Szakdolgozat

Sike Csaba

Debrecen

2011

Debreceni Egyetem

Informatika Kar

Információ Technológia Tanszék

Mobilszoftver készítése Java nyelven

Témavezető:

Dr. Kósa Márk

Egyetemi tanársegéd

Készítette:

Sike Csaba

Programtervező informatikus

Debrecen

2011

Tartalomjegyzék

1 Bevezetés

1.1 A mobiltelefonok társadalmi hatásai

A mobiltelefonok rohamos terjedése és fejlődése a mai embernek nem jelent újdonságot. Viszont ha belegondolunk, hogy ezek az eszközök eleinte csak hangátvitelre alkalmas adó-vevő eszközök voltak, már jobban elcsodálkozunk. Amúgy is rohamos fejlődésük csak egyre gyorsabb és gyorsabb lesz. Magyarországon a 2000es évek elejére a mobiltelefonok széles körben elterjedtek. A Nemzetközi Hírközlési Hatóság kiadványai szerint 1999 elején még csak 1,1 millió előfizetést tartottak nyilván az országban, 2000 elején ez a szám már 1,9 millió, 2001 elején 3,5 millió, 2002-es év elején 5,4 millió, 2003-ban 7 millió, 2004-ben 8 millió, 2005 elején 8,8 millió, 2008-ra ez a szám 11 millió lett, 2010 végére pedig elérte a 12 milliós értéket, vagyis számtanilag több mobiltelefon előfizetés van az országban, mint amennyi állampolgárja van hazánknak. A számkiszolgálás miatt 1998-ban a mobiltelefon számokat 6 jegyűről 7 jegyűre változtatták meg. Az NHH közleménye szerint 2010. január 15-től a mobiltelefon számokat csak a teljes formátum, azaz mind a 11 számjegy (előhívó, körzetszám és telefonszám) tárcsázásával lehet csak hívni, megszűnik az egy hálózaton belüli hét számjegyes hívás lehetősége. Ezt a rohamos terjedést több tényező is segítette. A legfontosabb, hogy a mobiltelefonok ára jelentősen csökkent. Másrészről a fenntartási költségek is egyre megfizethetőbbek, köszönhetően a szolgáltatók közötti intenzív versenynek. A jövő szempontjából gondot jelent a szolgáltatóknak, hogy a telekommunikációs szolgáltatásokból már nem tudnak számottevő újabb profitot termelni, hiszen új felhasználókat csak nehezen nyerhetnek meg.

1.2 A Java nyelv története

1990 decemberében a Sun Microsystems szórakoztató elektronikai készülékek számára fejlesztett beágyazott operációs rendszer szoftvereket, ez volt a Green Project. Ebben a fejlesztői csapatban dolgozott James Gosling, Patrick Naughton és Mike Sheridan. 1991-ben a C++ fordító kiterjesztésével foglalkoztak, céljuk egy olyan platformfüggetlen, objektum-orientált nyelv kidolgozása volt, mely lehetővé teszi a különféle eszközök (CD játszó, műholdvevő, stb.) közötti kommunikációt. Gosling felismerte a C++ alkalmatlanságát erre a feladatra, és a csapat megtervezte saját nyelvét, az Oak-ot. A név állítólag úgy jutott Gosling eszébe, hogy éppen egy könyvtárat hozott létre az új nyelvnek, és az ablakon kinézve meglátott egy tölgyfát. 1995-ben az Oak-ot egy már létező nyelvvel való névhasonlatosság miatt át kellett nevezni, ekkor kapta a Java nevet. A Java névválasztás a fejlesztők kávézási szokásából ered. (Jáva szigetéről származó kávét fogyasztottak az új név kiválasztásakor.) Ugyanebben az évben adta ki a nyelvet a Sun Microsystems, ekkor már a főbb gyártó és fejlesztő cégek bejelentették a Java technológiával való együttműködést. Az Internet, és a World Wide Web akkori rohamos terjedése miatt a Java hamarosan népszerű lett, mint biztonságos nyelv. 1996 körül a Microsoft-ot már annyira aggasztotta a Java sikere, hogy elkezdte fejleszteni saját Java verzióját, amely végül a Visual J++ nevű fejlesztői környezetben öltött testet. A Microsoft azonban igyekezett a Java-t a Windows-hoz kötni, emiatt a nyelv platformfüggetlenségét aláásó, Windows-specifikus megoldásokat valósított meg benne. Ezt a Java-t a szakmai tolvajnyelv "polluted Java"-nak, szennyezett Java-nak nevezte el. Mivel a Microsoft ezt a nyelvet is Java-nak hívta, a Sun pert indított a Microsoft ellen. Az évekig tartó eljárás vége az lett, hogy a Microsoft-nak fel kellett hagynia a „polluted Java" fejlesztésével, és a Windows XP-be már a Microsoft Java VM-et sem tehette bele. Természetesen ekkor a Microsoft már készült kitolni a gyárkapun saját „nagyágyúját”, a .NET platformot, és vele együtt a virtuális gép felett futó nyelveket, többek között a C#-ot. 2009-ben nagy változás következett be a Java „életében”, az Oracle 7,4 milliárd dollárt fizetett a Sun részvényeiért, ezzel együtt megvette a Java-t is. Nem sok idő telt el, és James Gosling otthagyta a Sun-t (vagyis már az Oracle-t). 2010-ben történt, hogy az Oracle beperelte a Google-t az Android-os Java miatt, valamint az Apple bejelentette, hogy a Mac-es Java-t nem fejlesztik tovább.

1.3 A Java nyelv jellemzői

- **Egyszerű:** A nyelv a C++ egyszerűsített változata. A Java nyelv sokkal kevesebb nyelvi eszközt sokkal nagyobb kötöttségekkel tartalmaz.
- **Objektumorientált:** A Java tisztán objektumorientált nyelv, egy Java alkalmazás nem tartalmaz globális, illetve osztályokhoz nem köthető kódot.
- **Elosztott:** A Java alkalmazások képesek hálózatos környezetben, az internet egységes erőforrás azonosítóival egyértelműen azonosított objektumokat kezelni. A Java rendszer továbbá képes a távoli eljáráshíváson alapuló kliens-szerver jellegű alkalmazások fejlesztésére.
- **Robosztus:** A nyelv tervezésekor fontos szempont volt a programozói hibák már fordítási időben való kiszűrése, a futási idejű hibák minimalizálása.
- **Hordozható:** A Java nyelv és a programozási környezet nem tartalmaz implementációfüggő elemeket. Egy adott forrásból minden Java fordító ugyanazt a tárgykódot állítja elő.
- **Architektúra-független:** A fordítás során az azonos típusok minden környezetben azonos méretű memóriaterületet foglalnak, a fordítás után előálló class fájlok bármelyik szabványos virtuális gépen futtathatóak. A futtató környezet a tárgykódú állományokat azonos módon hajtja végre. Az egyes architektúrákhoz és operációs rendszerekhez elkészített virtuális gépek alakítják át a class állományokat natív kódokra.
- **Interpretált:** Az egyes hardvereken futó natív kódot a virtuális gép futási időben hozza létre, a lefordított tárgykódú állományokból.
- **Többszálú:** A Java környezet támogatja a párhuzamos programozást, és alkalmas párhuzamos algoritmusok megvalósítására. Ezt az adott hardveren úgy valósítja meg, hogy egy időben egymástól függetlenül több programrészt futtat. Egyprocesszoros rendszereknél ezt természetesen egymás utáni végrehajtással, a szálak ütemezésével oldja meg.
- **Dinamikus:** Az osztálykönyvtárak szavadon továbbfejleszthetők, és már a tervezésnél figyelembe vettél, hogy egy esetleges későbbi bővítés a már megalkotott programok működését ne akadályozza.

1.4 Java verziók

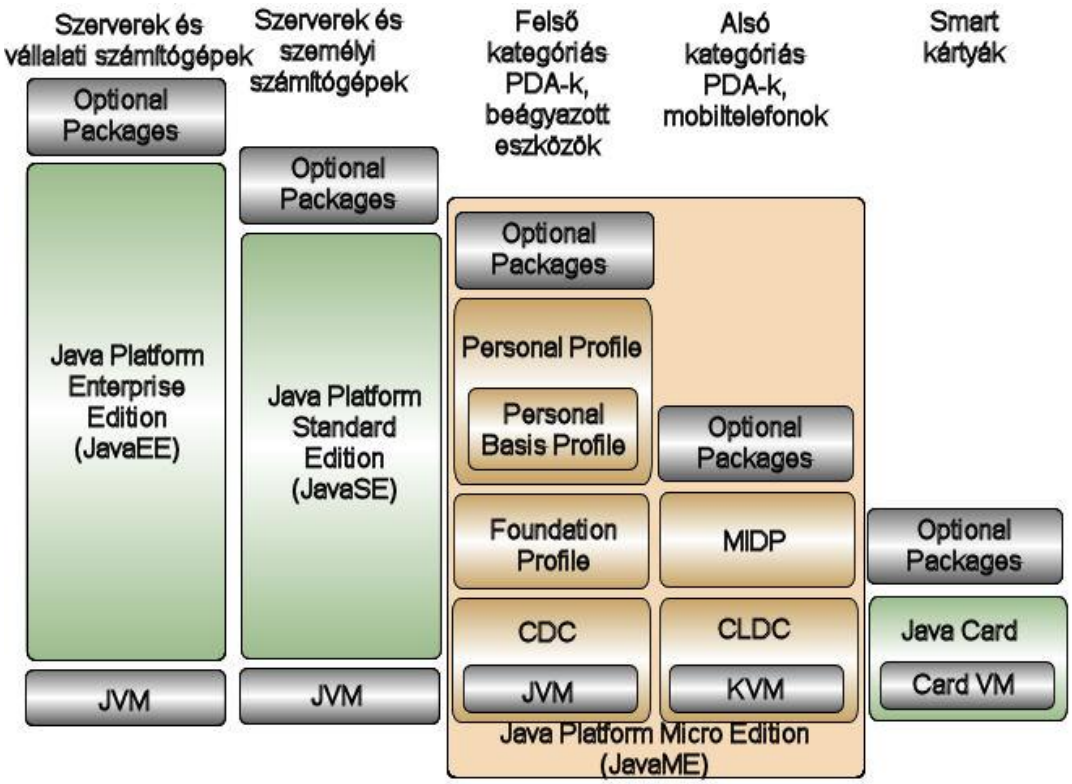
- 1.0 (1996) – ez volt az első verziója a Java virtuális gépnek és az osztálykönyvtáraknak.
- 1.1 (1997) – itt jelent meg először a belső osztály fogalom, ami lehetővé teszi több osztály egymásba ágyazását.
- 1.2 (1998) – kódneve Playground. Ez a verzió számottevő mérföldkő volt a nyelv evolúciójában. Hogy ezt kihangsúlyozza, a Sun hivatalosan *Java 2*-nek nevezte el.
- 1.3 (2000) – kódneve Kestrel. Csak néhány kisebb változtatást végeztek el rajta.
- 1.4 (2002) – kódneve Merlin.
- 5 (2004) – belső számozás szerint 1.5, kódneve Tiger, újdonsága például a továbbfejlesztett ciklusmegoldások, az adattípusok automatikus objektummá alakítása (autoboxing), a generic-ek.
- 6 (2006) – belső számozás szerint 1.6.0, kódneve Mustang. Decemberben jelent meg a végleges változat kiterjesztett nyomkövetési és felügyeleti megoldásokkal, szkriptnyelvek integrációjával, grafikusfelület-tervezést támogató kiegészítésekkel.
- 7 (2011?) – kódneve Dolphin
- 8 (2012?)

1.5 Java Virtuális Gép (JVM)

Java virtuális gépnek (röviden JVM) nevezik a Sun Microsystems által specifikált Java programozási nyelvhez készített virtuális gépeket. A JVM alapvető feladata a Java byte-kód futtatása, amely platform független. A Java byte-kód általában Java nyelvű forrás fordításával jön létre, de léteznek olyan fordítók, amelyek más programnyelvek forrásait fordítják Java byte-kódra. Fontosabb elemei:

- Osztálybetöltő (class loader), a főbb ellenőrzéseket végzi a byte-kódon, előkészíti futtatásra
- Szemétgyűjtő (garbage collector), működés közben a nem használt objektumokat eltávolítja a memóriából, ezzel helyet szabadít fel
- Végrehajtó motor (execution engine), a tulajdonképpeni végrehajtást végzi

1.6 Java platform



1.6.1 JavaEE

Enterprise Edition. Alapját a JavaSE jelenti. Vállalati, üzleti alkalmazások szerver oldali fejlesztésére szánt változat. Legfontosabb eleme a servlet, amely a szerver oldalon futva a szerver-futtatókörnyezet részeként a kliens felől érkező kérések kiszolgálását végzi.

1.6.2 JavaSE

Standard Edition. Főként hagyományos, általános célú programok és desktop alkalmazások fejlesztésére, futtatására, tesztelésére használják. Szokásos felhasználási módja még a böngészőkben az applet-ek futtatása. A platform megfelelően nagy memóriával és teljesítményű processzorral rendelkezik.

1.6.3 JavaME

Micro Edition. Kis erőforrású eszközökre, mobiltelefonokra, PDA-kra történő fejlesztésre találták ki.

1.6.4 Java Card

Kis teljesítményű processzorral és kevés memóriával rendelkező eszközökre való fejlesztésre találták ki. Elvárás, hogy az alkalmazások képesek legyenek együttműködni a különböző gyártók kártyáival.

2 Java Micro Edition (Java ME)

2.1 Konfiguráció

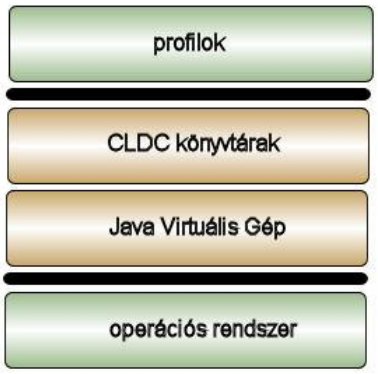
A konfigurációk a virtuális gépből, valamint az osztálykönyvtárak egy minimális halmazából állnak. A két fő konfiguráció a Connected Limited Device Configuration (CLDC) és a Connected Device Configuration (CDC). A CDC-t azokra az eszközökre tervezték, melyeknek gyorsabb a processzora, több memóriával rendelkezik és nagyobb a hálózati sávszélessége. (TV, videotelefon, GPS)

A CLDC-nek a Sun kifejlesztette a virtuális gép referencia implementációját, ami a Kilobyte Virtual Machine nevet kapta, vagy simán KVM. Tehát a CLDC tulajdonképpen a KVM és az alapvető API-k összessége.

2.2 KVM – Kilobyte Virtual Machine

128 kilobájt memóriával is elfut, 40-80 kilobyte statikus memória a virtuális gép magjának, (fordítási beállításoktól és a cél platformtól függ), 20-40 kilobyte dinamikus memória (heap), a maradék konfigurációs és profil osztály könyvtáraknak (class libraries) van fenntartva. 16-bites, 25 MHz-es processzorokon is elfut. A CLDC csak KVM-en fut, és a CLDC az egyetlen konfiguráció, amit a KVM támogat. (a CDC virtuális gépének specifikációja megegyezik a J2SE-vel)

2.3 CLDC – Connected Limited Device Configuration



2.3.1 CLDC 1.0 (JSR 30)

Célok:

- 160-512 kB memória a Java platformnak
  - 128 kilobyte memória a Java futtatáshoz
  - 32 kilobyte memória futásidejű memóriakiosztáshoz
- alacsony energiafelvétel, gyakran elemmel működik
- csatlakoztathatóság valamilyen hálózathoz, gyakran vezeték nélkülihez, szakaszos kapcsolat, korlátozott (gyakran 9600 bps vagy kevesebb) sávszélességgel

Biztonság:

- nem lehet képes kárt okozni az eszközben, amelyiken fut
- Classfile ellenőrző biztosítja, hogy a VM-be betölttött classfileok nem hajtódnak végre semmilyen módon, amit a Java VM specifikáció nem enged
- nem írhatja felül a rendszer-osztályokat a java.*, javax.microedition.*, vagy más profil- vagy rendszer-specifikus csomagokban

Kihagyott tulajdonságok:

- Java Native Interface (JNI)
- lebegőpontos ábrázolás
- felhasználó-definiált JAVA osztálybetöltő (class loader)
- Reflection támogatás (komponensek felderítése)

- szálcsoportok és démon szálak
- Finalization (nem tartalmazza az Object.finalize() metódust)
- gyenge hivatkozások
- korlátozott hibakezelés (A java.lang.Error legtöbb leszármazottja nem támogatott)

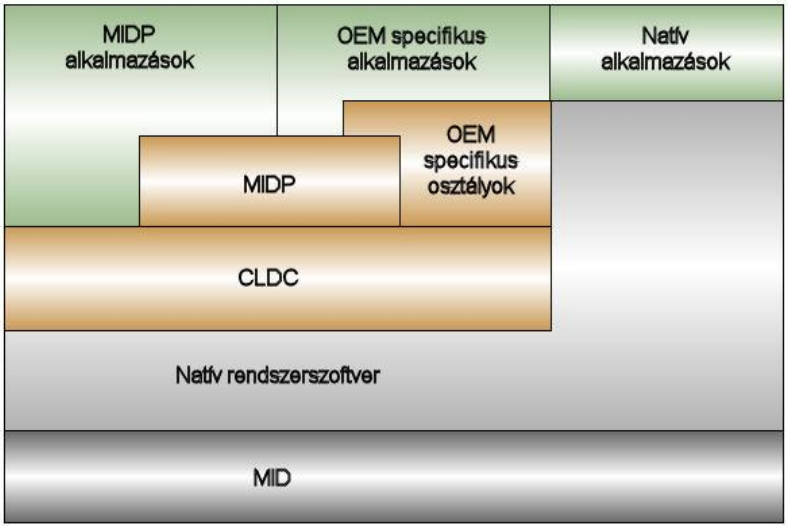
2.3.2 CLDC 1.1 (JSR 139)

Célok:

- legalább192 kB teljes memória a Java platformnak
    - legalább 160 kilobyte “nem felejtő memória” a virtuális gépnek és a CLDC könyvtáraknak.
    - legalább 32 kilobyte „felejtő memória” alkalmazásfuttatáshoz (pl. objektum heap)
- 16 vagy 32-bites processzor
- alacsony energiafelvétel, gyakran elemmel működik
- csatlakoztathatóság valamilyen hálózathoz, gyakran vezeték nélkülihez, szakaszos kapcsolat korlátozott sávszélességgel

2.4 MIDP – Mobile Information Device Profile

A CLDC a MIDP-vel kombinálva adja a mobiltelefonok és a belépő szintű PDA-k teljes Java futtató környezetét. Ezt a specifikációt a MIDPEG (Mobile Information Device Profile Expert Group) nevű csoport készítette.



2.4.1 MIDP 1.0 (JSR 37)

Hardveres követelmények:

- Kijelző:
    - méret: 96 x 54
    - színmélység: 1-bit
    - torzítási arány: 1:1
- Input:
    - egykezes billentyűzet (ITU-T); pl. szokványos mobiltelefon készülék
    - kétkezes billentyűzet (QWERTY); pl. PalmTop
    - érintőképernyő
- Memória:

    - 128 kilobyte non-volatile memória (tartalmát megőrzi a készülék) a MIDP komponenseknek
    - 8 kilobyte non-volatile memória az alkalmazás által létrehozott perzisztens adatoknak
    - 32 kilobyte volatile memória a Java futtatásának (pl. Java heap)

- Hálózat:
    - kétirányú
    - vezeték nélküli
    - korlátozott sávszélességgel

MIDP 1.0 API

- alkalmazás életciklusa (`javax.microedition.midlet`)
- felhasználói felület (`javax.microedition.lcdui`)
- perzisztens tárolás (`javax.microedition.rms`)
- hálózatkezelés (`javax.microedition.io`)

2.4.2 MIDP 2.0 (JSR 118)

Újdonságok:

- Game API (`javax.microedition.lcdui.game`)
    - GameCanvas (fullscreen lehetőség)
    - RGB képek
    - TiledLayer, Layer, Sprite (transzformációkkal és ütközésdetektálással)
- Media API (`javax.microedition.media`) – médiafájlok letöltése vagy folyamatos lejátszása
- UI
    - lehetséges a nyomógombok állapotát is követni (pl. folyamatosan lenyomva) és rendelkezik egy háttérbufferrel is, aminek tartalmát egy művelettel lehet a képernyőre másolni
    - a Layer egy rajzolható grafikus elem, amit rá lehet tenni a képernyő adott pozíciójára, megjeleníteni, ill. eltűntetni

    - a Sprite egy animáció, amit egyetlen kép részletei írnak le. Az animáció kockáit egy képbe öntik (pl. egymás mellé teszik), a sprite kijelezhető a képernyő adott pozícióján, az animációs fázis pedig beállítható
    - a TiledLayer nagyméretű scrollozó hátterek készítésére alkalmas kisebb képelemekből
- Háttérvilágítás, vibrálás
- Formok
- Biztonság
- Hálózatkezelés

2.5 MIDlet

A MIDP-környezetben futó Java-alkalmazásokat MIDleteknek nevezzük. A MIDleteket egy alkalmazásmenedzser (Application Management Software – AMS) kezeli. Saját életciklussal rendelkeznek.

Minden MIDlet-nek tartalmaznia kell a `javax.microedition.midlet.MIDlet` osztály leszármazottját, illetve további osztályokat, melyekre a MIDlet-nek szüksége van. Az alkalmazásvezérlő hozza létre a MIDlet egy példányát, ezért a MIDlet nem rendelkezik `public static void main()` metódussal.

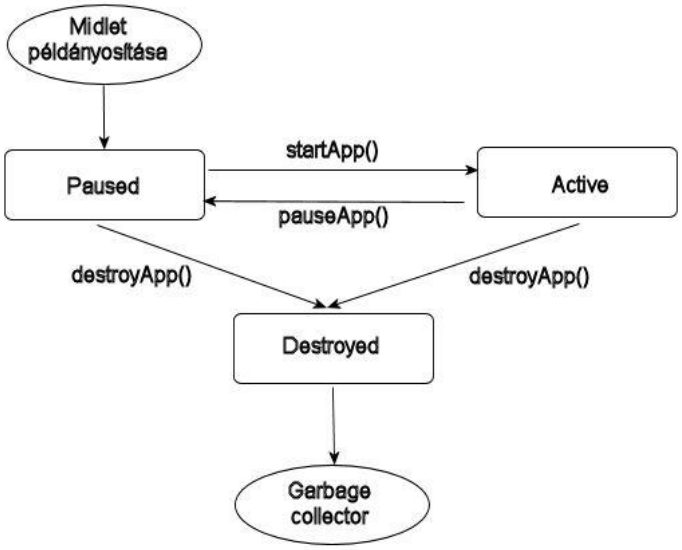
Egy üres MIDlet:

```
import javax.microedition.midlet.*;

public class uresMidlet extends MIDlet {

     public void startApp() {
     }
     pubilc void pauseApp() {
     }
     public void destroyApp(boolean unconditional) {
     }
```

```
}
```



Egy MIDlet három állapotban lehet:

- felfüggesztett (`Paused`)
- aktív (`Active`)
- leállított (`Destroyed`)

Alapesetben a MIDlet `Destroyed` állapotban van, ilyenkor az alkalmazás már feltelepült a mobilkészülékre, de még nem töltődött be a JVM-be. Amikor példányosítjuk a MIDlet-ünket, akkor `Paused` állapotba kerül. Ilyenkor a MIDlet már betöltődött a hivatkozott osztályokkal együtt a JVM-be, de az erőforrások lefoglalása még nem történt meg. `Active` állapotba akkor kerül az alkalmazás, ha meghívjuk a `startApp()` metódust. Létrejön a felhasználói felület, illetve felépülnek az esetleges kommunikációs kapcsolatok. Ha az alkalmazás futása közben szükséges állapotváltás, akkor az AMS meghívja a MIDlet `pauseApp()` metódusát, így minimálisra csökkentve az alkalmazás erőforrás-használatát. Újraindításához a `startApp()` metódus meghívása szükséges. Amikor bezárjuk az alkalmazást, akkor az AMS meghívja a MIDlet `destroyApp()` metódusát, így ismét `Destroyed` állapotba kerül a MIDlet.

A MIDleteket szabványos JAR (Java Archive Resource) csomagolják. Amennyiben több MIDlet van egy JAR fájlban, akkor őket MIDlet sorozatank nevezzük. Egy MIDlet a JAR

bármelyik osztályát elérheti. Az alkalmazásmenedzser a JAR fájlhoz tartozó leíró (manifest) alapján találja meg a JAR-ba csomagolt MIDleteket. A kezelésükhöz szükséges információk is itt találhatóak. A leírófájl kinézete, főbb információk:

- MIDlet-Name: a MIDlet készlet neve
- MIDlet-Version: a MIDlet készlet verziója
- MIDlet-Vendor: a MIDlet készlet forgalmazója
- MIDlet-Jar-URL: a Jar fájl honnan lett letöltve
- MIDlet-Jar-Size: a Jar fájl mérete
- MicroEdition-Profile: a MIDP verziója
- MicroEdition-Configuration: a CLDC verziója

3 A fejlesztéshez használt eszköz

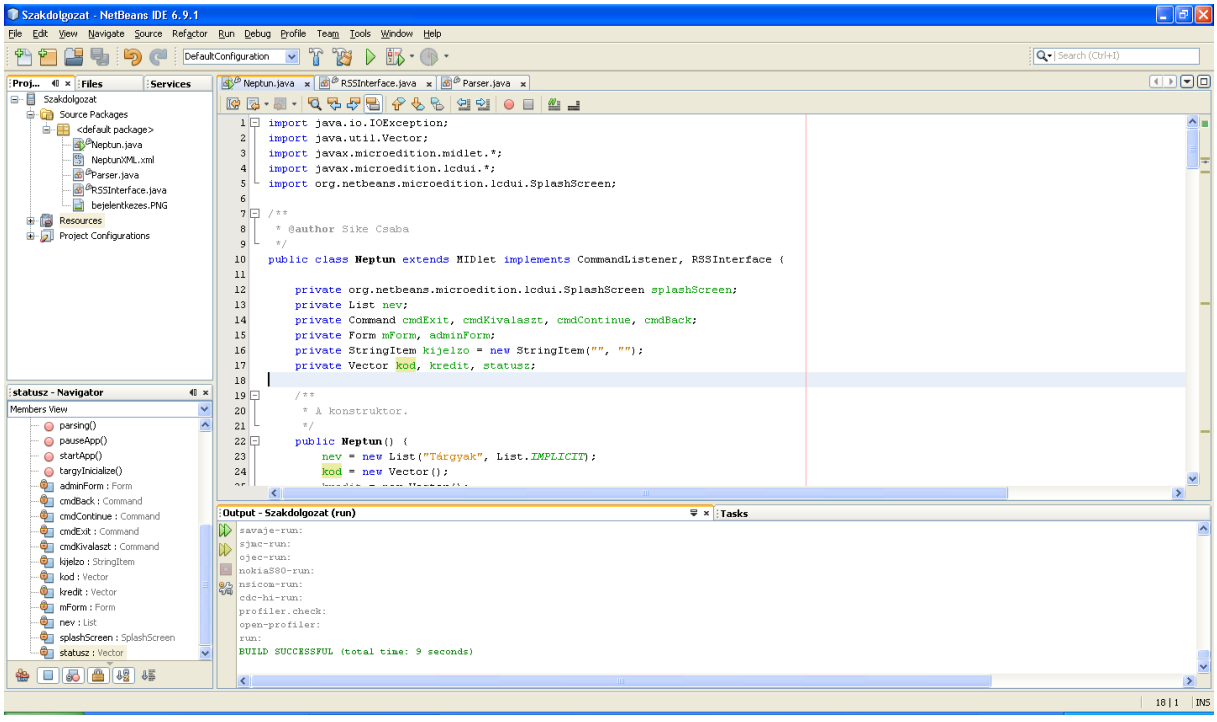
3.1 NetBeans IDE 6.9.1

A NetBeans egy java-ra alapuló platform független, integrált fejlesztői környezet, melynek segítségével a legtöbb ma használt programnyelven készíthetünk alkalmazásokat, komplett szoftvereket, kiegészítőket. A kezelésben nagyban segít a felhasználóbarát grafikus interface. Számos plugin-nel egészíthetjük ki, melyek tovább bővítik a felhasználás területeit, akár a népszerű Android környezetre is.

A NetBeans 6.9 újdonságai közé tartozik a JavaFX Composer, amivel JavaFX alkalmazások felhasználói felületét lehet elkészíteni. Fazonigazításon esett át a HTML- és CSS-szerkesztő is. Újdonság a Spring 3.0, a PHP Zend keretrendszer támogatása is.

Előnyei:

- ingyenes
- letölthető külön egyes programnyelvekre, vagy komplett csomagban a leggyakrabban használt nyelvekhez
- teljes körű support (tanulmányok, fejlesztői- felhasználói fórum, könyvek)
- folyamatos fejlesztés
- platform független



4 XML röviden, általánosan

4.1 Mi is az XML?

Az XML (Extensible Markup Language, *Kiterjeszthető Leíró Nyelv*) a W3C által ajánlott általános célú leíró nyelv, speciális célú leíró nyelvek létrehozására. Az SGML egyszerűsített részhalmaza, mely különböző adattípusok leírására képes. Az elsődleges célja strukturált szöveg és információ megosztása az Interneten keresztül. Az XML-en alapuló nyelvek (például RDF, RSS, MathML, XSIL, SVG) formális módon vannak leírva, így lehetővé téve a programok számára a dokumentumok módosítását és validálását a formátum előzetes ismerete nélkül.

4.2 A nyelv története

Az XML-t Tim Bray fejlesztette ki, miután az IBM, az Oxford University Press és a University of Waterloo által támogatott internetes szótáron dolgozott. Mivel hatalmas mennyiségű adatot kellett tárolni és feldolgozni, kereskedelmi szoftvermérnököket vontak be a projektbe, hogy megoldást találjanak az adatok indexelésére és tárolására. Az Association for Computing Machinery (ACM) számára adott interjú során Bray azt nyilatkozta, hogy bevonták a projektbe és megmutatták neki a szótár számára kifejlesztett belső struktúrát: "kis beágyazott címke határozta meg, hogy mi bejegyzés illetve szó, és aztán kiejtés, etimológia, rövid idézet, és aztán adat, forrás, szöveg és így tovább". Ez vált az XML elődévé. Miután kifejlesztették a technológiát a szótár projekthez, Bray megalapította az Open Text Corporation-t, kifejlesztett egy kereső motort, valamint meghívták a W3C-be, hogy legyen az XML specifikációjuk szerkesztője.

4.3 Az XML előnyei és hátrányai

Az XML azon tulajdonságai, amelyek alkalmassá teszik az adattovábbításra:

- mind ember, mind gép számára olvasható formátum
- támogatja az Unicode-ot, amely lehetővé teszi bármely információ bármely emberi nyelven történő közlését
- képes a legtöbb általános számítástudományi adatstruktúra ábrázolására (pl.: rekord, lista, fa, stb.)

- öndokumentáló formátum, amely struktúra- és mezőneveket ír le speciális értékekkel együtt
- szigorú szintaktikus és elemzési követelményeket támaszt, ami biztosítja, hogy a szükséges elemzési algoritmus egyszerű, hatékony és ellentmondásmentes maradjon

Az XML-t gyakran használják dokumentumtárolási és feldolgozási formátumként, mind online mind offline módon, és több előnnyel is jár:

- internetes szabványokon alapuló erőteljes, logikailag ellenőrizhető formátum
- a hierarchikus struktúrája megfelel a legtöbb (de nem mindegyik) dokumentum típusnak
- egyszerű szöveg formátumban valósul meg, licencektől és korlátozásoktól mentesen
- platform-független, így viszonylag immúnis a technológiai változásokkal szemben
- az XML-t, és elődjét, az SGML-t már több, mint tíz éve használják, így széles tapasztalat és eszközkészlet áll rendelkezésre

Bizonyos alkalmazások szempontjából a következő hátrányokkal rendelkezik:

- a szintakszisa elég bőbeszédű és részben redundáns. Ez nehezítheti az emberi olvashatóságot és az alkalmazások hatékonyságát, valamint nagyobb tárolási költséggel jár. Nehézzé teszi az XML alkalmazását korlátozott sávszélesség esetén, bár bizonyos esetekben a tömörítés csökkentheti a problémát. Ez részben igaz a telefonokon és PDA-kon futó multimédiás alkalmazásokra, melyek XML-t szeretnének használni képek és videók leírására.
- a szintaxis számos homályos, felesleges tulajdonsággal bír, ami az SGML hagyatéka
- az alapvető elemzési követelmények nem támogatják az adattípusok túl széles körét, így néha a kívánt adat kinyerése a dokumentumból plusz munkával jár az elemző részéről
- nincs lehetőség a dokumentum egyes részeinek közvetlen elérésére és frissítésére
- egymást részben átfedő (nem hierarchikus) adatstruktúrák modellezése külön erőfeszítést igényel
- az XML relációs és objektum orientált paradigmához kötése néha fáradságos

4.4 Az XML technológia a mobileszközökön

Az XML-állományok kapcsán az egyik legfontosabb feladat az állomány feldolgozása, az abban hordozott információ kinyerése. Az XML-állományok feldolgozására úgynevezett XML-elemzőket (parser) használnak. Ezeket nagy számításigény jellemzi, hiszen az XML-elemek felderítése egy XML-fájlban erőforrás-igényes feladat, amely megnehezíti az XML-feldolgozás alkalmazhatóságát korlátos számításkapacitású mobiltelefonokban. Fontos kérdés azonban a technológia mobiltelefonokon való alkalmazhatósága, mivel az XML révén akár komolyabb feladatok ellátására is alkalmasak lehetnek a mobilkészülékekre szánt alkalmazások. A JavaME-platformon különösen fontos kérdés az erőforrásigény, hiszen a virtuális gép képességei korlátozottak. A legfontosabb szempontok, amelyeket figyelembe kell venni összetett alkalmazások fejlesztésekor:

- kis számításkapacitás
- a hálózatelérés lassabb
- kevés rendelkezésre álló memória
- alacsony adatátviteli sebesség

Megállapítható tehát, hogy a JavaME-platformra készített XML-elemzőknek hatékony memória-felhasználásra és viszonylag alacsony számításigényre kell törekedniük.

4.5 Az XML feldolgozásának lehetőségei

Többféle módszer létezik az XML-állományok feldolgozására. Három különböző típusú feldolgozót fogok ismertetni:

- modellalapú feldolgozót
- push típusú feldolgozót
- pull típusú feldolgozót

Fontos kiemelni, hogy egy XML-állomány nagyobb méretű is lehet, ebből kifolyólag feldolgozása erőforrás-igényes művelet.

4.5.1 Modellalapú XML-feldolgozó

Működésük első lépésükként a modellalapú feldolgozók beolvassák a teljes XML-állományt, majd leképezik azt a memóriába. Ez a leképezés egy fajellegű struktúra. Mivel a leképezés elkészítése után már a memóriában van, ezért a különféle kérésekre (például keresés) már az elkészített modell alapján viszonylag gyorsan lehet válaszolni. A modellalapú feldolgozó legnagyobb hátránya a nagy memóriaigénye. Előnye viszont az, hogy a modell felépítése után az adatok gyorsan elérhetőek, és egy újabb lekérdezés esetén nem kell ismét felépíteni a modellt, ha az elfér a memóriában.

4.5.2 Push típusú feldolgozó

A push típusú XML-feldolgozók működésük során beolvassák az XML-állományt, és feldolgozás közben különféle eseményeket „generálnak” (például új paraméter, állomány vége, stb.). Ezeket az eseményeket a feldolgozó alkalmazásnak folyamatosan értelmezni és kezelni kell. A push típusú XML-feldolgozónak is hátránya, hogy viszonylag sok memóriát használnak az állomány beolvasása során. Előnyük viszont az egyszerű használatuk, amely az eseménykezelő jellegből adódik.

4.5.3 Pull típusú feldolgozó

A pull típusú XML-feldolgozók mindig csak egy kis részét olvassák be az XML-állományoknak. Tehát csak addig olvasnak egy lépésben, amíg egy újabb, még fel nem dolgozott XML-elemet nem találnak. Lényegében a pull típusú XML-feldolgozó lépésenként dolgozza fel az XML-állományt, és minden lépés után lekérdezhetjük az aktuálisan beolvasott elemet és annak tulajdonságait. A pull típusú elemző működése nem automatikus, nem küld folyamatos eseményeket az alkalmazásnak feldolgozásra, helyette az alkalmazás kér új adatokat a feldolgozóegységtől.

Összefoglalva láthatjuk, hogy a modellalapú és a push típusú XML-feldolgozók a memória- és számításigényük miatt sok esetben meghaladják a mobilkészülékek erőforrásait, ha egy viszonylag nagyobb méretű XML-állományt kell feldolgozniuk. A pull típusú XML-feldolgozóknál viszont ezt a hibát kiküszöbölték, így a választásom egy ilyen típusú XML-feldolgozóra esett.

5 A kXML ismertetése

A kXML egy pull típusú elemző, amelyet a MIDP-profil számára terveztek. Egyszerűen használható, stabil, az erőforrásokkal rendkívül takarékosan bánik, így ideális választás XML feldolgozására mobileszközökön. Az XML-állomány feldolgozása nem automatikusan történik, hanem az alkalmazás kérheti a feldolgozót az újabb lépés végrehajtására, miután sikeresen kezelte az előző lépést, így nincs állandó terhelés.

A kXML két legfontosabb osztálya az `XMLParser` és a `ParseEvent`. Előbbi gyakorlatilag a pull típusú XML-elemzőt, míg utóbbi a feldolgozás során a lépések eredményét szimbolizálja. Az `XMLParser` létrehozásához szükséges egy `Reader` objektum létrehozása, amely a bemeneti XML-állomány tartalmának olvasásáért felelős.

```
HttpConnection hc = (HttpConnection)Connector.open(url);
Reader reader = new InputStreamReader(hc);
XmlParser parser = new XmlParser(reader);
```

Az XMLParser-objektum `skip()` és `read()` függvénye segítségével lépésenként dolgozzuk fel az XML-állományt. A `read()` függvény visszatérési értéke egy `ParseEvent` objektum, amelytől lekérdezhető az esemény típusa és a lépés során beolvasott elem különféle tulajdonságai. Az esemény típusa a `public int getType()` függvény segítségével kérdezhető le. A lehetséges visszatérési értékek a következők:

- `org.kxml.Xml.START_TAG`
- `org.kxml.Xml.END_TAG`
- `org.kxml.Xml.TEXT`
- `org.kxml.Xml.WHITESPACE`
- `org.kxml.Xml.PROCESSING_INSTRUCTION`
- `org.kxml.Xml.COMMENT`
- `org.kxml.Xml.DOCTYPE`
- `org.kxml.Xml.END_DOCUMENT`

5.1 RSS-feldolgozás kXML segítségével

Az RSS (Really Simple Syndication) webes együttműködésre szolgáló XML-állományformátumok családja. Jelen példában egy olyan RSS tartalmát szeretnénk feldolgozni, amit bejelentkezés után egy HTTP-kapcsolaton keresztül ér el a feldolgozó. A dokumentum gyökéreleme az `rss`, amely a `targy` elemeket tartalmazza. Ezek hordozzák számunkra a fontos információkat, mégpedig `kod`-, `nev`-, `kredit`- és `statusz`-alelemeket. Ebben az XML-állományban egy programtervező-informatikus hallgató első féléves tárgyai találhatóak meg.

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="0.91">
   <targy>
        <kod>INAK101E</kod>
        <nev>Diszkret matematika 1</nev>
        <kredit>5</kredit>
        <statusz>teljesitett</statusz>
   </targy>
[…]
   </targy>
   <targy>
        <kod>INDK711E</kod>
        <nev>Szamitogep architechturak</nev>
        <kredit>5</kredit>
        <statusz>teljesitett</statusz>
    </targy>
</rss>
```

6 A szoftver ismertetése

Az általam választott példában egy képzeletbeli mobiltelefonos Neptun-rendszerben történik meg az XML-dokumentumok kezelése. A példa nem ismerteti a távoli szerver

folyamatokat, kizárólagosan a HTTP-kapcsolaton beérkező XML-dokumentum mobileszközön történő Java alapú kezelésére tér ki, valamint bemutat néhány JavaME eszközt is.

6.1 A szoftver felépítése

A program a `Neptun.java` és az `Parser.java` osztályokból, valamint az `RSSInterface.java` interfészből tevődik össze. A `Neptun` osztály értelemszerűen egy MIDlet. A szoftver a beépített emulátorra lett optimalizálva.

6.1.1 A Neptun.java elemzése

A `Neptun.java` osztály felelős a mobileszköz képernyőjén való megjelenítésért, valamint az egyes magas szintű események kezeléséért. Az osztályt az `import` utasítással kezdjük.

```
import java.io.IOException;
import java.util.Vector;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import org.netbeans.microedition.lcdui.SplashScreen;
```

Az egyes csomagok szükségessége:

- `java.io.IOException` – sikertelen vagy félbeszakított input/output műveletek váltják ki.
- `java.util.Vector` – dinamikus tömb osztály.
- `javax.microedition.midlet.*` - ez a csomag tartalmazza a `MIDlet` osztályt. Három absztrakt metódussal rendelkezik, nevezetesen `startApp()`, `pauseApp()` és `destroyApp()`. Ezeket a MIDlet-nek mindig implementálnia kell. A MIDlet ezeken keresztül kap értesítést az állapotváltozásokról.
- `javax.microedition.lcdui.*` - ez a csomag az alkalmazásokban létrehozható felhasználói felületek készítésére és vezérlésére tartalmaz osztályokat.
- `org.netbeans.microedition.lcdui.SplashScreen` – ez a csomag tartalmazza a SplashScreen teljes képernyős induló képernyő osztályt.

A `Neptun` osztályunk két interfészt implementál, az egyik a már említett `RSSInterface`, a másik a `CommandListener` interfész, amelynek segítségével magas szintű eseményeket fogadhat alkalmazásunk. Ehhez a

```
void commandAction(Command c, Displayable d)
```

metódust kell implementálnunk. Ebben kezeljük le az eseményeket. Minden esemény bekövetkezésekor lefut.

A `Neptun` osztály kiterjeszti a `MIDlet` osztályt, amelyet minden MIDlet-nek ki kell terjesztenie.

```
public class Neptun extends MIDlet implements CommandListener,
RSSInterface {

    private org.netbeans.microedition.lcdui.SplashScreen
    splashScreen;
    private List nev;
    private Command cmdExit, cmdKivalaszt, cmdContinue,
    cmdBack;
    private Form mForm, adminForm;
    private StringItem kijelzo = new StringItem("", "");
    private Vector kod, kredit, statusz;
```

Az adattagok ismertetése:

- `org.netbeans.microedition.lcdui.SplashScreen splashScreen` – teljes képernyős induló képernyő.
- `List nev` – a List olyan esetekben használatos, amikor a mobilkészülék kijelzőjén egy változó elemszámú listát szeretnénk megjeleníteni.
- `Command cmdExit, cmdKivalaszt, cmdContinue, cmdBack` – események, amelyek a programban történő navigálást segítik.
- `Form mForm, adminForm` – a `Form` összetett képernyőtartalom, különféle felületelemek helyezhetők el rá (táblázat, kép, rövidebb beviteli mezők stb.)

- `StringItem kijelzo` – a StringItem felületi elem szöveges adatok megjelenítésére használatos. Jellegzetessége, hogy két szöveges érték (`Label`, `Text`) is található rajta, amelyeket külön-külön is lekérdezhetünk és beállíthatunk.
- `Vector kod, kredit, statusz` – dinamikus tömbök, a beolvasott tárgyak adatait ezekben tároljuk.

Következhet a konstruktor:

```
public Neptun() {
        nev = new List("Tárgyak", List.IMPLICIT);
        kod = new Vector();
        kredit = new Vector();
        statusz = new Vector();
    }
```

A konstruktorban inicializáljuk a `List` típusú `nev` objektumunkat. Beállítjuk ennek a teljes képernyőlistának a nevét és a típusát. A típus 3 féle lehet:

- `IMPLICIT` – felsorolás jellegű.
- `EXCLUSIVE` – egyválasztásos.
- `MULTIPLE` – többválasztásos.

A nev listánk jeleníti meg az XML-állományból nyert tárgyak neveit, melyekből választani lehet. Mivel jelen listánk IMPLICIT típusú, ezért a kiválasztott elem indexének lekérdezése a következő módon történik:

```
int selectedIndex = nev.getSelectedIndex()
```

A konstruktorban példányosítjuk még a három Vector típusú objektumunkat (kod, kredit, statusz).

```
public void intro() {
    try {
        splashScreen = new SplashScreen(getDisplay());
        splashScreen.setCommandListener(this);
        splashScreen.setTitle("Neptun bejelentkezés");
        splashScreen.setImage(Image.createImage(
"/bejelentkezes.PNG"));
        splashScreen.setTimeout(6000);
        splashScreen.setAllowTimeoutInterrupt(true);
        splashScreen.setFullScreenMode(true);
    }
    catch (IOException e) {
    }
    getDisplay().setCurrent(splashScreen);
}
```

Ebben a metódusban állítjuk be a splashScreen objektumot. Legelőször példányosítjuk, majd meghatározzuk a nevét, a megjelenítendő képet. Megjelenítés előtt beállítjuk a parancskezelő objektumot a setCommandListener(…) függvényen keresztül. A setTimeout(6000) függvény segítségével meghatározzuk a kép megjelenítésének az idejét. Jelen esetben ez 6000 ms. Ennek a TimeOut értéknek a lejártakor automatikusan egy DISMISS_COMMAND keletkezik. Ezt az eseményt a lekezeljük a commandAction(…) parancskezelőben.

A felhasználónak szeretnénk biztosítani, hogy ne kelljen mindig kivárniuk a TimeOut leteltét, ezért meghívjuk a setAllowTimeoutInterrupt(true) függvényt.

Ekkor, ha a felhasználó bármilyen billentyűt üt le, automatikusan egy, az adott `SplashScreen`-hez tartozó `DISMISS_COMMAND` keletkezik.

Mivel azt szeretnénk, hogy a `splashScreen`-ünk teljes képernyős módban jelenjen meg, ezért meghívjuk a `setFullScreenMode(true)` függvényt.

Kötelező kivételkezelést használni, mivel a megjelenítendő kép megnyitásakor kivételek váltódhatnak ki. Végül a `getDisplay().setcurrent(splashScreen)` függvény meghívásával megjelenítjük a képernyőn.

Ha már szóba került a `getDisplay()` függvény, vessünk rá egy pillantást:

```
public Display getDisplay() {
        return Display.getDisplay(this);
}
```

A Java ME platformon a `Display`-objektum felelős a képernyő és a beviteli eszközök kezeléséért. Minden `MIDlet`-hez pontosan egy ilyen objektum tartozik, amelyre a referenciát a `Display`-osztály statikus `getDisplay(MIDlet myMIDlet)` függvényével kérhetjük le. A függvény paraméterében meg kell adnunk a `MIDlet`-objektumot, amelyhez el szeretnénk kérni a `Display`-t. Ezt a függvényt a NetBeans minden `MIDlet`-hez automatikusan generálja, így könnyen elérhetjük a szükséges `Display` objektumot.

```
public void admin() {
    Spacer spacer = new Spacer(10, 20);
```

A következő metódusunk egy Spacer-objektummal kezdődik. Ennek a felületi elemnek a segítségével a felületek közti távolságot állíthatjuk be megkönnyítve egy Form áttekinthetőségét. A Spacer konstruktorának első paraméterében az elem szélessége állítható. A második paraméterében adhatjuk meg, hogy mekkora helyet hagyjon üresen a két elem között.

```
    adminForm = new Form("Bejelentkezés");
    TextField userName = new TextField("Felhasználói név:",
null, 40, TextField.ANY);
    TextField password = new TextField("Jelszó", null, 40,
TextField.PASSWORD);
```

Folytatva az `admin()` metódus elemzését az `adminForm` `Form`-objektumunk példányosítása következik. A `Form` képernyő a leggyakrabban használt képernyőtípus, különféle felhasználói felületelemek helyezhetők el rajta. Példányosításnál megadjuk a `Form`-objektumunk nevét.

Elhelyezünk az `adminForm` felületünkön két `TextField` mezőt. A `TextField` egy egyszerű szövegbeviteli mező, amelyen keresztül a felhasználók rövidebb adatokat tudnak megadni. A `TextField` konstruktorában megadhatjuk a címét, szövegét, maximális beírható karakterek számát, illetve a beolvasandó tartalomra vonatkozó korlátozásokat.

```
    cmdExit = new Command("Kilépés", Command.SCREEN, 1);
    cmdContinue = new Command("Bejelentkezés", Command.SCREEN,
1);
```

Következik két parancs, azaz két `Command`-objektum példányosítása. Beállítjuk a kijelzőn megjelenő nevét, a hozzájuk rendelt parancstípust, és a prioritást. A `Command`-típusú objektumok tartalmazzák a magas szintű események tulajdonságait.

```
    adminForm.addCommand(cmdExit);
    adminForm.addCommand(cmdContinue);
    adminForm.setCommandListener((CommandListener)this);
    adminForm.append(spacer);
    adminForm.append(userName);
```

```
        adminForm.append(new Spacer(10, 15));
        adminForm.append(password);
    }
```

A metódus hátralevő részében hozzárendeljük az adminForm felületünkhöz az előbb tárgyalt parancsokat, beállítjuk hozzá a parancskezelő objektumot. Legvégül elhelyezzük rajta a különböző felhasználói felületelemeket.


```
public void mainForm() {
    Screen waitScreen = new Form("Kapcsolódás...");
    getDisplay().setCurrent(waitScreen);
    targyInicialize();
    parsing();
    getDisplay().setCurrent(nev);
}
```

A metódust egy `waitScreen` nevű `Screen`-objektummal kezdjük. Ezt azért használjuk, mert ha a mobilalkalmazás háttérben valamilyen összetettebb feladatot végez, például jelen esetben hálózati kommunikációra várakozik, akkor szükség lehet rá, hogy ezt a felhasználónak valamilyen módon jelezzük. Miután megjelenítjük a képernyőn két metódushívást hajtunk végre. Ezek a metódusok a továbbiakban lesznek kifejtve. Elöljáróban annyi elmondható, hogy e két metódus által beállított `nev List`-objektumunk kerül megjelenítésre a metódus végén.

```
public void targyInicialize() {
    cmdKivalaszt = new Command("Kiválaszt", Command.SCREEN,
1);
    nev.addCommand(cmdExit);
    nev.addCommand(cmdKivalaszt);
    nev.setCommandListener(this);
}
```

A `targyInicialize()` metódus egyszerű szerepet lát el. Először példányosítja a `cmdKivalaszt Command`-objektumunkat. A példányosítás után két commandot hozzárendel a `nev` objektumhoz (a `cmdExit`-et már korábban példányosítottuk a bejelentkező felületnél, most újra felhasználjuk). Legvégül beállítjuk hozzá a parancskezelő objektumot.

```
public void parsing() {
    Parser parser = new Parser();
    parser.setRSSInterface(this);
    parser.parse("http://users.atw.hu/mobilszakdoga/
    NeptunXML.xml?atw_rnd=1312692958");
}
```

Meghívjuk a feldolgozót, majd a bemeneti XML-állomány elérhetőségét beállítjuk.

```
public void commandAction(Command command, Displayable screen)
{
    if (command == cmdExit)
        notifyDestroyed();
    else if (command == cmdContinue) {
        mainForm();
    }
    else if (command == cmdBack) {
        mForm.deleteAll();
        getDisplay().setCurrent(nev);
    }
```

```
    else if (command == cmdKivalaszt) {
        listaForm(nev.getSelectedIndex());
    }
    else if (screen == splashScreen) {
        if (command == SplashScreen.DISMISS_COMMAND)
            getDisplay().setCurrent(adminForm);
    }
}
```

A `commandAction()` osztály funkcionalitását már tárgyaltuk, jelen helyzetben röviden ismertetve lesz, hogy az egyes események bekövetkezésekor milyen utasításokat kell végrehajtani. A `cmdExit` eseménynél a már ismertetett `notifyDestroyed()` metódus kerül meghívásra, befejezzük a program futását. A `cmdContinue` esemény a bejelentkező felületnél található meg, bekövetkezésekor továbblépünk az kapcsolódás felületre a már tárgyalt `mainForm()` metódus meghívásával. A következő eseményünk a `cmdBack`. Ekkor az `mForm` From-objektumunk tartalmát töröljük, majd a képernyőn a `getDisplay().setCurrent(nev)` hívással megjelenítjük a tárgyak listáját. A `cmdKivalaszt` esemény bekövetkezésekor meghívjuk a pár oldallal feljebb tárgyalt `listaForm()` metódust. A `nev.getSelectedIndex()` visszatérési értéke megadja a kiválasztott tárgy indexét a listában. Végül az osztály elején tárgyalt `splashScreen` objektumunkhoz tartozó eseményt is lekezeljük. Ennek bekövetkezésekor (ami az időlimit lejárta, vagy egy gomb megnyomása) az `adminForm` Form-objektumunkat jelenítjük meg.

```
public void itemParsed(String sKod, String sNev, String
sKredit, String sStatusz) {
    nev.append(sNev, null);
    kod.addElement(sKod);
    kredit.addElement(sKredit);
    statusz.addElement(sStatusz);
}
```

Ez a metódus az `RSSInterface` interfészen keresztül kerül meghívásra az `Parser` osztályból. Feladata, hogy a paraméterként kapott négy String-elemet beillessze a megfelelő

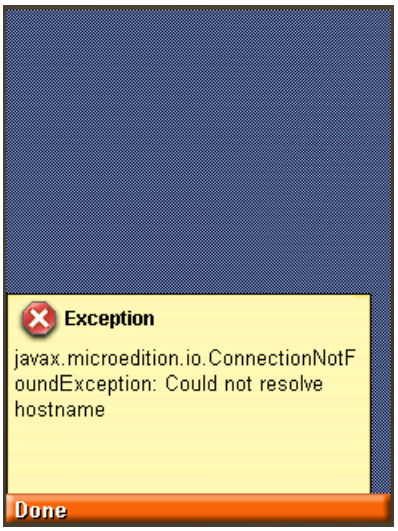
listába, vagy vektorokba. A vektorba illesztés egyszerűen érthető, a `nev` List-objektumba illesztésnél az első paraméter a beillesztendő String-elem, a második pedig az esetlegesen az elemhez illesztendő (és kiíratáskor a listában megjelenítendő) kép elem lenne. Jelen esetben itt `null` értéket állítunk be, mivel nem kívánunk képet illeszteni a String-elemünkhöz.

```
public void exception(java.io.IOException ioe) {
    Alert a = new Alert("Exception", ioe.toString(), null,
AlertType.ERROR);
    a.setTimeout(Alert.FOREVER);
    getDisplay().setCurrent(a);
}
```

A `Neptun` osztály utolsó metódusa. Szintén az `RSSInterface` interfészen keresztül kerül meghívásra az `Parser` osztályból. Az esetlegesen bekövetkező kivételeket jeleníti meg a képernyőn. A megjelenítendő kivételt paraméterként kapja.
A megjelenítésért egy figyelmeztető képernyő a felelős. Az Alert képernyő feladata, hogy tájékoztassa a felhasználót a különféle eseményekről. Ez a különböző gyártók készülékein más-más módon jelenhet meg. Az Alert felület megtervezésekor csak pár dolgot adhatunk meg. Az `a` Alert-objektumunk példányosításánál a következő dolgokat állíthatjuk be:

- az Alert címe (Exception)
- a megjelenítendő szöveg (`ioe.toString()`)
- egy megjelenítendő képet (jelen esetben nem jelenítünk meg képet, tehát `null`-értéket állítunk be)
- az Alert típusát (`AlertType.ERROR` – hiba jelzése)

Fontos beállítani az Alert esetében, hogy a figyelmeztető ablak mennyi idő múlva tűnjön el automatikusan. Jelen esetben ezt az értéket `Alert.FOREVER`-re állítjuk, ezzel biztosítva azt, hogy a figyelmeztetés csak akkor tűnjön el, ha azt a felhasználó tudomásul vette. Ezt a `Done` parancson keresztül jelzi.



Az első Alert-képernyő akkor jelenik meg, amikor a bejelentkezésnél nincsen hálózati kapcsolat, így nem lehetséges létrehozni HttpConnection-t. A második Alert-képernyőnél pedig nem megfelelő formátumú az XML-fájl elérési útvonala.

6.1.2 Az RSSInterface.java elemzése

```
public interface RSSInterface {
    void itemParsed(String kod, String nev, String kredit,
String statusz);
    void exception(java.io.IOException ioe);
}
```

Jelen interfész mindössze két metódust tartalmaz, az `itemParsed`-et és az `exception`-t. Ezeket a `Neptun.java` osztályban implementáljuk. Ismertetésük fentebb látható.

6.1.3 Az Parser.java elemzése

Az `Parser.java` osztály felelős a HTTP kapcsolat felépítéséért és a beolvasott XML-dokumentum feldolgozásáért. Az osztályt itt is az `import` utasításokkal kezdjük:

```
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;
import javax.microedition.io.Connector;
```

```
import javax.microedition.io.HttpConnection;
import org.kxml.Xml;
import org.kxml.parser.ParseEvent;
import org.kxml.parser.XmlParser;
```

Az egyes csomagok szükségessége:

- `java.io.IOException` – sikertelen vagy félbeszakított input/output műveletek váltják ki.
- `java.io.InputStream` – minden bemeneti bájtcsatorna ősosztálya.
- `java.io.InputStreamReader` – olyan bemeneti karaktercsatorna, mely az adatokat a konstruktorban megadott bájtcsatornáról veszi.
- `java.io.Reader` – a bemeneti karaktercsatornák (absztrakt) ősosztálya. Az `InputStream` megfelelője a karaktercsatornáknál.
- `javax.microedition.io.Connector` – az osztály segítségével hozunk létre új `Connection` objektumokat.
- `javax.microedition.io.HttpConnection` – ez az interfész definiálja egy HTTP kapcsolathoz szükséges metódusokat és konstansokat.
- `org.kxml.Xml` – ez az osztály tartalmaz néhány statikus xml metódust, főleg speciális karakterek kihagyására, mint például a hegyes zárójelek és idézőjelek.
- `org.kxml.parser.ParseEvent` – minden parser eseménynek egy absztrakt szuperosztálya.
- `org.kxml.parser.XmlParser` – egyszerű pull típusú XML-feldolgozó.

```
public class Parser {

    protected RSSInterface rssInterface;
```

Az osztály egyetlen adattagot implementál, az `rssInterface`-t.

```
public void setRSSInterface(RSSInterface rss) {
    rssInterface = rss;
}
```

Következő metódusunk a `connection()`, amely paraméterként megkapja az XML-dokumentum elérhetőségét:

```
public void connection(final String url) {
    Thread t = new Thread() {
        public void run() {
            HttpConnection hc = null;

            try {
                hc = (HttpConnection)Connector.open(url);
                parse(hc.openInputStream());
            }
            catch (IOException ioe) {
                rssInterface.exception(ioe);
            }
            finally {
                try { if (hc != null) hc.close(); }
                catch (IOException ignored) {
                    rssInterface.exception(ignored);
                }
            }
        }
    };
    t.start();
}
```

Ezzel el is kezdjük az XML-dokumentum feldolgozását. A feldolgozást külön szálban folytatjuk, melynek `run()` metódusában a `HttpConnection` osztály segítségével létrehozzuk a kapcsolatot, majd megnyitjuk az `url` String-ben kapott elérhetőség alapján az XML-állományt. Ha sikeres a kapcsolat, akkor InputStream-en keresztül továbbítjuk az adatokat az osztály `parse()` metódusának, majd lezárjuk a kapcsolatot. Ha valamilyen okból kivétel keletkezik, azt lekezeljük, és a kivétel szövegét átadjuk az a `Neptun` osztálynak a megfelelő `rssInterface.exception()` metódushívással.

```
public void parse(InputStream in) throws IOException {
```

Az osztályunk utolsó metódusa, a parse(). Ebben valósul meg ténylegesen az adatok kezelése. Első lépésként létrehozzuk az elemzőnket az alábbi módon:

```
     Reader reader = new InputStreamReader(in);
    XmlParser parser = new XmlParser(reader);
    ParseEvent pe = null;
```

Most már csak a `parser` példányt szükséges használni a feldolgozáshoz.

```
        parser.skip();
        parser.read(Xml.START_TAG, null, "rss");

        boolean trucking = true;
        String kod, nev, kredit, statusz;
```

Az XML-állomány első sorát kihagyjuk, majd a második sort beolvassuk. Ezek számunkra nem tartalmaznak használható információt, ezért nem is foglalkozunk velük. A beolvasott hasznos adatokat négy `String`-ben tároljuk le. Létrehozunk egy `trucking` nevű `boolean` változót, értékét `true`-ra állítjuk. Addig fut a feldolgozás, amíg a `trucking` változó értéke `false` nem lesz:

```
while (trucking) {
    pe = parser.read();
```

Beolvassuk a következő sort a `parser.read()` hívással, eredménye egy `ParseEvent` objektum lesz.

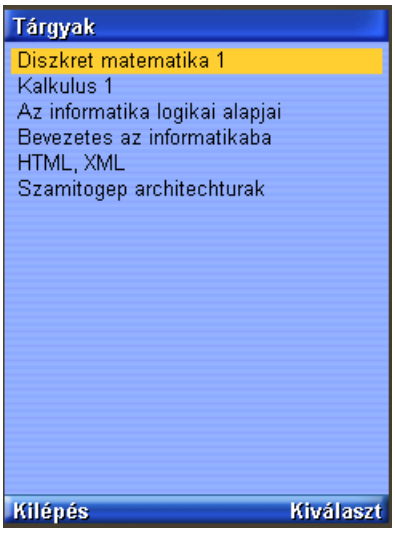
```
kod = nev = kredit = statusz = null;
while ((pe.getType() != Xml.END_TAG) ||
(pe.getName().equals(name) == false)) {
    pe = parser.read();
    if (pe.getType() == Xml.START_TAG &&
pe.getName().equals("kod")) {
        pe = parser.read();
```

```
        kod = pe.getText();
    }
    else if (pe.getType() == Xml.START_TAG &&
pe.getName().equals("nev")) {
        pe = parser.read();
        nev = pe.getText();
    }
    else if (pe.getType() == Xml.START_TAG &&
pe.getName().equals("kredit")) {
        pe = parser.read();
        kredit = pe.getText();
    }
    else if (pe.getType() == Xml.START_TAG &&
pe.getName().equals("statusz")) {
        pe = parser.read();
        statusz = pe.getText();
    }
}
rssInterface.itemParsed(kod, nev, kredit, statusz);
```

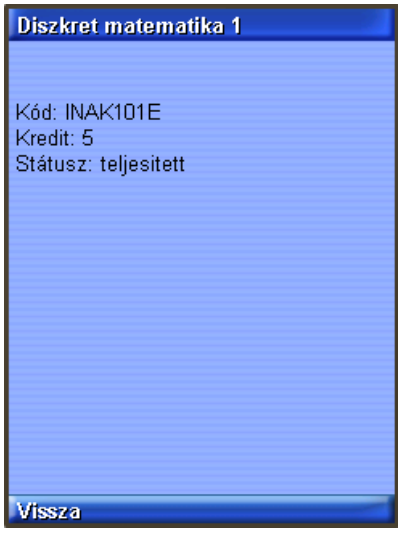
Ha a `pe ParseEvent` objektumunk kezdőelem, akkor a megfelelő `String`-be bemásoljuk belőle a hasznos információt. Miután megvan a kód, név, kredit és státusz információnk az adott tárgyról, az `rssInterface` objektumon keresztül meghívjuk a `Neptun.java` osztály itemParsed() metódusát a kinyert információkkal.

```
if (pe.getType() == Xml.END_TAG && pe.getName().equals("rss"))
{
    trucking = false;
}
```

Végezetül meghatározzuk a kilépési feltételt. Ha a `pe` objektumunk típusa `Xml.END_TAG` és a `getName()` hívással visszakapott `String` egyenlő az „`rss`” `String`-gel, akkor elértünk az XML-dokumentumunk végéhez.
Ezzel be is fejeztük az XML-dokumentum feldolgozását.



Tárgyak listája



A kiválasztott tárgy adatai

Összegzés

Szakdolgozatom választott főtémája az XML-dokumentumok kezelése mobilkörnyezetben, JavaME platformon. Azért tartottam fontosnak ezt a témát, mert ki lehet jelenteni, korszakváltás küszöbén állunk, a személyi számítógépek szerepét egyre jobban átveszik a mobileszközök, az XML-állományok pedig a legjobban használható kommunikációs dokumentumok. Egyszerűek, könnyen értelmezhetőek, gyorsan feldolgozhatóak. Mindazonáltal sok információt képesek hordozni, ember és gép számára egyaránt olvashatóak.

Mivel nem csak az XML-dokumentumok kezelése volt a fő cél, ezért sikerült bemutatni főleg a feldolgozott adatok megjelenítése során alkalmazott JavaME eszközöket, mint a SplashScreen, a Form, a List, a Command, a Spacer, a TextField, a Screen és az Alert. Ezeken kívül a szakdolgozat olvasója végigkísérheti, hogyan kell kifejleszteni egy midlet alkalmazást, hogyan kell lekezelni a különböző eseményeket, kivételeket. Bemutatásra került a HTTP kommunikáció felépítése és folyamata is.

Dolgozatom során nem törekedtem a szerver oldal elkészítésére, mivel ez független a JavaME környezettől. Ezért az XML-dokumentum közvetlen linken keresztül, de HTTP kapcsolat segítségével kerül feldolgozásra.

Remélem a dolgozat megfelelő alapot szolgál azoknak az olvasóknak is, akik nem XML-kezelés témában akarnak mobileszközre fejleszteni, és találnak hasznos információkat, eszközöket a sorok között, kedvet kapva mobileszközökre való fejlesztésre.

Irodalomjegyzék

- Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba, 2008, Szak Kiadó
- http://developers.sun.com/mobility/midp/articles/parsingxml/
- http://java.sun.com/products/midp/whatsnew.html
- http://netbeans.org/
- http://kxml.objectweb.org/software/downloads/
- http://kxml.objectweb.org/software/documentation/apidocs/
- http://www.oracle.com/technetwork/java/javame/tech/index.html
- http://pallergabor.uw.hu/hu/java-app/

Plágium - Nyilatkozat

Szakdolgozat készítésére vonatkozó szabályok betartásáról nyilatkozat

Alulírott (Neptunkód: ……) jelen nyilatkozat aláírásával kijelentem, hogy a

……..

című szakdolgozat/diplomamunka

(a továbbiakban: dolgozat) önálló munkám, a dolgozat készítése során betartottam a szerzői jogról szóló 1999. évi LXXVI. tv. szabályait, valamint az egyetem által előírt, a dolgozat készítésére vonatkozó szabályokat, különösen a hivatkozások és idézések tekintetében.

Kijelentem továbbá, hogy a dolgozat készítése során az önálló munka kitétel tekintetében a konzulenst, illetve a feladatot kiadó oktatót nem tévesztettem meg.

Jelen nyilatkozat aláírásával tudomásul veszem, hogy amennyiben bizonyítható, hogy a dolgozatot nem magam készítettem vagy a dolgozattal kapcsolatban szerzői jogsértés ténye merül fel, a Debreceni Egyetem megtagadja a dolgozat befogadását és ellenem fegyelmi eljárást indíthat.

A dolgozat befogadásának megtagadása és a fegyelmi eljárás indítása nem érinti a szerzői jogsértés miatti egyéb (polgári jogi, szabálysértési jogi, büntetőjogi) jogkövetkezményeket.

hallgató

Debrecen,