vezérlőt) is használ. A nézet többnyire egy JSP lap, a modell, illetve a vezérlő pedig menedzselt JavaBean (Managed Bean, Backing Bean) lesz. Ezáltal átláthatóbb lesz az alkalmazás szervezése és a feladatok nagymértékben szétválaszthatók és külön fejleszthetők lesznek.

A JSF tartalmaz grafikus felhasználói komponenseket megvalósító API-kat, továbbá ezekhez kapcsolódó eseménykezelést és más szolgáltatásokat megvalósító komponenseket is. Például, a konverterek a böngésző által szövegesen elküldött adatokat a kívánt formátumra konvertálják, a validátorokkal pedig az adatok szemantikai helyességét lehet ellenőriztetni (például azt, hogy egy érték benne van-e az általunk elvárt intervallumban). Ugyanakkor egyéni validációs komponensek készítésére is van lehetőség. A JSF olyan JSP könyvtárakat is tartalmaz, melyek a JSP oldalakról JSF komponenseket vezérlő interfészeket valósítanak meg.

Egy JSF alkalmazás lényegében a következőképpen épül fel. Először is meg kell írnunk az üzleti logikát megvalósító bean osztályokat. Egy bean osztály akkor lesz menedzselt, hogyha a megfelelő konfigurációs állományban megadjuk, hogy a JSF-et használó `.jsp` állományban milyen néven akarjuk elérni annak egy példányát, továbbá a bean objektum hatókörét, mely lehet kérés, munkamenet vagy alkalmazás szintű. Természetesen meg kell írnunk a megjelenítést szabályozó JSP oldalakat is, melyek tartalmazhatnak klasszikus HTML tagokat, illetve speciális JSF elemeket, például olyan HTML tagok helyett, melyek kiszolgálóval történő kommunikációt igényelnek (kiíratás, űrlapmezők, gombok stb.). Az ilyen JSF elemekhez hozzárendelhető az adott elemet kezelő metódus, mely tipikusan a konfigurációs állományban a JSP oldalnak megfeleltetett menedzselt bean adott példányának egy megfelelő adattagjához tartozó `set` vagy `get` metódus vagy eseménykezelő lesz. Ily módon lehetséges a felhasználó eseményeinek szerver oldali kódhoz történő kötése.

A fentebb említett konfigurációs állományban az is leírható, hogy adott nézetek (`.jsp` fájlok) esetén, adott menedzselt bean osztályban specifikált adott metódus (vagy az összes) meghatározott visszatérési értéke esetén a vezérlés mely másik erőforrásnak legyen átadva. A JSF támogatja az AJAX használatát is.

JSF fájlok szerkesztésére számos fejlesztőeszköz rendelkezésre áll, például a Netbeans IDE, vagy az Eclipse.

JSF-re vonatkozó további leírásokat az olvasó az alfejezet forrásaként is használt [14], továbbá [79], [80], [81], [82] irodalmakban talál.

7.2. Google Web Toolkit

A felhasználói élmény és a termelékenység fokozása érdekében számos további technológiát fejlesztettek ki, melyek alkalmasak úgynevezett gazdag internet alkalmazásoknak (Rich Internet Application, RIA) készítésére. A gazdag alkalmazások magukban hordozzák az asztali alkalmazások számos jellemzőjét, és telepíthetők önálló alkalmazásként, böngésző kiegészítőként vagy virtuális géppel együtt. Gazdag alkalmazások fejlesztésére alkalmas technológiák például az Adobe Flash, JavaFX, Microsoft Silverlight, Google Web Toolkit. A következőkben az utóbbi technológia főbb jellegzetességeit fogjuk bemutatni.

A Google Web Toolkit (GWT) ([82]) nem egy tipikus keretrendszer, hanem egy olyan eszköztár, mely lehetővé teszi asztali alkalmazások fejlesztésénél megszokott eszközök használatát JavaScript kódot tartalmazó alkalmazások készítésére. GWT használata esetén Javaban programozunk, ezért telepítenünk kell a JDK-t és a Google Web Toolkit SDK-t, mely tartalmaz osztálykönyvtárakat, fordítót és Eclipse kiegészítőt is.

Amint az AJAX tárgyalásánál láttuk, annak ügyféldali részét hagyományosan JavaScriptben írják. Csakhogy a JavaScriptnek számos olyan hiányossága van, mely nehézkessé teszi nagyobb programok fejlesztését. Például nem rendelkezik az objektumorientáltság olyan szintű megvalósításával, mely lehetővé tenné az alkalmazások tiszta objektumközpontú fejlesztését, továbbá értelmező nyelvként a hibák többnyire csak futási időben derülnek ki, így nehéz azok korábbi tesztelése. A GWT a JavaScript hiányosságait oly módon küszöböli ki, hogy olyan Java komponenseket bocsát a fejlesztő rendelkezésére, melyek később ügyféloldali (JavaScript) kódokra lesznek lefordíthatók. Ily módon a fejlesztő az ügyféloldalinak szánt programkódot is Java nyelven írja, és ehhez megszokott fejlesztőkörnyezeteket (például Eclipse) használhat. Figyelni kell azonban arra, hogy bizonyos Java eszközök a Java programozásban megszokottól eltérően (többnyire a keletkező JavaScript kódnak megfelelően) vagy egyáltalán nem használhatók. A fejlesztés során az alkalmazás úgynevezett gazda módban futtatható és tesztelhető a GWT beágyazott böngészőjében és általa ellenőrzött környezetben. Itt a programozó azt a kimenetet látja, ami majd a végleges változatban a böngészőben is látszani fog, de hibakeresést segítő szolgáltatásokat is igénybe vehet. Ugyanakkor továbbra is lehetőség van a DOM objektumok elérésére és a CSS eszközkészlet használatára. Végül, az elkészült kód lefordítható JavaScript kódra, így a felhasználó már csak a keletkezett JavaScript kódot fogja futtatni, amihez a böngészőn kívül nincs szüksége semmilyen további programra (például bővítményre vagy virtuális gépre). A generált kód kezeli a böngészők közötti különbségeket, a népszerűbb számítógépes és mobil böngészőkre optimalizált kódot kapunk.

A GWT-t nagyon sok szolgáltatást nyújtó könyvtárral is ellátták, mely lehetővé teszi gazdag internet alkalmazások készítését. A felhasználói felület kezelésére a vezérlők (widgetek), míg a megjelenítés szabályozására panelek használhatók.

Az AJAX lényegét képező aszinkron kommunikációra több modult használhatunk.

A GWT támogatja az XML és JSON adatok kezelését is.

Az olvasó GWT-vel kapcsolatos részletesebb információkat az alfejezet forrásaként is használt [12] és [83] irodalmakban talál.

8. Web design

A web design olyan irányelveket fogalmaz meg, melyek lehetővé teszik, hogy oldalaink mások számára elérhetők és minél vonzóbbak legyenek. Az előző fejezetekben már említettünk web design jellegű ajánlásokat, most ezeket fogjuk kissé kiegészíteni.

1994-ben Tim Berners-Lee kezdeményezésére létrejött a W3C (World Wide Web Consortium) ([84]), mely 1997-ben létrehozta a WAI (Web Accessibility Initiative) ([85]) kezdeményezést és megfogalmazta a web elérhetőség irányelveit. 1999-ben az Európa Tanács az „e-Europe: An information society for all" című dokumentumban leszögezte, hogy a tagállamoknak 2001 végéig a közügyekkel foglalkozó weboldalak tartalmát és szerkezetét fogyatékosok számára hozzáférhetővé kell tenni ([86]). Ezzel szemben jelenleg a magyar oldalak csak nagyon kis számban teljesítik ezt a feltételt, pedig az elérhetőség biztosítása nem csak etikai, hanem üzleti okokból is hasznos lenne. Egyrészt valószínűsíthető, hogy a fogyatékkal élő emberek az átlagosnál sokkal nagyobb arányban vennének igénybe internetes szolgáltatásokat, ha ezek számukra könnyen használhatók lennének. Például, egy vak ember könnyebben tudna vásárolni egy webáruházban, mint egy valódi áruházban, ha a webáruház támogatná a felolvasó szoftvert. Továbbá közcélú oldalak esetén, az élet más területeihez hasonlóan, az akadálymentesség nem lehet opció kérdése, hanem mindenképpen megvalósítandó. Másrészt, az elérhetőségi irányelvek betartása lehetővé tenné az oldalak használhatóságát különböző feltételek (eszközök, sávszélesség stb.) mellett is, ami egyre több felhasználót elcsábíthatna a konkurenciától. Mit lehet tenni az oldalak elérhetősége érdekében? Először is írjunk szabványos oldalakat, mert a szabványok úgy vannak kitalálva, hogy minél szélesebb támogatást biztosítsanak. Ugyanakkor az érvényesség ellenőrző nem érti a weboldal tartalmát, ezért a szabványosság csak szükséges, de nem elégséges. Nagyon fontos az is, hogy mindent arra használjunk, amire való, hiszen a weboldal kódját értelmező programok csak így tudják a tartalmat a céljuknak és körülményeiknek legjobban megfelelően feldolgozni és a kimenetre küldeni. Például ne használjunk címsort kizárólag formázás érdekében, de tegyünk címsorba mindent, ami tartalmilag címnek számít. Hasonlóképpen, az űrlapok mezőit lássuk el helyesen megvalósított elemfeliratokkal. Ne használjunk táblázatokat kizárólag formázásra, illetve ha a tartalom jellege miatt valóban indokolt táblázat használata, akkor a cellákat rendeljük hozzá a fejlécekhez. A képeknek mindig adjuk meg a funkcióját az alt jellemzővel és kerüljük képek vezérlőelemekként történő (például képtérképek) használatát, vagy adjunk meg alternatív lehetőséget is. A progresszív fejlesztés irányelve azt mondja ki, hogy az oldalt úgy kell kialakítani, hogy a lehető legegyszerűbb eszközökkel is használható legyen, és a kényelmi, illetve kinézetet javító funkciókat egy további rétegként kell elhelyezni. A szín ne hordozzon másképp nem érzékelhető információt, és legyen elég kontraszt a háttér és az előtér színei között. Multimédiás tartalmakat lehetőleg ne erőltessünk a felhasználóra, mert nem biztos, hogy rendelkezik a megjelenítéséhez szükséges programmal (és csak miattunk nem fogja letölteni), kisebb sávszélesség mellett ezek lassan töltődhetnek le, zavarók lehetnek, illetve ha már ilyeneket használunk, akkor célszerű ezeket a tartalommal nem interferáló redundáns információkkal (hanghoz szöveg és fordítva) ellátni. Fontos, hogy az oldal struktúráját úgy alakítsuk ki, hogy annak lényegi tartalma a tervezettől eltérő méretben (más felbontásban, más betűmérettel), ha nem is olyan szép, de elérhető legyen (például tegyük lehetővé szükség esetén a görgetősávok megjelenítését). Kétségtelen, hogy nem lehet minden körülményre optimalizált oldalt készíteni, viszont a statisztikai adatok azt mutatják, hogy a felhasználók a webet első sorban információk keresésére használják, ezért a legfontosabb, hogy az oldalak minél külön-

bözőbb feltételek között is használhatók legyenek ([15]). Mozgáskorlátozott felhasználók érdekében a vezérlőelemeket ne helyezzük túl közel egymáshoz, az időzített funkcióknál pedig elegendően nagy reakcióidőket biztosítsunk. Tegyük lehetővé az oldalak használatát az egértől eltérő input eszközökkel is (pl. billentyűzettel a 2.10. fejezetben leírtak segítségével, vagy emberi hang útján). Az akadálymentességről további információkat a [15], [85], [88], [89] irodalmakban találunk.

Természetesen az akadálymentesség önmagában még nem elégséges a látogatók gyűjtéséhez és megtartásához. A látogatók gyűjtéséhez az oldalakat úgy kell kialakítanunk, hogy könnyen megtalálhatók legyenek (például keresők által, ezt nevezik keresőoptimalizálásnak [90], [91]). Hasznos továbbá felkerülni minél több, az oldalunkkal kapcsolatos tartalmú más megbízható oldalak linkjei közé. A felhasználók megtartásához oldalainknak megbízható, könnyen átlátható és elérhető tartalommal kell rendelkezni, annak stílusa és kinézete a megcélzott felhasználóknak megfelelő kell legyen. Ez azt is jelenti, hogy az oldal elkészítése előtt pontosan be kell határolnunk a megcélzott felhasználók körét, számítva arra, hogy minél szélesebb rétegeket célzunk meg, annál több (idő és munka) költséget fog igényelni az oldal megfelelő kialakítása. Hogyha népszerű oldalakat akarunk készíteni, akkor alapszabály, hogy az oldalak kialakításánál a saját önzőségünk helyett a látogató igényeit részesítsük előnyben. A látogatók gyűjtésére és megtartására vonatkozó további információkat az olvasó a fejezet alapvető forrásaként is használt [15] irodalomban talál.

Irodalomjegyzék

1. http://tananyagfejlesztes.mik.uni-pannon.hu
2. http://tananyagfejlesztes.mik.uni-pannon.hu/
   - Dr. Kató Zoltán (SZTE) TÁMOP programnyitó értekezleten elhangzott előadása (TAMOP412Anyito.pdf)
3. http://www.nefmi.gov.hu/felsooktatas
4. Bilicki Vilmos: Programrendszerek fejlesztése, http://www.tankonyvtar.hu/, TÁMOP-4.1.2-08/1/A-2009-0008
5. Virginia DeBolt: HTML és CSS. Webszerkesztés stílusosan, Kiskapu Kft., 2005
6. Eric A. Meyer: CSS zsebkönyv, Kiskapu Kft, 2006
7. Michael Moncur: Tanuljuk meg a JavaScript (2.0) használatát 24 óra alatt, Kiskapu Kft., 2006 (Michael Moncur: SAMS Teach Yourself JavaScript in 24 Hours, Sams Publishing, 2007)
8. Christian Warez: JavaScript zsebkönyv, Kiskapu Kft., 2006
9. Hans Bergsten: Java Szervletek Programozása, Kiskapu Kiadó, 2002 (Hans Bergsten: Java Servlet Programming, O'Reilly, 2002)
10. Gál Tibor: Webprogramozás, Műegyetemi Kiadó Budapest, 2006
11. N. C. Zakas, J. McPeak, J. Fawcett: Professzionális AJAX, a második kiadás fordítása, Szak Kiadó, 2007 (Nicholas C. Zakas, Jeremy McPeak, Joe Fawcett: Professional Ajax, 2nd Edition (Kindle Edition), Wrox, 2007)
12. Ryan Dewsbury: Google Web Toolkit alkalmazások, Kiskapu Kft. 2008 (Ryan Dewsbury: Google Web Toolkit Applications, Addison-Wesley Professional, 2007)
13. Antal Margit: Java alapú webtechnológiák, Scientia Kiadó, Kolozsvár, 2009
14. Balogh Péter, Berényi Zsolt, Dévai István, Imre Gábor, Soós István, Tóthfalussy Balázs: Szoftverfejlesztés Java EE platformon, Szak Kiadó, 2007.
15. Jakob Nielsen: Web-design, Typotex, 2004.
16. Wallace, Patricia M.: Az internet pszichológiája, Osiris, 2002.
17. http://webtortenet.lap.hu/
18. http://www.w3.org/2011/02/htmlwg-pr.html
19. http://www.ietf.org/rfc/rfc1738.txt
20. http://www.w3.org/2002/11/dbooth-names/rfc2396-numbered_clean.htm
21. http://unicode.org/reports/tr36/tr36-8.html
22. Jeffrey E. F. Friedl: Reguláris kifejezések mesterfokon, Kossuth Kiadó, 2003.
23. http://tomcat.apache.org/
24. http://www.apache.org/licenses/LICENSE-2.0.html
25. http://tomcat.apache.org/tomcat-3.3-doc/tomcat-ug.html
26. http://classicasp.aspfaq.com/forms/what-is-the-limit-on-querystring/get/url-parameters.html
27. http://download.oracle.com/javaee/5/api/javax/servlet/http/HttpServletResponse.html
28. http://support.microsoft.com/kb/310676
29. http://support.microsoft.com/kb/2004188
30. http://www.w3.org/wiki/HTML/Specifications
31. http://www.w3.org/Style/CSS/
32. http://www.w3.org/2005/10/howto-favicon
33. http://www.evotech.net/blog/2008/09/http-equiv-meta-attribute-values-for-http-equiv/

34. http://www.w3schools.com/TAGS/att_meta_http_equiv.asp
35. http://www.w3.org/International/O-HTTP-charset
36. http://weblabor.hu/cikkek/karakterkodolasiproblemakkikuszobolese
37. http://wiki.ham.hu/index.php/Karakterk%C3%B3dol%C3%A1s
38. http://www.w3.org/TR/CSS2/fonts.html
39. http://www.standardsmode.hu/html-css/szoveg/
40. http://xhtml.com/en/xhtml/reference/
41. http://webdesign.about.com/od/layout/a/aa111102a.htm
42. http://webdesign.about.com/od/css/a/aa102102a.htm
43. http://www.prosign.hu/lap/cikk/szin_2_csoportositas.htm
44. http://www.epab.bme.hu/epinf1/S-Weblap-Gyorsreferencia/Szinek.html
45. http://www.w3.org/TR/CSS21/box.html
46. http://www.standardsmode.hu/html-css/dobozmodell/
47. http://www.standardsmode.hu/html-css/alosztalyok-alelemek/
48. http://www.w3schools.com/media/media_browservideos.asp
49. http://weblabor.hu/cikkek/cssalapjai5
50. http://www.w3.org/TR/CSS21/generate.html
51. http://www.webstandards.org/learn/tutorials/accessible-forms/intermediate/
52. http://xhtml.com/en/xhtml/reference/label/
53. http://reference.sitepoint.com/html/a/tabindex
54. http://www.w3.org/TR/html4/interact/forms.html
55. http://ergonomia.ginweb.hu/book/export/html/48
56. http://www.w3.org/TR/1999/xhtml-modularization-19990406/DTD/doc/xhtml1-f.html
57. http://www.createblog.com/html-tutorials/118-replacing-the-iframe-by-the-object-in-xhtml-1.1/
58. http://www.w3.org/DOM/
59. http://www.dustindiaz.com/getelementsbyclass/
60. Ben Laurie, Peter Laurie: Apache, Kossuth Kiadó, 2001.
61. http://www.oracle.com/technetwork/java/javaee/servlet/index.html
62. http://www.oracle.com/technetwork/java/download-139443.html
63. http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html
64. http://download.oracle.com/javase/1.5.0/docs/guide/net/http-keepalive.html
65. http://download.oracle.com/javaee/1.3/api/javax/servlet/ServletResponse.html
66. http://docstore.mik.ua/orelly/java-ent/servlet/ch08_01.htm
67. http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/Servlet-Tutorial-Session-Tracking.html
68. http://www.javamex.com/tutorials/servlets/session_api.shtml
69. http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-servlets.html?page=1
70. http://download.oracle.com/javaee/1.3/api/javax/servlet/http/HttpSession.html
71. http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications
72. http://www.oracle.com/technetwork/articles/javaee/ajax-135201.html
73. http://www.ibm.com/developerworks/web/library/wa-ajaxintro1/index.html
74. http://www.ibm.com/developerworks/web/library/wa-ajaxintro2/
75. http://www.ibm.com/developerworks/web/library/wa-ajaxintro3/index.html
76. http://www.oracle.com/technetwork/java/javaee/jsp/index.html
77. http://download.oracle.com/javaee/5/tutorial/doc/bnagx.html
78. http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/
79. http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html
80. http://www.jsftutorials.net/

81. http://www.oracle.com/technetwork/articles/javase/javaserverfaces-135231.html
82. http://lib.bioinfo.pl/courses/view/185
83. http://code.google.com/intl/hu-HU/webtoolkit/
84. http://www.w3.org/History.html
85. http://www.w3.org/WAI/
86. http://europa.eu/legislation_summaries/information_society/strategies/l24221_en.htm
87. http://www.w3.org/People/Berners-Lee/
88. http://www.w3c.hu/forditasok/WAI/intro/accessibility.html
89. http://www.d.umn.edu/itss/support/Training/Online/webdesign/accessibility.html
90. http://www.oldalgazda.hu/
91. http://www.xn--keresoptimalizls-smbb03s.net/

 © Holló Csaba, SzTE

Tárgymutató

© Holló Csaba, SzTE

© Holló Csaba, SzTE

© Holló Csaba, SzTE