Sike Sándor

Mobiltelefonok programozása Java nyelven

A tananyag kidolgozását támogatta a Nokia Siemens Networks Kft.

Tartalomjegyzék

1. Bevezetés

A mobiltelefonok fejlődése lehetővé tette, hogy azokat ne csak telefonálásra, üzenetek küldésére használjuk, hanem különböző alkalmazásokat futtathassunk rajtuk. Ilyen szolgáltatások közé tartoznak multimédiás anyagok lejátszása, azok kezelése; adatkommunikációs alkalmazások telepítése, használata; játékok. Ehhez természetesen a megfelelő programokat el kell készíteni, és telepíteni kell a mobiltelefonra.

Mobileszközökre szánt alkalmazások fejlesztésének egy lehetséges, platformfüggetlen eszköze a Java. Ez akkor használható, ha az adott eszközön futtatható Java virtuális gép. A mobiltelefonok esetén figyelembe kell venni, hogy az erőforrások jelentősen korlátozottak, ezért ezekben az esetekben a teljes Java SE szolgáltatásait elvárni irreális. Ennek megfelelően létrehoztak egy speciális Java platformot, amely a Java ME (Micro Edition) nevet kapta. A jelenleg forgalomba kerülő, nem túl egyszerű mobiltelefonokon rendelkezésre áll a Java ME virtuális gép, és lehetőséget adnak ilyen programok telepítésére is.

1.1. Korlátok

Mielőtt a Java ME részleteivel foglalkoznánk, vizsgáljuk meg, hogy mik azok a korlátozások mobiltelefonokon, amit figyelembe kell vennünk a fejlesztés során.

Korlátozott energiaellátás: Az akkumulátor kapacitása véges, előbb-utóbb lemerül. Bizonyos műveletek különösen sok energiát igényelnek: hálózati kapcsolat, kommunikáció, kijelző megvilágítás, és esetenként a processzor.

Kisméretű kijelző: Mobiltelefonoknál a 240x320 pixelnél nagyobb felbontás még nagyon ritka, hétköznapi készülékekre nem jellemző.

Speciális adatbevitel: Az úgynevezett „okostelefonok” kivételével nincs QWERTY billentyűzet, azaz speciális módon lehet szöveget bevinni. Az érintőképernyős telefonoktól eltekintve, nincs mutató eszköz (egér).

Kicsi memória: A programok futtatására használható memória (nem háttértár) mérete erősen korlátozott. (128 MB már jó készüléket jelent.)

Gyengébb processzor: A processzor sebessége jóval elmarad a PC-kben található processzorokétól.

1.2. Java ME konfigurációk

A Java ME a legelterjedtebb mobil platform, amint már írtuk a legtöbb mobiltelefon támogatja valamelyik konfigurációját és profilját. Ezek a konfigurációk és profilok nem teljesen egységesek, ezért a fejlesztés előtt pontosan meg kell határozni a célplatformot. A következőkben ezeket tekintjük át röviden.

1.2.1. CLDC és CDC

Jelenleg két Java ME konfiguráció létezik: a **CLDC** (Connected Limited Device Configuration) és a **CDC** (Connected Device Configuration). A kettő közötti lényeges eltérést az adott készülék hardverével szemben támasztott követelmények adják.

A CDC konfiguráció a lehető legjobban igyekszik megtartani a Java SE-vel a kompatibilitást, csak optimalizálják az osztálykönyvtárakat kevesebb memóriára. Ez a konfiguráció nagyobb mobileszközökön (min. PDA) használható, hiszen a szükséges memória még így is több MB.

A CLDC konfiguráció figyelembe veszi a mobiltelefonok kisebb memóriáját (száz kB a megkövetelt memória nagyságrendje), és jelentős eltérések is megtalálhatóak benne a Java SE-hez képest. Ezen belül két változat van: az 1.0 az alap, amelyet a mobiltelefonok fejlődésével kibővítettek és létrehozták az 1.1-es változatot, amely egy kicsivel többet tud (pl.: lebegőpontos számok). A két konfiguráció közötti eltérés azonban elhanyagolható a profilok közötti eltérésekhez képest.

1.2.2. MIDP

Mobileszközökön a Java futtatókörnyezet (JRE) másik összetevője a CLDC mellett a MIDP (Mobile Information Device Profile). Ennek első változatát, az 1.0 verziót, a régebbi mobiltelefonok lehetőségeit figyelembe véve alakították ki. A hardver fejlődése és a felmerülő igények miatt ezt jelentősen továbbfejlesztették, és így jelent meg a 2.0 verzió. Ebben az első változathoz képest jelentősen bővültek a felhasználói felülethez kapcsolódó lehetőségek, és új funkciók (pl.: fájlkezelés) is bekerültek. Jelenleg a 2.1 változat a legújabb, de ez már nem tér el jelentősen a 2.0 verziótól. (Már dolgoznak a 3.0 kialakításán.) Szerencsére akárcsak a CLDC esetén a MIDP esetén is igaz, hogy egy későbbi verzió kompatibilis bármelyik előző verzióval, azaz korábbi verziójú programok magasabb verziójú környezetben futtathatóak.

További funkciók elérését teszik lehetővé az úgynevezett JSR (Java Specification Request) elemek. Például a MIDP változatok közötti különbségek egy része is a támogatott JSR-ek közötti eltérésekre vezethetőek vissza.

1.3. Java alkalmazások telepítése

Mielőtt a mobiltelefonra Java alkalmazást telepítenénk meg kell vizsgálni, hogy az adott eszköz

- képes-e Java alkalmazást futtatni, azaz van-e rajta Java virtuális gép;
- a telepítendő alkalmazás által használt CLDC, MIDP és esetleg JSR konfigurációk rendelkezésre állnak-e.

A telepítéshez a telefont csatlakoztatni kell a számítógéphez. Erre manapság két lehetőségünk van.

1. A telefonhoz adott vagy beszerezhető adatkábel segítségével a számítógép USB portján keresztül teremtjük meg a kapcsolatot. Ekkor (valószínűleg) az adatkábel meghajtóját is telepítenünk kell a számítógépre.
2. Bluetooth kapcsolatot használhatunk, ha a telefon és a számítógép is rendelkezik ezzel a lehetőséggel. (Beszerezhetünk USB portos Bluetooth eszközt a megfelelő meghajtóval, ha a számítógép nem rendelkezik ezzel.)

A csatlakoztatás után a telefon gyártója által biztosított (vagy attól valamilyen módon megszerezhető) program[1] segítségével telepíthetjük az alkalmazást a telefonra.

A telepítéshez két állomány szükséges: egy `jad` és egy `jar` kiterjesztésű fájl. A `jar` állomány tartalmazza a lefordított és összecsomagolt Java programot (osztályok, erőforrások), a `jad` állomány az alkalmazáshoz tartozó információkat adja meg. Ezt a két fájlt kell a telefonra másolni[2].

1.4. Fejlesztőkörnyezet

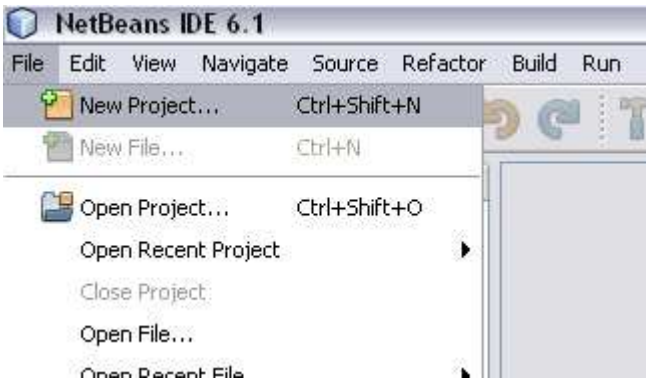
Az előző pontban megadott két fájlt kell a megfelelő Java forráskódból előállítani. Ezt támogatja a NetBeans környezet, ha annak a Mobility Pack-et tartalmazó változatát használjuk. Ekkor fordítás után a projekt `dist` alkönyvtárában létrejön a két szükséges fájl.

Rendelkezésre áll egy emulátor is (Sun Wireless Toolkit), amellyel a program futását a mobiltelefonon szimulálni tudjuk, így a fejlesztés során nem kell állandóan a telefonra letölteni a programot, elegendő csak a végleges változatot[3]. Mi ezeket fogjuk

[1] Nokia telefonok esetén erre használható a Nokia PC Suite program, amelyben megtalálható az Alkalmazások telepítése pont.

[2] A telepítő program lehet, hogy csak a `jar` állományt mutatja, és automatikusan kezeli a `jad` fájlt.

[3] Az egyes mobiltelefonok között, illetve az emulátor és egy mobiltelefon között bizonyos eltérések lehetnek a Java futtató környezet megvalósításában, ezért az egyes programok használata kis mértékben eltérő lehet különböző készülékeken.



1.1. ábra. Java ME projekt létrehozása

a továbbiakban használni. (A NetBeans-hez szükséges a Java környezet is, ezért ha az nincs telepítve a számítógépre, akkor azt kell először megtenni, illetve a NetBeans-t azzal együtt telepíteni.)

A szükséges eszközök szabadon letölthetőek a következő helyekről.

Java: `http://java.sun.com/javase/downloads/index.jsp`

NetBeans: `http://www.netbeans.org/downloads/index.html`

Sun Wireless Toolkit: `http://java.sun.com/products/sjwtoolkit/download.html`

Néhány gyártó biztosít saját fejlesztőkörnyezetet a Java programok fejlesztéséhez és telepítéséhez, ezeket azonban itt nem tudjuk áttekinteni.

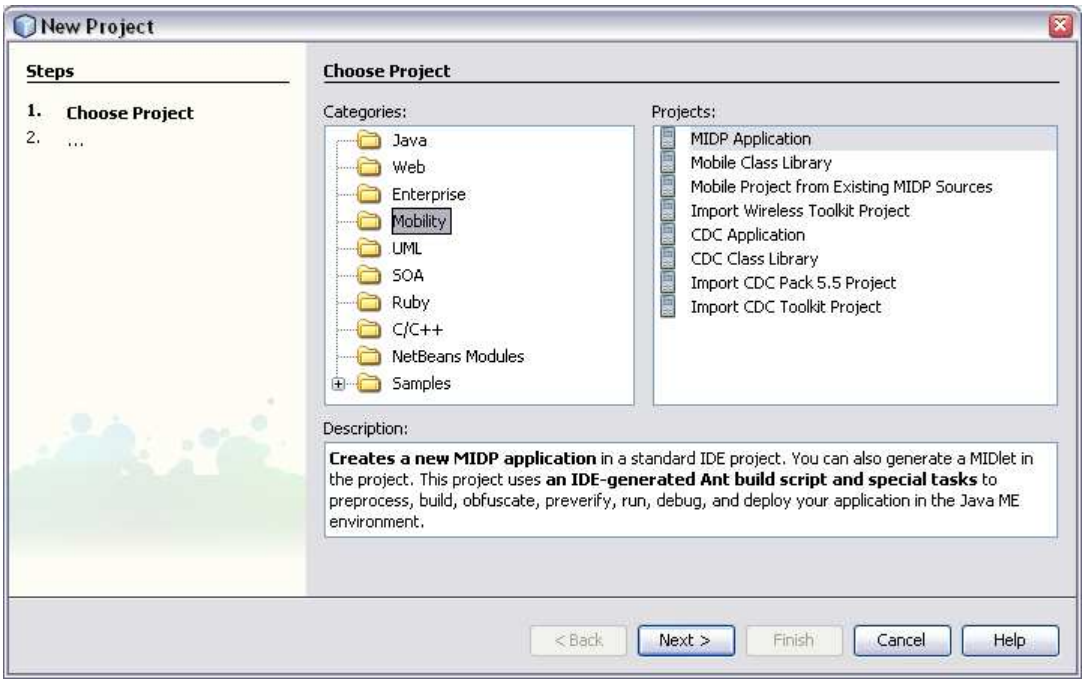
1.4.1. Java ME projekt létrehozása

A továbbiakban feltesszük, hogy a NetBeans-t és az emulátort telepítettük, valamint a NetBeans-t elindítottuk.

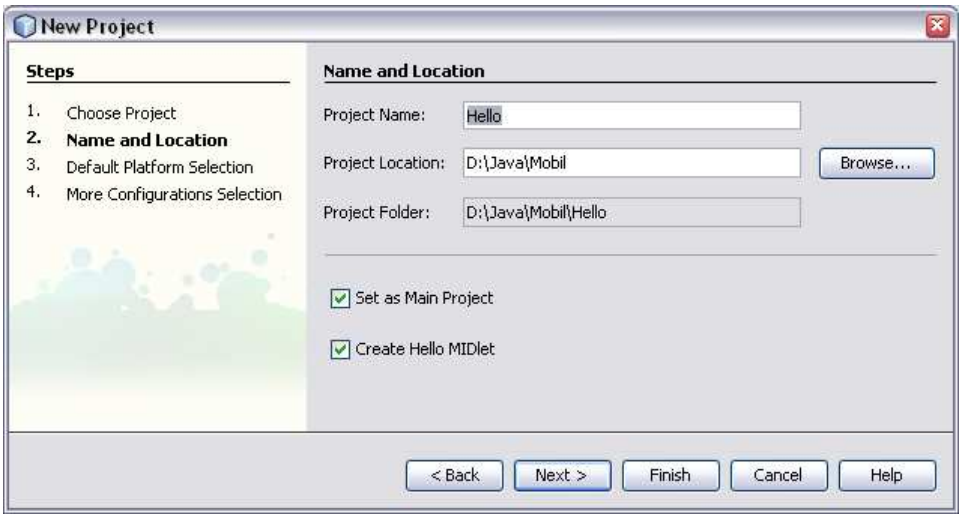
1. Hozzunk létre egy megfelelő projektet a `File` menüben a `New Project` menüpont segítségével (1.1. ábra).

2. A megjelenő párbeszédablakban a megfelelő kategóriát és projektet kell kiválasztanunk: `Mobility` kategória és `MIDP` projekt (1.2. ábra). Ha ez megvan a `Next` gomb lenyomásával mehetünk tovább.

3. Meg kell adnunk a projekt nevét, helyét, és állítsuk ezt be főprojektnek (ezt fordítja és futtatja, ha más projekt is nyitva van), valamint hagyjuk meg az alap projekt (`Create Hello MIDlet`) létrehozását (1.3. ábra), majd nyomjuk meg a `Next` gombot.

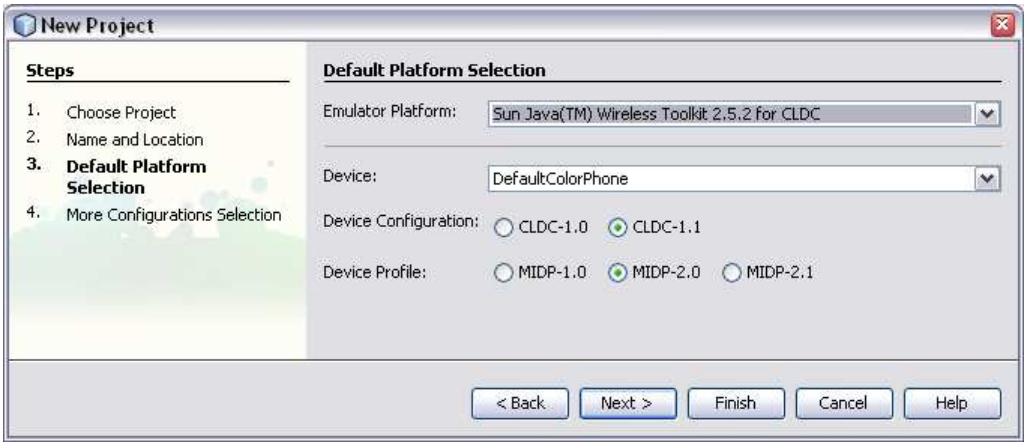


1.2. ábra. Projekt választása



1.3. ábra. Projekt adatai

4. A következő lépés a megfelelő platform kiválasztása. Emulátornak válasszuk a telepített emulátort. Ezután a mobiltelefon tulajdonságait kell ismernünk, és annak megfelelően eljárunk. Egységnek (device) az alapértelmezésként felkínált



1.4. ábra. Platform megadása

`DefaultColorPhone` megfelel a legtöbb mai telefon esetén. (Lehetőség van szürkeárnyalatos kijelzővel rendelkező, illetve teljes – QWERTY – billentyűzettel ellátott telefon kiválasztására. Ez az emulátoron történő tesztelés esetén lehet érdekes.) Ezt követően kell a telefonnak megfelelő konfigurációt (CLDC) és profájlt (MIDP) kiválasztanunk. Ezt követően befejezhetjük a projekt létrehozását.

A projekt létrejötte után azt a `Build` menü `Build Main Project` menüpontjával, illetve az `F11` gomb megnyomásával fordíthatjuk le. Az esetleges hibaüzenetek az alsó részen látható területen jelennek meg. Hibátlan fordítás után a `Run` menü `Run Main Project` menüpontjával, illetve az `F6` gomb megnyomásával futtathatjuk a programot. Futtatáskor elindul az emulátor (ez eltarthat egy kis ideig), ahol a `Launch` segítségével indíthatjuk el a programot.

Az előzőekben létrehozott projekt egy egyszerű "Hello Word" alkalmazás. A generált kód teljes, így az fordítható és futtatható. A futás eredménye látszik az 1.5. ábrán. Az emulátoron a gombokat egérkattintások segítségével "nyomhatjuk" meg[4].

1.4.2. A fejlesztőkörnyezet nézetei

A fejlesztőkörnyezet projekt nézetében láthatjuk a projekt szerkezetét, ami esetünkben egyetlen forrásfájlt tartalmaz (1.6. ábra). Itt lehet a szokásos módon navigálni a projekt elemei között. Egy adott elemre kattintva a megfelelő elem jelenik meg a

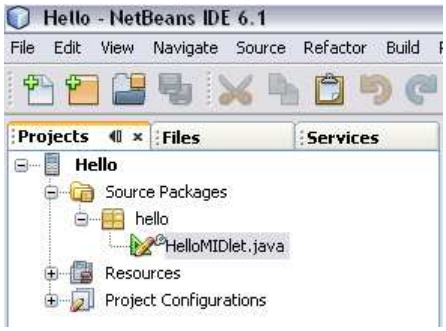
[4]Ez a használat felel meg a tényleges telefon használatnak, de lehetőségünk van "csalni", ugyanis a számítógép billentyűzete használható gépelésre, stb. A csalással az emulátor használata jóval kényelmesebb lehet, azonban soha ne feledkezzünk el arról, hogy a program ténylegesen egy telefonon fut majd, így a felület, adatok megadásának tervezése során azt vegyük figyelembe. A mobil telefonokra szánt alkalmazások felülete – ma még – lényegesen eltér a számítógépen futó programokétól.

1.5. ábra. A `Hello` program futása

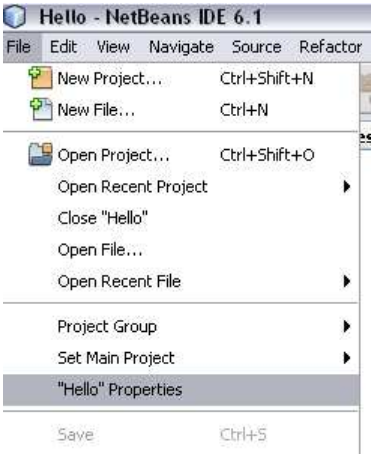
szerkesztő nézetben (pédául a Java forráskód). Ha a projekt nevére kattintunk a jobb egérgombbal, akkor a projekthez kapcsolódó tevékenységeket tartalmazó menü jelenik meg. Vagy a menü utolsó pontjában található `Properties`, vagy a `File` menü *"projektnév"* `Properties` menüpont (1.7. ábra) aktivizálásával tudjuk a projekt tulajdonságait megtekinteni, módosítani[5].

A központi részen elhelyezkedő szerkesztő nézetben dolgozhatunk az éppen aktuális elemmel. Ha az előzőleg létrehozott projektből a `HelloMIDlet.java` állományt választjuk ki, akkor a szerkesztő nézetben módosíthatjuk a kódot (`Source`) és a ké-

[5]Fejlesztés során például a debug információk generálását (`Build->Compiling`) célszerű megtartani, de a végleges változatban ezt jó kikapcsolni és az optimalizálást bekapcsolni. Az `Application Descriptor` részben lehet az alkalmazás tulajdonságait – terjesztő, ikon – megadni.



1.6. ábra. Projekt nézet



1.7. ábra. Projekt tulajdonságai

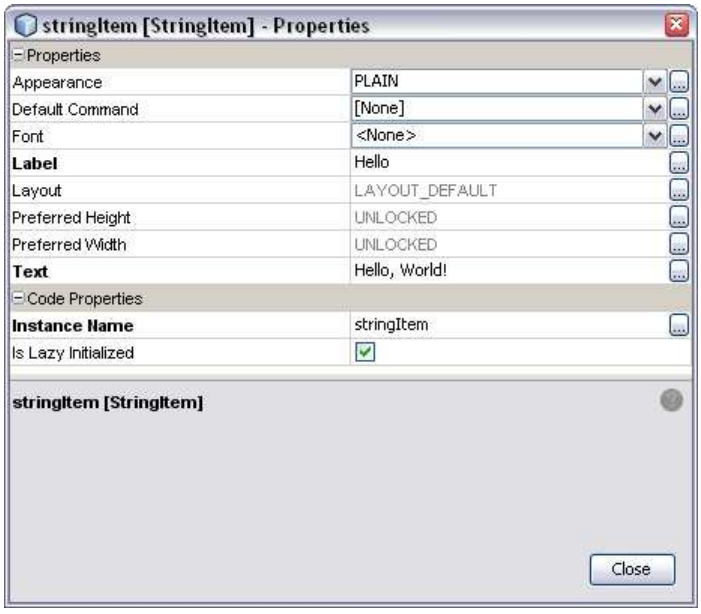
pernyőtervet (`Screen`). Ezen kívül[6] lehetőségünk van az egyes képernyők közötti kapcsolatot megtekinteni, tervezni, módosítani a `Flow` választásával. A negyedik opció (`Analyzer`) választásával a felesleges elemeket azonosíthatjuk, illetve a MIDP változatok közötti kompatibilitást vizsgálhatjuk.

Az 1.8 ábrán láthatjuk a programunkhoz tartozó képernyőtervet. Az egyes elemeket kiválasztva tekinthetjük meg és módosíthatjuk azok tulajdonságait. Ezt megtehetjük a jobb oldalon elhelyezkedő tulajdonság leíró rész segítségével, illetve jobb egérgombos kiválasztás esetén a legördülő menü `Porperties` pontjának aktivizálásával. Az utóbbi esetben megjelenő párbeszédablakot mutatja az 1.9. ábra. A szerkesztő nézettől jobbra, az elem tulajdonságok felett található a paletta. Itt találhatóak azok az elemek, amelyeket felhasználhatunk a képernyők kialakításához.

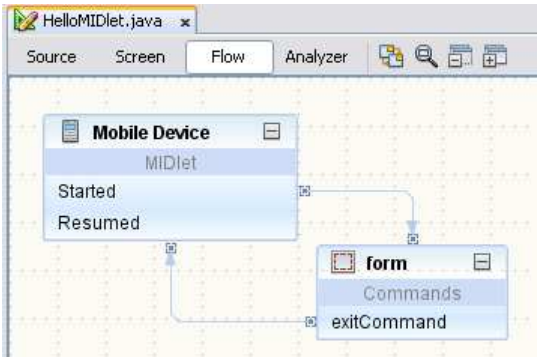
Vizsgáljuk meg a képernyők közötti kapcsolatokat leíró `Flow` nézetet! Itt láthatjuk, hogy milyen parancsokra, hogyan váltunk az egyes képernyők között. Esetünk-

[6]Ezek már a projekthez kapcsolódnak, és nem a kiválasztott elemhez.

1.8. ábra. A `Hello` program képernyőterve



1.9. ábra. A feliratot megadó (StringItem típusú) elem tulajdonságai



1.10. ábra. Kapcsolat a képernyők között

ben (1.10. ábra) ez rendkívül egyszerű, hiszen csak egyetlen képernyőnk van (`form`), amelyhez indítás után jutunk, és kilépéskor (`exitCommand`) visszatérünk az indító képernyőre. Itt lenne lehetőségünk újabb képernyők felvételére, azok tulajdonságainak beállítására, és megfelelő parancsok létrehozása után, a rákövetkezések megadására.

A fejlesztőkörnyezet további komponensei értelemszerűen használhatóak[7]. Felhívjuk a figyelmet, hogy a bemutatott projektben vizuális tervező eszközt használtunk, ezért állt rendelkezésünkre ennyi lehetőség. Ebben az esetben a Java kódban bizonyos részeket a környezet hoz létre, ezeket nem lehet változtatni. A kódban az ilyen részek szürkék. Megjegyzésekkel jelölik azokat a helyeket, ahol a kód (műveletek) kiegészíthető. Ez a kézi kódolást nagyon nehézkessé teszi, amire azért bizonyos funkciók elérésére szükség lehet.

Nem kötelező ezt az utat választani, lehet teljes egészében kódolásra hagyatkozni. Ekkor elveszítjük a vizuális konstrukció előnyeit, ugyanakkor a kód felett teljes egészében rendelkezhetünk, tetszőlegesen módosíthatjuk. A továbbiakban ezt mutatjuk be, ugyanis ennek ismeretében a vizuális megközelítés is használható, ugyanakkor az alapvető elemek ismerete nélkül az az út sem járható.

[7] Az olvasó feladata hogy ezeket kipróbálja, illetve ezek segítségével a bemutatott példát átalakítsa, kibővítse, kísérletezzen vele.

2. Java ME alapok

Ebben a fejezetben áttekintjük azokat az alapvető elemeket, amelyekkel Java ME környezetben készülő programok esetén találkozunk.

2.1. MIDlet

A MIDP környezetben futó alkalmazások a *MIDlet*-ek, mobiltelefonok esetén ilyen programokat kell készítenünk, azaz ilyen típusú oblektumokat kell létrehoznunk. A MIDleteknek három állapotuk lehet:

Aktív: az alkalmazás indításakor a MIDlet a `startApp()` művelet hívásával kerül ebbe az állapotba. A `startApp()` műveletben kell gondoskodni a kezdő képernyő megjelenítéséről, a szükséges kezdeti inicializálásokról.

Felfüggesztett: Ha szükséges, például hívás érkezik, az alkalmazást felfüggeszti az alkalmazás felügyelő (AMS – Application Management Software), és ekkor a MIDlet `pauseApp()` műveletét hívja meg. Újraindításkor a `startApp()` művelet hajtódik végre, ezért ha az újraindítás eltér az első indítástól, akkor arra figyelni kell.

Megszüntetett: Az alkalmazás befejezésekor a MIDlet `destroyApp()` műveletét hívja meg az AMS. Itt lehet a befejezéshez kapcsolódó tevékenységeket (állapotmentés, megszüntetések) elvégezni. Ez az állapot a MIDlet alapértelmezett állapota. (Az alkalmazás a telefonon van, de még nem töltötte be a JVM.)

Az előzőekben tárgyalt három (`startApp()`, `pauseApp()`, `destroyApp()`) művelettel minden MIDlet rendelkezik, és valamilyen megvalósítást kell azokhoz rendelnünk. (Ha mást nem az üres programot.)

Ahhoz, hogy egy MIDletet készítsünk két csomag szükséges a Java ME könyvtárból. A Java ME elemeket a `javax.microedition` csomag alcsomagjai tartalmazzák. Esetünkben a `midlet` és az `lcdui` csomagok kellenek. Ezek alapján egy üres MIDlet a következőképpen nézne ki.

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

public class Üres extends MIDlet
{
    public void startApp()
    {
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean unconditional)
    {
    }
}
```

Egy MIDlet a telefon kijelzőjét használja, és a számítógépektől eltérően nincs lehetőség több ablak használatára, azaz egyszerre csak képernyőt tud megjeleníteni. A kijelző lekérdezésére szolgál a `Display` osztály statikus `getDisplay` művelete, amelynek paramétere egy MIDlet. Ezzel érhetjük el a MIDlet képernyőjét, ami a művelet visszatérési értéke, és `Display` típusú.

Egy Display típusú objektum `setCurrent` művelete szolgál a képernyő tartalmának beállítására. Ennek paramétere egy megjeleníthető (`Displayable`) objektum, és ez lesz az új tartalom. A paraméter rendszerint egy `Form` vagy `Screen` típusú objektum.

Egy Form típusú `form` megjelenítését mutatja a következő kódrészlet:

```
Display.getDisplay(this).setCurrent(form);
```

Annak érdekében, hogy ezt a hosszadalmas leírást elkerüljük mi rendszerint bevezetünk egy `show` műveletet erre a célra.

```
public class Üres extends MIDlet
{
    ...
    public void show(Displayable next)
    {
        Display.getDisplay(this).setCurrent(next);
    }
    ...
}
```

2.2. Form

Az egyik legalapvetőbb megjeleníthető elem a `Form`, ami egy olyan képernyő, amelyre felület elemek (`Item`) helyezhetőek el, és azokat együtt jeleníti meg. Ilyen elem lehet például egy címkézett szöveg (`StringItem`), egy kép (`ImageItem`), egy szerkesztőmező (`TextField`), egy választási értékhalmaz (`ChoiceGroup`). Ezeket az elemeket a konstruktorban, vagy az `append` művelettel vehetjük a formhoz.

2.3. Parancsok

Minden képernyőhöz parancsokat rendelhetünk, amelyeket a telefon megfelelő gombjaival (softkey) aktivizálhatunk. Amennyiben több parancsot rendelünk egy képernyőhöz, mint ahány gomb használható, akkor egy menübe kerülnek a meg nem jeleníthető parancsok, és az egyik gomb hatására a menü jelenik meg, ahonnan kiválaszthatjuk a kívánt parancsot.

Parancsok megadására szolgál a `Command` osztály. Ennek egy példánya lehet egy parancs, amelynek konstruktorában lehet megadni a megjelenítendő szöveget, a parancs típusát (azaz, hogy melyik gombhoz próbáljuk rendelni), és egy prioritási értéket.

A létrehozott parancsot az `addCommand` művelettel rendelhetjük a képernyőhöz. A parancsok kezelését végző objektumot a `setCommandListener` művelettel állíthatjuk be.

2.4. Parancsok kezelése

A parancsok kezelését végző objektumnak meg kell valósítania a `CommandListener` interfészt. Ez egyetlen művelet, a `commandAction`, megvalósítását írja elő. A művelet első paramétere a kiváltott parancs, a második paramétere az a képernyő, amin kiváltották. (Ugyanazt a parancsot több képernyőhöz is hozzá lehet rendelni.)

Az egyszerű környezet és az erőforrások kímélése miatt a parancsok kezelése során figyelni kell arra, hogy a műveletek nem lehetnek időigényesek, mert a `commandAction` művelet közvetlenül kerül meghívásra a parancs kiadásakor, nincs eseménysor, illetve egyéb esemény. Időigényes műveletek blokkolnák a felhasználói felületet, ami nem engedhető meg, hiszen bármikor képesnek kell lenni azon váltani, például beérkező hívás esetén. Ha időigényes műveletet kell végrehajtani, akkor azt külön szálban kell megtenni, ahogy ezt majd a fájlkezelés során látni fogjuk.

2.5. Mobiltelefon tulajdonságainak lekérdezése

Az előző fejezetben láttuk, hogy egy projekt létrehozásakor meg kell adnunk a CLDC és MIDP verziót, ami illeszkedik a mobiltelefonhoz. A kérdés, hogy miként válasszunk a lehetőségek közül. A legalacsonyabb verzió biztosan futni fog minden telefonon, amin van Java, de lehet, hogy nagyon sok megszorítást tartalmaz, illetve bizonyos funkciók nem használhatóak. Ennek megfelelően a következő szempontokat kell figyelembe vennünk. A lehető legkisebb verzió választása, ami biztosítja a számunkra szükséges szolgáltatásokat, annak érdekében, hogy a legtöbb telefonnal kompatibilis legyen. Ugyanakkor a kiválasztott verzió nem haladhatja meg a célkészülék által támogatott szabványt.

Honnan tudhatjuk meg, hogy a telefon mit támogat? Az alábbi lehetőségek közül választhatunk, ha valamelyik rendelkezésre áll.

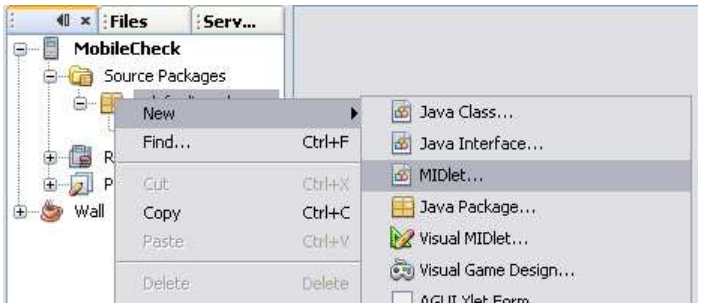
- A telefonon le lehet ezt az adatot kérdezni.
- A telefonhoz adott kezelési útmutató megadja a rendelkezésre álló CLDC és MIDP verziót.
- A gyártó honlapján szerepel a készülék leírásában.

Ha ezek közül egyik sem áll rendelkezésre, akkor készíthetünk egy alkalmazást, ami egyéb jellemzőkkel együtt ezt is megadja. A következőkben bemutatjuk miként készíthetünk el egy ilyen alkalmazást.

A NetBeans elindítása után hozzunk létre egy új Java ME projektet! A `File` menü `New Project` pontja után válasszuk a `Mobility` és `MIDP Application` lehetőségeket ugyanúgy, mint az előző fejezetben. Ezután adjuk meg a projekt nevét (pl: MobileCheck), a helyét, állítsuk be főprojektnek, de az előző fejezettel ellentétben, most ne hozzunk létre egy hello alkalmazást, azaz a `Create Hello MIDlet` lehetőséget ne válasszuk ki! Ezt követően az emulátor platform és az eszközt válasszuk ki ugyanúgy mint eddig, és most konfiguráció, illetve profájl esetén válasszuk a lehető legkisebb verziót! (CLDC 1.0 és MIDP 1.0.) Ezt kell megtennünk, hogy minden telefonon fusson majd az alkalmazás. Ezután fejezzük be a projekt létrehozását!

Az eredmény egy üres projekt, amit a projekt nézetben láthatunk. A projektet kibontva a `Source Packages` alatti `default package` "csomagban" hozzuk létre az egyetlen Java fájlt, amit a program tartalmaz. Ehhez a `default package` elemen nyomjuk meg az egér jobb gombját, a felbukkanó menüből válasszuk a `New` almenüt, ezen belül pedig a `MIDlet` pontot (2.1. ábra). A párbeszédablakban adjuk meg a MIDlet nevét (MobileCheck)[1].

[1]Ne a Visual MIDlet lehetőséget válasszuk, mert akkor a grafikus tervezővel alakíthatjuk ki a projekt elemeit, ezt pedig most nem akarjuk.



2.1. ábra. MIDlet hozzáadása a projekthez

Látható, hogy a létrejövő Java forrásfájl tartalmazza a szükséges importokat és a MIDlethez tartozó három műveletet. Ezt kell kiegészítenünk.

Első lépésben vezessünk be egy logikai értéket (`midletPaused`) annak jelzésére, hogy felfüggesztés után vagy elölről indul-e a program. Ennek kezdeti értéke hamis. A `pauseApp()` műveletben értékét igazra állítjuk, a `startApp()` műveletben pedig hamisra. Mielőtt hamisra állítanánk az értékét, megvizsgáljuk, és ha hamis, akkor meg kell jelenítenünk az alkalmazás egyetlen képernyőjét az előzőekben megismert `show` művelet segítségével. Az alkalmazás befejezésekor nincs teendőnk, ezért a `destroyApp` művelet az üres program lesz.

Az alkalmazás képernyője egy `Form` típusú objektum lesz, amelynek legyen a neve `info`. Erre kell elhelyeznünk a kívánt információkat. Mindegyik szöveges, amelyhez egy címke tartozik. Erre használhatjuk a `StringItem` típust. A következő adatokat fogjuk megadni a feltüntetett változókban:

`platform` a Java VM típusa (készítője),

`conf` a CLDC konfiguráció verziója,

`prof` a MIDP profájl verziója,

`fv` fájlkezelés verziója, lehetősége,

`enc` karakterkódolás,

`loc` nyelvi beállítás,

`memory` teljes tárterület,

`freemem` szabad tárterület,

`rmsmem` RMS-es keresztül elérhető tárterület,

`screen` képernyő mérete,

`touch` érintőképernyős-e a készülék.

Az értékek első csopotját a `System.getProperty` művelet hívásával tudjuk meghatározni. Ennek paramétere egy string, ami megadja a kérés típusát. Az eredmény egy string, illetve `null`, ha az adott jellemző nem érhető el, nem használható a telefonon.

A tárterület lekérdezéséhez a `Runtime` osztály egy példánya szükséges, amelynek `totalMemory` és `freeMemory` függvényei adják meg a szükséges értékeket. (Az RMS használatáról később lesz szó, ezt most nem részletezzük.) A képernyő adatainak lekérdezésére létrehozunk egy speciális képernyőt, amelynek `getWidth` és `getHeight` függvényei adják meg a képernyő kiterjedését, valamint a `hasPointerEvents` művelettel ellenőrizhetjük, hogy képes-e érintéseket kezelni a rendszer. (Az objektumnak meg kell valósítania az osztály absztrakt `paint` műveletét, még akkor is, ha semmit sem akarunk tenni.)

Ezeket a `StringItem` objektumokat az `append` művelettel adhatjuk hozzá egy (kezdetben üres) Formhoz, illetve a konstruktorban egy tömbfelhasználásával tehetjük meg ugyanezt. A programban mindkét lehetőséget használjuk.

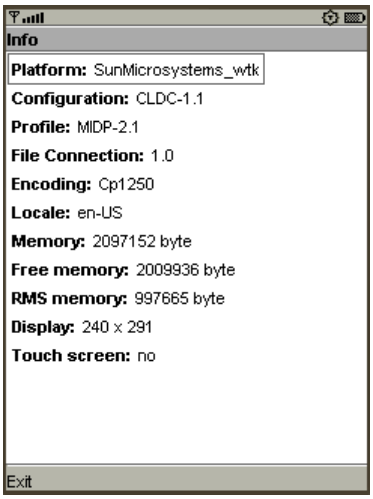
Szükségünk lesz még egy parancsra, amellyel kiléphetünk az alkalmazásból. Erre szolgál a `Command` típusú `exit` változó. A létrehozás után ezt kell a képernyőhöz adnunk az `addCommand` művelettel, és be kell állítanunk a képernyő parancskezelőjét a `setCommandListener` művelettel. A parancskezelő a MIDlet lesz, ehhez annak meg kell valósítania a `CommandListener` interfészt, azaz az `commandAction` műveletet. A műveletben meghívjuk a kilépést kezelő `exit` eljárást, amiben levesszük a képernyőt a megjelenítőről, és értesítjük a környezetet a kilépésről.

Az eddigieket összegezve a következő programhoz jutunk, amelynek eredménye a 2.2. ábrán látható.

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

public class MobileCheck extends MIDlet implements CommandListener
{
    private boolean     midletPaused;
    private Form        info;
    private StringItem  fv;
    private StringItem  conf;
    private StringItem  prof;
    private StringItem  platform;
    private StringItem  enc;
    private StringItem  loc;
    private StringItem  freemem;
    private StringItem  memory;
```

```
private StringItem  rmsmem;
private StringItem  screen;
private StringItem  touch;
private Command     exit;

public MobileCheck()
{
    exit = new Command("Exit", Command.EXIT, 0);
    fv = new StringItem("File Connection: ", "");
    conf = new StringItem("Configuration: ", "");
    prof = new StringItem("Profile: ", "");
    platform = new StringItem("Platform: ", "");
    enc = new StringItem("Encoding: ", "");
    loc = new StringItem("Locale: ", "");
    memory = new StringItem("Memory: ", "");
    freemem = new StringItem("Free memory: ", "");
    rmsmem = new StringItem("RMS memory: ", "0 byte");
    screen = new StringItem("Display: ", "");
    touch = new StringItem("Touch screen: ", "no");
    String  s = System.getProperty(
                      "microedition.io.file.FileConnection.version");
    if ( s == null )    fv.setText("Unsupported");
    else                fv.setText(s);
    s = System.getProperty("microedition.configuration");
    if ( s == null )    conf.setText("Unsupported");
    else                conf.setText(s);
    s = System.getProperty("microedition.profiles");
    if ( s == null )    prof.setText("Unsupported");
    else                prof.setText(s);
    s = System.getProperty("microedition.platform");
    if ( s == null )    platform.setText("Unsupported");
    else                platform.setText(s);
    s = System.getProperty("microedition.encoding");
    if ( s == null )    enc.setText("Unsupported");
    else                enc.setText(s);
    s = System.getProperty("microedition.locale");
    if ( s == null )    loc.setText("Unsupported");
    else                loc.setText(s);
    Runtime     rt = Runtime.getRuntime();
    memory.setText(rt.totalMemory() + " byte");
    freemem.setText(rt.freeMemory() + " byte");
    RecordStore     memrms;
```

```
        try
        {
            memrms = RecordStore.openRecordStore("memory", true);
            rmsmem.setText(memrms.getSizeAvailable() + " byte");
            memrms.closeRecordStore();
        } catch (RecordStoreException e) {}
        Canvas      test = new Canvas()
        {
            protected void paint(Graphics arg0) {}
        };
        screen.setText(test.getWidth() + " x " + test.getHeight());
        if ( test.hasPointerEvents() )  touch.setText("yes");
        info = new Form("Info", new Item[]
                                { platform, conf, prof, fv, enc, loc });
        info.append(memory);
        info.append(freemem);
        info.append(rmsmem);
        info.append(screen);    info.append(touch);
        info.addCommand(exit);
        info.setCommandListener(this);
    }

    public void startApp()
    {
        if ( !midletPaused )    show(info);
        midletPaused = false;
    }

    public void pauseApp()  { midletPaused = true; }

    public void destroyApp(boolean unconditional)   {}

    public void exit()
    {
        show(null);
        destroyApp(true);
        notifyDestroyed();
    }

    public void show(Displayable next)
    {
        Display.getDisplay(this).setCurrent(next);
    }
```



2.2. ábra. A tulajdonság lekérdező program futási eredménye emulátoron

```
    public void commandAction(Command cmd, Displayable disp)
    {
        if ( cmd == exit )  exit();
    }
}
```

3. Felhasználói felület

Mobiltelefonokra szánt alkalmazások készítésekor nem használhatóak a Java SE platformon használt elemek (AWT, Swing) a bevezetőben említett korlátozott erőforrások miatt. A Java ME-ben használatos felhasználói felület elemei az `lcdui` csomagba kerültek, és ez számos olyan elemet tartalmaz, amelyek megkönnyítik a mobiltelefonos programok fejlesztését.

A mobilokra szánt alkalmazásokat a felületük szerint kétféleképpen csoportosíthatjuk.

- Adatokkal foglalkozó alkalmazások, amelyekben adatokat kell szövegesen, választással, vagy egyéb módon megadni. Az elemeket rendszerint űrlapokba szervezzük, és az alkalmazásban menük segítségével navigálhatunk.

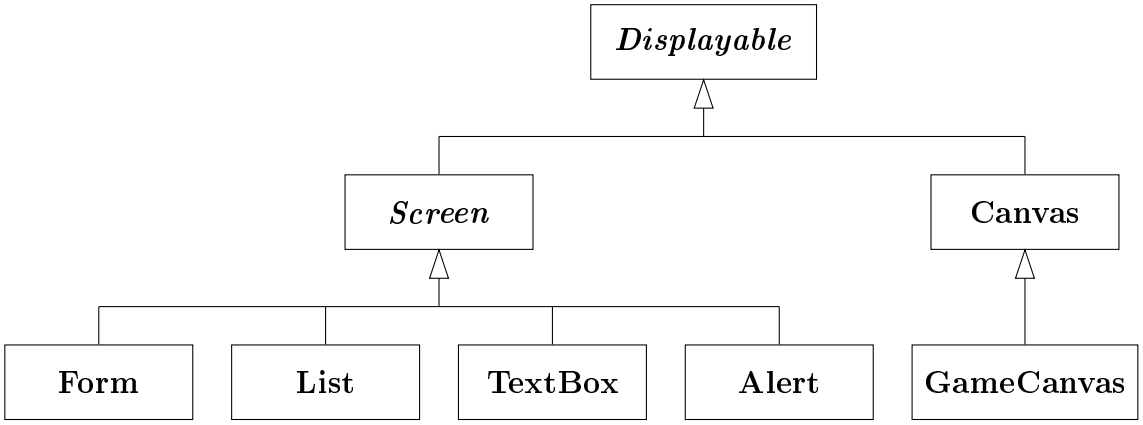
- Grafikus alkalmazások, ahol a kijelzőre rajzolni kell, azt pixelesen kell kezelni. Ilyen alkalmazások tipikus példái a játékok.

Ennek megfelelően az `lcdui` csomagban is két nagy csoportba sorolhatóak a kijelzőn megjeleníthető képernyők. Ezeket lehet a MIDlethez tartozó egyetlen kijelzőn megjeleníteni. Ahogy az előző fejezetben láttuk, ezt a `Display` osztály statikus `getDisplay` műveletével érhetjük el, amelynek paramétere a MIDlet objektum. Az ezen történő megjelenítés (`setCurrent`) egyszerűsítésére vezettük be a `show` műveletet. A `Display` osztály egyéb műveletekkel is rendelkezik, ezeket meg lehet ismerni a Java dokumentációjából, számunkra a további műveletek itt nem fontosak.

3.1. Képernyők

A kijelzőn megjeleníthető képernyők egy absztrakt osztályból származnak (3.1. ábra). Ez az osztály a `Displayable` osztály, és ez alkalmas arra, hogy a képernyőket egységesen tudjuk kezelni[1]. Az absztrakt osztály és az ebből származtatott osztályok mindegyike értelemszerűen a `javax.micoredition.lcdui` csomagban található.

[1]Például a megjelenítéshez használt `setCurrent` (a mi programjainkban `show`), és a parancskezelésben a `commandAction` művelet paramétere.



3.1. ábra. A képernyők osztályai

`void addCommand(Command cmd)`	: Felveszi a parancsot a képernyőhöz.
`void removeCommand(Command cmd)`	: Eltávolítja a parancsot.
`void setCommandListener(CommandListener l)`	: Beállítja a parancskezelőt.
`int getHeight()`	: Megadja a képernyő magasságát.
`int getWidth()`	: Megadja a képernyő szélességét.

3.1. táblázat. A `Displayable` osztály fontosabb műveletei

A képernyők mindegyike a teljes kijelzőt használja[2], a Java ME ebben is eltér a számítógépen megszokott ablakozós rendszertől. Ennek megfelelően az ilyen típusú elemek közül pontosan egy látható a MIDlet futása alatt a kijelzőn, és ezek között lehet a `Display.setCurrent` művelettel váltani.

A képernyők közös, `Displayable` osztályban, megvalósított fontosabb műveleteit adja meg a 3.1. táblázat.

A `Screen` absztrakt osztályból származtatott képernyők az adatközpontú alkalmazások elemei[3], míg a `Canvas` osztály a grafikus programokhoz nélkülözhetetlen. A MIDP 2.0-ban ebből hozták létre speciálisan játékok számára a `GameCanvas` osztályt. A következőkben ezeket a képernyőket tekintjük át röviden.

[2]A parancskezelő gombokhoz szükséges területtől eltekintve. (A MIDP 2.0-tól kezdődően a `Canvas` esetén lehetőség van a teljes képernyő használatára.)

[3]Ha a grafikus alkalmazás tartalmaz menüt, akkor értelemszerűen `List` típusú elemeket ott is használ(hat)nak.

`void append(String s, Image im)`	: A lista végére felveszi az adott elemet.
`void delete(int i)`	: Az adott indexű elemet törli a listából.
`void deleteAll()`	: Az összes elemet törli (MIDP 2.0-tól).
`int getSelectedIndex()`	: A kiválasztott elem indexe.
`String getString(int i)`	: Az adott elem szövege.
`void insert(int i, String s, Image im)`	: Beszúr egy elemet az adott helyre.
`boolean isSelected(int i)`	: Kiválasztott-e az adott indexű elem?
`void set(int i, String s, Image im)`	: Az adott indexű elemet kicseréli.
`void setSelectedIndex(int i, boolean s)`	: Az elem kiválasztottságát állítja.
`int size()`	: A lista elemszáma.

3.2. táblázat. A `List` osztály fontosabb műveletei

3.2. List

Ez az elem a teljes kijelzőn jeleníti meg értékek halmazát listaszerűen, és ezen értékekből lehet választani. A választás miatt megvalósítja a választások egységes kezelésére szolgáló `Choice` interfészt. A választásoknak, így a listáknak is három fajtája létezik, amelyet a lista létrehozásakor a konstruktor második paraméterében adhatunk meg.

Implicit A kiválasztásakor a listához rendelt `Command.SELECT` típusú, illetve az előre definiált alapértelmezett `List.SELECT_COMMAND` parancs aktivizálódik. A paraméter ekkor `List.IMPLICIT`.

Kizáró Egyetlen érték jelölhető ki. A kijelölés nem von maga után akciót. A paraméter értéke `List.EXCLUSIVE`.

Többszörös Tetszőleges számú érték kijelölhető. A paraméter `List.MULTIPLE`.

Listákban csak speciális értékeket jeleníthetünk meg. Az érték egy szöveg (`String`), amelyhez tartozhat egy ikon (kép) is. Ha nem akarunk ikont felvenni a szöveghez, akkor a `null` értéket adjuk meg a szöveg mellett.

`List` típusú objektumot kétféleképpen hozhatunk létre. Megadjuk a lista (képernyő) nevét és típusát, illetve ezeken kívül megadjuk a lista elemeit szövegek és képek tömbjével. A 3.2. táblázat tartalmazza a listák legfontosabb műveleteit a teljesség igénye nélkül.

A következő egyszerű alkalmazás szemlélteti a listák használatát. A projektet az előző fejezetben leírtak szerint hozzuk létre. A program kezdeti képernyőn egy menüt jelenít meg listaként, amiből választhatunk kizáró vagy többszörös lista használata között, amelyekből a vissza gomb megnyomásával a menübe jutunk (3.2. ábra). A menün kívüli kiválasztásokat a rövidség érdekében nem kezeljük. Az olvasó feladata ugyanennek az alkalmazásnak az elkészítése a vizuális tervező segítségével.

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

public class Lista extends MIDlet implements CommandListener
{
    private boolean paused = false;
    private List    menü;
    private List    egy;
    private List    több;
    private Command kilép;
    private Command vissza;

    public Lista()
    {
        menü = new List("Menü", List.IMPLICIT);
        egy = new List("Egyszeres", List.EXCLUSIVE);
        több = new List("Többszörös", List.MULTIPLE);
        menü.append("Egyszeres", null);
        menü.append("Többszörös", null);
        egy.append("Első", null);
        egy.append("Második", null);
        egy.append("Harmadik", null);
        több.append("Első", null);
        több.append("Második", null);
        több.append("Harmadik", null);
        kilép = new Command("Vége", Command.EXIT, 0);
        vissza = new Command("Vissza", Command.BACK, 0);
        menü.addCommand(kilép);
        egy.addCommand(vissza);
        több.addCommand(vissza);
        menü.setCommandListener(this);
        egy.setCommandListener(this);
        több.setCommandListener(this);
    }

    public void startApp()
    {
        if ( !paused )    show(menü);
        paused = false;
    }

    public void pauseApp()  { paused = true; }
```

```
    public void destroyApp(boolean unconditional)   {}

    public void exit()
    {
        show(null);
        destroyApp(true);
        notifyDestroyed();
    }

    public void show(Displayable next)
    {
        Display.getDisplay(this).setCurrent(next);
    }

    public void commandAction(Command cmd, Displayable disp)
    {
        if ( cmd == kilép )          exit();
        else if ( cmd == vissza )    show(menü);
        else if ( cmd == List.SELECT_COMMAND )
        {
            if ( menü.getSelectedIndex() == 0 ) show(egy);
            else                                show(több);
        }
    }
}
```

3.2. ábra. A listákat bemutató program futási képei

A programban nem vettünk fel külön gombot és parancsot a választás kezelésére, az alapértelmezett kiosztást használtuk. (Ez a középső "navigáló gomb", illetve egy Választ vagy Select feliratú gomb lesz a telefonon.)

3.3. TextBox

Ez a képernyő lehetővé teszi, hogy a felhasználó egy szöveget adjon meg a telefonokon szokásos (pl.: SMS) módon.A létrehozáskor meg kell adnunk a szerkeszthető szöveg maximális méretét, ezt nem léphetjük túl. Figyelni kell arra is, hogy a telefon által biztosított maximális hosszt ne lépjük túl.

A konstruktor paraméterei a képernyő címe, a kezdeti szöveg, a maximális szöveghossz és a tartalom jellegének megszorítása. Ezeket a megszorításokat egy konstanssal adhatjuk meg a következők szerint. (Ugyanezek érvényesek a nem teljes képernyős szerkesztő (`TextField`) esetén is.)

`TextField.ANY` Tetszőleges karakter bevihető.

`TextField.EMAILADDR` Egy e-mail címben szereplő jelek adhatóak meg.

`TextField.NUMERIC` Egész számot fogad el a szerkesztő.

`TextField.DECIMAL` Előjeles decimális szám vihető be.

`TextField.PHONENUMBER` Telefonszám írható be (számjegyek és `+`, `*`, `#` jelek).

`TextField.URL` Egy internetcím adható meg.

Ezeken az értékeken kívül még megadhatunk módosító értékeket is, például jelszó jellegű bevitelt akarunk, szerkeszthető-e a tartalom, stb.

Az osztály fontosabb műveleteit tartalmazza a 3.3. táblázat. Ezek segítségével az ilyen típusú objektumok használata értelemszerű.

`int getMaxSize()`	: A maximálisan tárolható szöveg hossza.
`String getString()`	: A szerkesztő tartalma.
`void delete(int p, int h)`	: Az adott pozíciótól töröl adott számú jelet.
`int getCaretPosition()`	: Az aktuális szerkesztési pozíció.
`void insert(String s, int p)`	: Az adott helyre beszúrja a szöveget.
`void setString(String s)`	: Beállítja a szerkesztő tartalmát.
`int size()`	: Az aktuális szöveg hossza.

3.3. táblázat. A `TextBox` osztály fontosabb műveletei

3.4. Alert

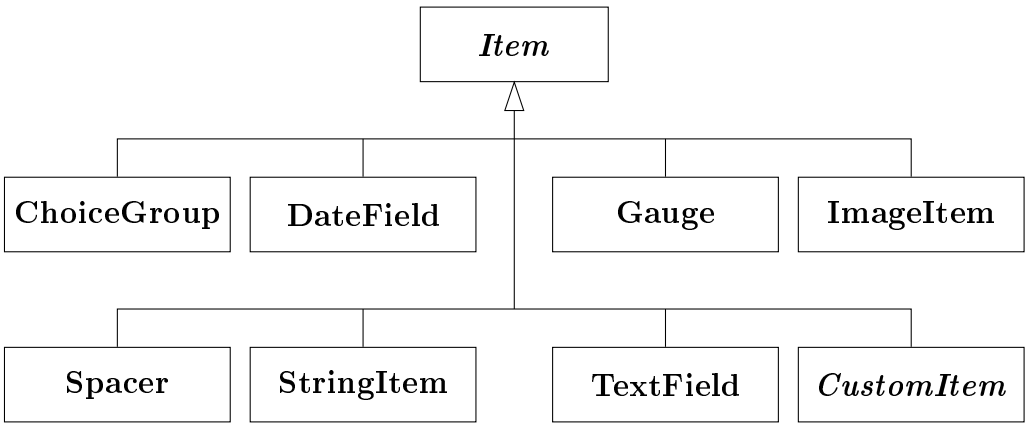
Ezek a speciális képernyők szolgálnak arra, hogy a felhasználónak bizonyos ideig valamilyen adatot, információt, figyelmeztetést jelenítsünk meg, mielőtt egy másik képernyőre váltanánk. Az idő beállítható (`setTimeOut`), illetve akár "örökre" is a kijelzőn maradhat egy ilyen képernyő. (Ekkor értelemszerűen egy parancs segítségével lehet továbblépni. Nem árt ellenőrizni, hogy a telefon ezt a lehetőséget támogatja-e, régebbi készülékeknél kivételt dobhat a program.)

Különböző típusai lehetnek: riasztás (`ALARM`), megerősítés (`CONFIRMATION`), hiba (`ERROR`), információ (`INFO`) és figyelmeztetés (`WARNING`). Ez a képernyőn látható képet befolyásolja. A konstansokat az `AlertType` osztály tartalmazza. Az üzenet címét, illetve szövegét a konstruktorban adhatjuk meg.

3.5. Form

Az előző fejezetekben már láttunk példákat formok használatára. Az ilyen típusú objektumok adnak lehetőséget űrlapszerűen elhelyezett adatbeviteli elemek együttes megjelenítésére, kezelésére. Egy form tetszőlegesen válogatott elemeket tartalmaz. A megvalósítás kezeli az elemek elhelyezkedését, bejárásukat, illetve a képernyő görgetését is szükség esetén.

Az `Item` absztrakt osztály adja meg a formra elhelyezhető elemek (vezérlők) alaptípusát és közös viselkedését, ebből kell származtatni a konkrét típusokat (3.3. ábra). A leggyakrabban használt beviteli elemeket megvalósították, és ezek a rendelkezésünkre állnak, de ha szükségünk van speciális elemre, akkor a `CustomItem` osztályból létre-



3.3. ábra. Egy formra helyezhető elemek osztályai

`void append(Item it)`	: Elem hozzáfűzése a formhoz.
`void delete(int index)`	: Az adott elemet eltávolítja.
`Item get(int index)`	: Adott elem lekérdezése.
`void insert(int index, Item it)`	: Az adott helyre beszúrja az elemet.
`void set(int index, Item it)`	: Kicseréli az elemet.
`int size()`	: Az elemek száma.

3.4. táblázat. A `Form` osztály fontosabb műveletei

`void append(String s, Image i)`	: Elem hozzáfűzése.
`void delete(int index)`	: Az adott elemet eltávolítja.
`int getSelectedIndex()`	: A kijelölt elem indexe.
`boolean isSelected(int index)`	: Kiválasztott-e az adott elem?
`void getString(int index)`	: Az elem felirata.
`int size()`	: Az elemek száma.

3.5. táblázat. A `ChoiceGroup` osztály fontosabb műveletei

hozhatjuk az igényeinknek megfelelő elemet[4]. Az elemek közül a `StringItem` osztállyal az előző fejezetekben foglalkoztunk, a többit a következőkben röviden áttekintjük.

Egy formra az elemeket vagy a létrehozáskor helyezzük fel, ekkor a konstruktor paramétere a cím mellett az elemeket tartalmazó tömb (`Form(String c, Item[] its)`), vagy az `append` művelettel vehetjük azokat a formhoz. A második esetben elegendő csak a form címét megadni a konstruktorban (`Form(String c)`). A 3.4. táblázat tartalmazza a `Form` osztály fontosabb műveleteit.

3.5.1. ChoiceGroup

Ezzel a vezérlővel választható elemek egy halmazát jeleníthetjük meg egy formon belül. A választás módja lehet kizáró (`EXCLUSIVE`), többszörös (`MULTIPLE`), illetve MIDP 2.0-tól kezdődően a kizáró választást `POPUP` is jellemezheti, ekkor az összes elem csak a vezérlő aktivizálásakor jelenik meg.

A konstruktorban meg kell adnunk a vezérlő nevét és a választás módját, amit kiegészíthet az elemek és ikonok tömbje. Az elemek a listákhoz hasonlóan ekkor is csak szövegek (stringek) lehetnek. A 3.5. táblázat tartalmazza az osztály műveleteit.

[4]Erre példa a NetBeans fejlesztői által létrehozott `TableItem` osztály, amelynek segítségével adatokat jeleníthetünk meg egy táblázatban. (Használatához importálni kell az `org.netbeans.microedition.lcdui` csomagot.)

3.5.2. DateField

Ez egy szerkeszthető komponens, amelyben dátumot és időt jeleníthetünk meg. A komponens a dátum és idő külön-külön és együtt is kezelhető az osztály `DATE`, `TIME`, illetve `DATE_TIME` konstansait felhasználva a konstruktor második paramétereként. A konstruktor első paramétere a komponens címkéje.

Az időpontot állítani, illetve lekérdezni a `setDate` és `getDate` műveletekkel lehet, amelyek a `Date` típust használják.

3.5.3. Gauge

Egy zárt intervallumba eső egész szám megadására, megjelenítésére alkalmas eszköz, amely egy folyamatjelzőhöz hasonlóan mutatja az értéket. Az intervallum alsó határa 0, felső határa megadható[5]. A komponens lehet interaktív, ekkor a felhasználó is állíthatja az értékét, illetve nem interaktív, ekkor csak érték megjelentésre szolgál, a felhasználó nem módosíthatja.

A konstruktor első paramétere a komponens címkéje, második az interaktivitást szabályzó logikai érték (igaz esetén interaktív), harmadik a maximális érték, és utolsó az aktuális érték. A `getValue` és `setValue` műveletekkel kérdezhető le, illetve állítható az aktuális érték.

3.5.4. ImageItem

Képek (`Image`) formon belüli megjelenítésére használható komponens. A kép elhelyezési módja (layout) szabályozható az osztály konstansai segítségével[6]. A komponensben a képhez tartozhat felirat, illetve olyan alternatív felirat, ami akkor látható, ha a kép nem jeleníthető meg.

A konstruktorban megadható a címke, a kép, az elhelyezési mód, az alternatív felirat. MIDP 2.0-tól kezdve ötödik paraméterként a megjelenési mód (sima, hivatkozás, gomb) is szabályozható.

3.5.5. TextField

Szövegbeviteli eszköz, amely hasonlít a `TextBox`-hoz, csak nem teljes képernyős, hanem formon belül elhelyezhető komponens. Ennek megfelelően programbeli használata (műveletek) megegyezik a `TextBox` használatával (műveletek, 3.3. táblázat).

[5]Nem feltétlen különül el minden érték a kijelzéskor, ha a felső határ egy bizonyos értéket meghalad. Az egyes részterületek részintervallumoknak felelnek meg.

[6]MIDP 2.0-tól kezdődően az összes `Item` típusú elem elhelyezkedése így szabályozható. `ImageItem` esetén MIDP 1.0-ban lehetővé tették a layout megadását.

3.5.6. Spacer

Ez egy olyan felületi elem, amely nem tartalmaz semmit, csak a formon belüli elemek távolságát szabályozhatjuk vele, azonban csak MIDP 2.0-tól használható. A konstruktor paraméterei az elem szélessége és magassága.

3.5.7. Példa alkalmazás

Készítsünk egy alkalmazást az eddigiek szemléltetésére. A programban árucikkekről tartunk nyilván adatokat. Az árukat a nevük alapján egy listából választhatjuk ki, illetve a listát bővíthetjük új áruval[7].

Egy árucikkről a következőket tartjuk nyilván:

- név,
- ár,
- a pénznem, amiben az ár értendő (forint, euró, dollár, jen),
- mennyiség,
- kiegészítő leírás,
- kezelési (szállítási) prioritás.

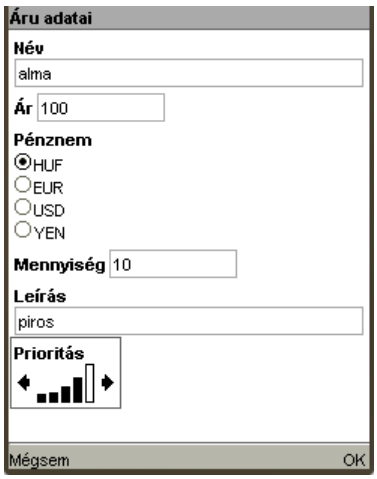
Az előzőekben megismert módon hozzunk létre egy üres `Mobility MIDP` alkalmazást. Ehhez vegyünk fel egy MIDletet (`Cikkek`), valamint két osztályt (`Adatlap` és `Cikk`) az áruk adatainak megjelenítésére, illetve tárolására. A tárolásért felelős osztály egyszerű, a szükséges adattagokat tartalmazza, azok lekérdező és beállító műveleteivel együtt. Ezt az osztályt adja meg a következő kód.

```
public class Cikk
{
    private String      név;
    private int         ár;
    private String      pénznem;
    private int         mennyiség;
    private String      leírás;
    private int         prioritás;
```

[7] A nyilvántartás adatait nem tudjuk még elmenteni jelenlegi ismereteink alapján, ezért a program mindig üres listával indul. Az adatkezelési ismeretek tárgyalása során ki fogjuk egészíteni az alkalmazást (4.1.4.).

```
    public Cikk()
    {
        név = leírás = "";
        pénznem = "HUF";
        ár = mennyiség = prioritás = 0;
    }

    public String neve()        { return név; }
    public int ára()            { return ár; }
    public String pénzneme()    { return pénznem; }
    public int mennyisége()     { return mennyiség; }
    public String leírása()     { return leírás; }
    public int prioritása()     { return prioritás; }

    public void újNév(String név)   { this.név = név; }
    public void újÁr(int ár)        { this.ár = ár; }
    public void újPénznem(String p) { pénznem = p; }
    public void újMennyiség(int me) { mennyiség = me; }
    public void újLeírás(String s)  { leírás = s; }
    public void újPrioritás(int pr) { prioritás = pr; }
}
```

Egy áru adatait egy formon jelenítjük meg. Ebben `TextField` komponensekben helyezzük el a név, ár, mennyiség, leírás és adatokat. Ár és mennyiség esetén numerikus értéket kell megadnunk, a másik két esetben bármilyen jel beírható. A pénznemet egy `ChoiceGroup` segítségével kezelhetjük, a prioritást pedig egy `Gauge` komponenssel (3.4. ábra).



3.4. ábra. Az árucikkek adatlapja

A komponenseket a konstruktorban hozzuk létre, és rendeljük a formhoz. Az osztály három szolgáltatást nyújt publikus műveletek formájában:

`ürít()` az adatlapot üres értékekkel tölti fel;

`adat(Cikk áru` az adatlapot a megadott áru jellemzőivel tölti ki;

`feltölt(Cikk áru)` a megadott áru mezőit az adatlap komponenseinek értékeire állítja.

A `hely` segédfüggvény megadja, hogy a paraméterként kapott felirat hányadik eleme a pénznemeket tartalmazó `ChoiceGroup` komponensben.

Az eddigiek alapján adódik a következő osztály.

```
import javax.microedition.lcdui.*;

public class Adatlap extends Form
{
    private TextField   név;
    private TextField   ár;
    private ChoiceGroup pénznem;
    private TextField   mennyiség;
    private TextField   leírás;
    private Gauge       prioritás;

    public Adatlap()
    {
        super("Áru adatai");
        név = new TextField("Név", "", 32, TextField.ANY);
        ár = new TextField("Ár", "", 8, TextField.NUMERIC);
        pénznem = new ChoiceGroup("Pénznem", Choice.EXCLUSIVE);
        mennyiség = new TextField("Mennyiség", "", 8, TextField.NUMERIC);
        leírás = new TextField("Leírás", "", 128, TextField.ANY);
        prioritás = new Gauge("Prioritás", true, 5, 0);
        pénznem.append("HUF", null);
        pénznem.append("EUR", null);
        pénznem.append("USD", null);
        pénznem.append("YEN", null);
        append(név);    append(ár);
        append(pénznem);
        append(mennyiség);
        append(leírás);
        append(prioritás);
    }
```

```
    public void ürít()
    {
        név.setString("");
        ár.setString("");
        pénznem.setSelectedIndex(0, true);
        mennyiség.setString("");
        leírás.setString("");
        prioritás.setValue(0);
    }

    public void adat(Cikk áru)
    {
        név.setString(áru.neve());
        ár.setString(Integer.toString(áru.ára()));
        pénznem.setSelectedIndex(hely(áru.pénzneme()), true);
        mennyiség.setString(Integer.toString(áru.mennyisége()));
        leírás.setString(áru.leírása());
        prioritás.setValue(áru.prioritása());
    }

    public void feltölt(Cikk áru)
    {
        áru.újNév(név.getString());
        áru.újÁr(Integer.parseInt(ár.getString()));
        áru.újPénznem(pénznem.getString(pénznem.getSelectedIndex()));
        áru.újMennyiség(Integer.parseInt(mennyiség.getString()));
        áru.újLeírás(leírás.getString());
        áru.újPrioritás(prioritás.getValue());
    }

    private int hely(String s)
    {
        for ( int i = 0; i < pénznem.size(); i++ )
            if ( s.equals(pénznem.getString(i)) )   return i;
        return 0;
    }
}
```

A MIDletben fel kell vennünk az áruk megadására használható űrlapot, ehhez hozzárendelni a parancsokat, az áruk választására szolgáló listát a parancsaival, és az áruk tárolására egy tömböt (`Vector`). Az `index` változó adja meg a kiválasztott áru indexét, ami az extremális -1 érték, ha új árut veszünk fel. A Java megvalósítás ezek után értelemszerű.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.Vector;

public class Cikkek extends MIDlet implements CommandListener
{
    private boolean     paused = false;
    private List        lista;
    private Adatlap     űrlap;
    private Command     új;
    private Command     választ;
    private Command     ok;
    private Command     mégsem;
    private Command     kilép;
    private Vector      áruk;
    private int         index;

    public Cikkek()
    {
        lista = new List("Áruk", List.IMPLICIT);
        új = new Command("Új", Command.ITEM, 0);
        választ = new Command("Választ", Command.ITEM, 0);
        kilép = new Command("Vége", Command.EXIT, 0);
        lista.addCommand(új);
        lista.addCommand(választ);
        lista.addCommand(kilép);
        űrlap = new Adatlap();
        ok = new Command("OK", Command.OK, 0);
        mégsem = new Command("Mégsem", Command.CANCEL, 0);
        űrlap.addCommand(ok);
        űrlap.addCommand(mégsem);
        áruk = new Vector();
        lista.setCommandListener(this);
        űrlap.setCommandListener(this);
    }

    public void startApp()
    {
        if ( !paused )  show(lista);
        paused = false;
    }

    public void pauseApp()  { paused = true; }
```

```
public void destroyApp(boolean unconditional)  {}

private void show(Displayable d)
{
    Display.getDisplay(this).setCurrent(d);
}

private void exit()
{
    show(null);
    destroyApp(true);
    notifyDestroyed();
}

public void commandAction(Command cmd, Displayable disp)
{
    if ( cmd == List.SELECT_COMMAND || cmd == választ )   módosít();
    else if ( cmd == új )                 új();
    else if ( cmd == ok )                 felvesz();
    else if ( cmd == mégsem )             show(lista);
    else if ( cmd == kilép )              exit();
}

private void módosít()
{
    index = lista.getSelectedIndex();
    űrlap.adat((Cikk)áruk.elementAt(index));
    show(űrlap);
}

private void új()
{
    index = -1;
    űrlap.ürít();
    show(űrlap);
}

private void felvesz()
{
    try
    {
        if ( index == -1 )
        {
```

```
                Cikk áru = new Cikk();
                űrlap.feltölt(áru);
                áruk.addElement(áru);
                lista.append(áru.neve(), null);
            }
            else
            {
                Cikk áru = (Cikk)áruk.elementAt(index);
                űrlap.feltölt(áru);
                áruk.setElementAt(áru, index);
                lista.set(index, áru.neve(), null);
            }
            show(lista);
        } catch (Exception e)   {}
    }
}
```

3.6. Canvas

Ez az osztály adja az alapot a kijelzőre vonatkozó alacsony szintű grafikus műveletekhez. Rendszerint ezt használjuk, ha az alkalmazásban grafikus képernyőre van szükségünk. Az ilyen típusú képernyők használata egy MIDleten belül keverhető a magasabb szintű képernyőkkel, azaz például meg lehet jeleníteni egy listát, majd annak egy pontjának aktivizálása után egy `Canvas` típusú kijelzőt, és onnan vissza lehet térni a listába.

Egy `Canvas` típusú objektum lehetőséget ad a mobiltelefon billentyűzetén végrehajtott események figyelésére. Előre definiált konstansokat tartalmaz a gombok azonosítására, illetve azok konvertálására (kurzor irányok és tűz gomb). A `keyPressed`, `keyReleased` műveleteket kell megvalósítanunk, ha a mobiltelefon egy gombjának lenyomását, felengedését akarjuk figyelni. A műveletek paramétere a gomb kódja, amelyet az osztály konstansaival azonosíthatunk. A `getGameAction` művelet szolgál az úgynevezett játék gombok (kurzor, tűz) konvertálására.

Az osztály rendelkezik egy absztrakt `paint` függvénnyel, amit a származtatott konkrét osztályokban meg kell valósítani, és ez felel a megjelenítésért. Ez a művelet kerül meghívásra, amikor a képernyőt, vagy annak egy részét meg kell jeleníteni. Ezt a `repaint` művelet hívásával érhetjük el[8], az alkalmazás a `paint` műveletet direkt módon nem hívhatja meg.

A `paint` művelet paramétere egy `Graphics` objektum, amire rajzolhatunk. A műveletben csak ezt az objektumot használhatjuk a megjelenítésre. Továbbá egy `Canvas`

[8]Ha nem akarjuk a `repaint` hívásával a teljes képernyőt újrarajzolni, akkor ennek paramétereként megadható a szükséges téglalap alakú terület.

3.5. ábra. A labdamozgató program futási képe

típusú elem rajzolását csak a `paint` művelettel valósíthatjuk meg[9]. Nem lehet a művelet paraméterében kapott objektumot tárolni, és arra a későbbiekben hivatkozni, az objektum hatóköre a művelet, ugyanis a művelet végrehajtása után a grafikus objektumon műveletek nem értelmezettek.

Egy `Graphics` típusú objektum hasonló lehetőségekkel rendelkezik, mint a Java SE-ben. Azaz képesek vagyunk egyszerű alakzatokat rajzolni, kitölteni, képet, szöveget megjeleníteni. Beállítható az aktuális szín, vonalstílus és font. A választható értékek halmaza értelemszerűen jóval elmarad a Java SE-ben biztosított lehetőségektől. (Például vonalstílus csak folyamatos vagy pontozott lehet, fontok esetén három méret és lényegében két családból lehet választani a szokásos – normál, félkövér, döntött, aláhúzott – stílusok kombinálásával.)

A következőkben bemutatunk egy egyszerű programot a `Canvas` használatának szemléltetésére. Ebben egy labdát mozgathatunk a képernyőn a telefon gombjai segítségével (3.5. ábra). Az első megközelítésben a labda elindul a megfelelő irányba, ha egy gombot lenyomunk, és megállíthatjuk a tűz vagy `5` gombbal.

A labda kezelésére vezessük be a `LabdaObj` osztályt, amelyből egy objektumot a `LabdaCanvas` típusú elemen jelenítünk meg. A MIDlet neve legyen `Labda` és hozzunk létre egy üres projektet ezzel a MIDlettel.

A MIDlet egyetlen feladata a képernyő megjelenítése, és ezzel egyidejűleg azon a labda mozgásának elindítása. Egyetlen parancs szükséges, amellyel kilépünk az alkalmazásból. Ekkor a szokásos tevékenységeken kívül a mozgatást is le kell állítani.

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

public class Labda extends MIDlet implements CommandListener
{
    private boolean     paused = false;
    private Command     vége;
    private LabdaCanvas nézet;
```

[9]Figyelembe kell venni, hogy egy bejövő telefonhívás megszakíthatja az alkalmazás futását, ezért a műveletben semmit sem tehetünk fel a hívási környezetről, a teljes megjelenítést kell implementálni.

```
    public Labda()
    {
        vége = new Command("Vége", Command.EXIT, 0);
        nézet = new LabdaCanvas();
        nézet.addCommand(vége);
        nézet.setCommandListener(this);
    }

    public void startApp()
    {
        if ( !paused )
        {
            show(nézet);
            nézet.start();
        }
        paused = false;
    }

    public void pauseApp() { paused = true; }

    public void destroyApp(boolean unconditional) {}

    private void show(Displayable d)
    {
        Display.getDisplay(this).setCurrent(d);
    }

    private void exit()
    {
        nézet.stop();
        show(null);
        destroyApp(true);
        notifyDestroyed();
    }

    public void commandAction(Command cmd, Displayable disp)
    {
        if ( cmd == vége )      exit();
    }
}
```

Egy labdának ismernie a képernyő méretét, amelyen mozoghat. Az értékeket a konstruktorban adhatjuk meg. Tudni kell az aktuális pozíció koordinátáit, a labda

méretét és a labda megjelenését (képét)[10]. Két művelet szükséges: a labda mozgatása megadott irányba (`mozog`), illetve a labda kirajzolása (`rajzol`). A mozgatás során figyelni kell arra, hogy a labda ne hagyja el a képernyő területét.

```
import javax.microedition.lcdui.*;

public class LabdaObj
{
    private int     maxx, maxy;
    private int     x, y;
    private Image   kép;
    private int     méret;

    public LabdaObj(int maxx, int maxy)
    {
        this.maxx = maxx;   this.maxy = maxy;
        try
        {
            kép = Image.createImage("/res/labda.png");
            méret = kép.getWidth();
        }
        catch (Exception e) { e.printStackTrace(); }
        x = (maxx - méret) / 2;     y = (maxy - y) / 2;
    }

    public void mozog(int dx, int dy)
    {
        x += dx;    y += dy;
        if ( x < 0 )   x = 0;
        else if ( x + méret > maxx )    x = maxx - méret;
        if ( y < 0 )   y = 0;
        else if ( y + méret > maxy )    y = maxy - méret;
    }

    public void rajzol(Graphics g)
    {
        g.drawImage(kép, x, y, Graphics.TOP | Graphics.LEFT);
    }
}
```

[10] A kép betöltésekor figyelni kell az esetleges hibákra, ezért fel kell készülni kivételekre. Most nem teszünk semmit hiba esetén, mert tudjuk, hogy biztosan betölthető a kép, egyébként valamilyen képet létre kellene hozni.

A `LabdaCanvas` osztály megvalósítja a `Runnable` interfészt, hogy egy szálat tudjon futtatni a `run` művelet segítségével. Hivatkozik egy labdára, tartalmazza a mozgatás irányát, illetve tudni kell, hogy a szál fut-e. A `start` és `stop` műveletekkel indíthatjuk el, illetve állíthatjuk le a mozgatást (szálat). A többi művelet mindegyike átdefiniált művelet, amelyek megvalósítása értelemszerű.

```
import javax.microedition.lcdui.*;

public class LabdaCanvas extends Canvas implements Runnable
{
    private static final int    periódus = 20;
    private boolean     fut;
    private int         dx, dy;
    private LabdaObj    labda;

    public LabdaCanvas()
    {
        fut = false;
        dx = dy = 0;
        labda = new LabdaObj(getWidth(), getHeight());
    }

    public void start()
    {
        fut = true;
        new Thread(this).start();
    }

    public void stop()      { fut = false; }

    protected void paint(Graphics g)
    {
        g.setColor(0xFFFFFF);
        g.fillRect(0,0,getWidth(), getHeight());
        labda.rajzol(g);
    }

    public void run()
    {
        long        idő = System.currentTimeMillis();
        int         eltelt;
        while ( fut )
        {
```

```
            if ( dx != 0 || dy != 0 )
            {
                labda.mozog(dx, dy);
                repaint();
            }
            eltelt = (int)(System.currentTimeMillis() - idő);
            if ( eltelt < periódus )
                try { Thread.sleep(periódus - eltelt); }
                catch (Exception e) {}
            idő = System.currentTimeMillis();
        }
    }

    public void keyPressed(int keycode)
    {
        if ( keycode >= KEY_NUM0 && keycode <= KEY_NUM9)
        {
            switch ( keycode )
            {
                case KEY_NUM8:  dx = 0; dy = 1;     break;
                case KEY_NUM2:  dx = 0; dy = -1;    break;
                case KEY_NUM4:  dx = -1; dy = 0;    break;
                case KEY_NUM6:  dx = 1; dy = 0;     break;
                case KEY_NUM5:  dx = dy = 0;        break;
                default:        break;
            }
        }
        else
        {
            int gc = getGameAction(keycode);
            switch ( gc )
            {
                case DOWN:  dx = 0; dy = 1;     break;
                case UP:    dx = 0; dy = -1;    break;
                case LEFT:  dx = -1; dy = 0;    break;
                case RIGHT: dx = 1; dy = 0;     break;
                case FIRE:  dx = dy = 0;        break;
                default:    break;
            }
        }
    }
}
```

A módosítsuk úgy a programot, hogy a labda mozgatása során csak a szükséges területet rajzoljuk újra, illetve a labda addig mozogjon, amíg a gombot nyomva tartjuk, a felengedéskor álljon le!

Ennek érdekében a `LabdaObj` osztály három művelettel egészítjük ki, amelyek megadják a rajzolandó téglalap bal-felső sarkának két koordinátáját és méretét.

```
public class LabdaObj
{
    ...
    public int bfx()    { return Math.max(0, x - 1); }

    public int bfy()    { return Math.max(0, y - 1); }

    public int méret()  { return méret + 2; }
}
```

A teljes képernyő újrarajzolását elkerülendő a `LabdaCanvas` osztály `run` műveletében a `repaint();` utasítást a

```
repaint(labda.bfx(), labda.bfy(), labda.méret(), labda.méret());
```

utasításra kell cserélnünk, a gombok kezelése pedig a következőképpen változik.

```
    public void keyPressed(int keycode)
    {
        if ( keycode >= KEY_NUM0 && keycode <= KEY_NUM9)
        {
            switch ( keycode )
            {
                case KEY_NUM8:  dx = 0; dy = 1;     break;
                case KEY_NUM2:  dx = 0; dy = -1;    break;
                case KEY_NUM4:  dx = -1; dy = 0;    break;
                case KEY_NUM6:  dx = 1; dy = 0;     break;
                default:    break;
            }
        }
        else
        {
            int gc = getGameAction(keycode);
            switch ( gc )
            {
                case DOWN:  dx = 0; dy = 1;     break;
                case UP:    dx = 0; dy = -1;    break;
                case LEFT:  dx = -1; dy = 0;    break;
```

```
            case RIGHT: dx = 1; dy = 0;     break;
            default:    break;
        }
    }
}

public void keyReleased(int keycode)
{
    dx = dy = 0;
}
```

A MIDP 2.0 verzióban bevezették `GameCanvas` osztályt, amely a `Canvas` osztály kiegészítése játékok támogatására szolgáló lehetőségekkel, például háttér (off-screen) grafikus puffer a rajzolások gyorsítására. Ezen kívül megjelent a `Sprite` osztály is, amellyel különböző alakzatokat kezelhetünk a képernyőn. Ez lehetőséget ad animált (több képből álló) elemek megjelenítésére a képernyőn, ezek ütközésének vizsgálatára, stb.

4. Adatkezelés

A mobiltelefonok korlátozott lehetőségei miatt az adatkezelést eleinte jelentősen korlátozták, nem lehetett a telefon fájlrendszeréhez közvetlenül hozzáférni, azaz fájlokat betölteni, azokba írni. Ugyanakkor az alkalmazásoknak biztosítani kellett állandó adatok tárolását és beolvasását, ezért bevezettek egy speciális mechanizmust erre a célra, amit rekord kezelő rendszernek, Record Management System, röviden RMS, neveznek. Egy másik lehetőséget adatok elérése a http kapcsolat biztosította.

Az eszközök fejlődése lehetővé tette fájlrendszerek létrehozását, kezelését, ezért a MIDP újabb változatai már támogatják ezt az adatkezelési módot is. További kapcsolati módok is megjelentek, így azok kezelése is szükségessé vált. Ennek érdekében létrehozták a `Connection` interfészt, ami megadja az összes ilyen jellegű kapcsolat, beleértve a http kapcsolatot is, közös felületét. Ebből származtatják a konkrét kapcsolatokat leíró interfészeket.

4.1. Record Management System

Ez a rendszer teszi lehetővé, hogy a MIDletek az adataikat állandó módon tárolhassák a telefonon, azaz az alkalmazás bezárása, és újraindítása után is rendelkezésre álljanak az adatok. A szolgáltatásokat a `javax.microedition.rms` csomag tartalmazza. A tárolás elve egy egyszerű rekord alapú adatbázis elvein alapul, ezt nevezzük RMS-nek.

4.1.1. Rekordtár

A csomag alapeleme a `RecordStore` osztály, amelynek segítségével rekordok gyűjteményét tárolhatjuk. Az osztály egy elemét a továbbiakban *rekordtárnak* nevezzük. Egy MIDlet által létrehozott rekordtár csak a MIDlet által érhető el, a telefon egyéb alkalmazásai ahhoz nem férhetnek hozzá[1]. A rekordtár elhelyezkedéséről a telefonon semmit sem tehetünk fel, a környezet felelős annak kezeléséért. Ha az alkalmazást

[1]Ebből következik, hogy az a tárolási mód nem alkalmas alkalmazások közötti kommunikációra. MIDP 2.0-tól kezdődően megengedett, hogy egy csoportba tartozó alkalmazások bizonyos szabályozások mellett ugyanazt a rekordtárat elérjék.

`openRecordStore(String rsn, boolean b)`	: Megnyitja az adott nevű rekordtárat. : Ha `b=true`, szükség esetén létrehozza. : Statikus, a visszaadott érték a rekordtár.
`deleteRecordStore(String rns)`	: Megszünteti az adott nevű rekordtárat.
`closeRecordStore()`	: A rekordtár bezárása.
`getNumRecords()`	: Megadja a rekordok számát.
`addRecord(byte[] r, int s, int h)`	: Felveszi `r[s..s+h-1]`-et új rekordként.
`deleteRecord(int id)`	: Törli az adott rekordot.
`getRecord(int id)`	: Visszaadja az adott rekordot tömbként.
`setRecord(int id,byte[] r,int s,int h)`	: Átírja az adott rekordot.

4.1. táblázat. A `RecordStore` osztály fontosabb műveletei

töröljük a telefonról, akkor az általa létrehozott rekordtárak is törlődnek. Egy alkalmazás több rekordtárat is használhat, ekkor azok nevének különbözniük kell. A nevek legfeljebb 32 jel hosszúak lehetnek, a kis- és nagybetűk megkülönböztetettek.

Definíció szerint a futtató környezetnek biztosítania kell, hogy egy rekordtáron végzett műveletek megszakíthatatlanok legyenek, biztosítva ezzel a konzisztenciát. Ugyanakkor ha egy alkalmazás több szálon keresztül kezeli a rekordtárat, akkor az alkalmazáson belüli konzisztencia fenntartása a program feladata[2]. A fontosabb műveleteket tartalmazza a 4.1. táblázat. Rekordtárak használatakor hiba esetén kivételeket vált ki a környezet, amelyek kezelésére a programban fel kell készülni.

4.1.2. Rekord

A rekordtár elemeit rekordoknak nevezzük, azonban ezek csak bájt tömbök. A `java.io` csomag elemei használhatóak a programban arra, hogy elemeket ilyen formátumra alakítsunk. Egy `DataOutputStream`-be ágyazott `ByteArrayOutputStream` segítségével hozhatjuk létre a megfelelő tömböt. A `DataOutputStream` lehetővé teszi elemi adatok kiírását a beágyazott bájt sorozatba, a `ByteArrayOutputStream` `toByteArray` műveletével pedig ebből előállíthatjuk a tömböt. Betöltéskor az input típusú elemeket kell használni.

Egy rekordtáron belül a rekordokat egy egész szám azonosít egyértelműen. Az első bejegyzés azonosítója 1, azaz ez lehet a legkisebb azonosító. Egy rekordhoz rendelt azonosító értéke monoton nő, ahogy újabb rekordokat veszünk a rekordtárhoz, azaz, ha az utolsó rekord azonosítója n, akkor a következőnek felvett rekord azonosítója $n + 1$ lesz. Rekordok törlésekor az azonosító is megszűnik, így a rekordtáron belül az azonosítók nem feltétlen egyesével követik egymást.

[2]A környezet sorrendbe állítja a hozzáféréseket, ami olvasás esetén nem okoz gondot, de íráskor az egyik művelet felülírhatja a másik művelet eredményét, amire figyelni kell.

4.1.3. Rekordtár elemeinek felsorolása

A rekordtár elemeit nem csak az azonosítókon keresztül érhetjük el. Lehetőség van az elemeket bejárni a `RecordEnumaration` interfész megvalósításával. Ilyen elemet a `RecordStore` osztály `enumerateRecords` műveletével hozhatunk létre. A művelet paramétereiben megadhatunk egy szűrőt (`RecordFilter`) a tárban szereplő rekordok részhalmazának kiválasztására és egy megelőzési relációt (`RecordComparator`) a sorrend szabályozására. A harmadik paraméterben adhatjuk meg, hogy a felsorolást időszerűsítse-e a rendszer, ha a használat közben a rekordtár módosulna. Szűrőnek és megelőzési relációnak megadhatjuk az üres (`null`) értéket, ekkor az összes rekordot megkapjuk, a sorrend pedig definiálatlan.

Az elemek elérésére szolgálnak a `nextElement()` és `previousElement()` műveletek. A `hasNextElement()` és `hasPreviousElement()` függvényekkel ellenőrizhetjük, hogy az adott irányban van-e még elem.

A `RecordEnumeration` objektumot a használat után a `destroy()` művelettel szüntethetjük meg. Ezt meg kell hívni, hogy felszabadítsuk a lefoglalt erőforrásokat. Értelemszerűen a megszüntetés után a bejárót nem lehet használni.

4.1.4. Példa alkalmazás

A 3.5.7. pontban bemutatott programot egészítsük ki úgy, hogy az áruk adatait a MIDlethez tartozó rekordtárban tároljuk! Ennek érdekében a `Cikkek` és a `Cikk` osztályokat kell módosítanunk a következők szerint.

1. A `Cikk` osztályban be kell vezetnünk két műveletet, amelyek segítségével egy áru adatait egy bájtokat tartalmazó tömbbe írjuk (`kiír`), illetve az adatokat beállíthatjuk egy tömb alapján (`feltölt`). Ennek a legegyszerűbb módja, ha tömböt egy megfelelő ByteArrayStreambe ágyazzuk, amit egy DataStreambe lehet helyezni. Egy DataStream esetén pedig stringek és egészek írására, illetve olvasására rendelkezésre állnak megfelelő műveletek.

2. Annak érdekében, hogy a streameket használhassuk az osztályban importálni kell a `java.io` csomagot.

3. A `Cikkek` osztályban importálni kell a `javax.microedition.rms` csomagot, hogy az adatokat az RMS segítségével tárolni tudjuk.

4. Meg kell adnunk a tárolásra szánt rekordtár nevét (`rsnév` statikus attribútum).

5. A konstruktorban az áruk tömbjét és listáját fel kell töltenünk, amire vezessük be a `cikkekTöltése` műveletet.

6. Cikkek töltésekor meg kell nyitni a megfelelő rekordtárat, és ennek összes elemét beolvasni, áruvá alakítani, és a tömbhöz, illetve a listához fűzni.

7. Kilépéskor az `exit` műveletben tárolni kell az árukat a `cikkekMentése` művelet segítségével.

8. Tárolás során a rekordtárat először ürítjük (töröljük), annak érdekében, hogy az elemek folytonos indexeléssel kerüljenek bele. Ezután az üres tárolóhoz fűzzük hozzá a tömb elemeit[3].

Ezek értelmében a következő változtatások szükségesek a `Cikk` osztályban.

```
import java.io.*;

public class Cikk
{

    ...

    public byte[] kiír() throws IOException
    {
        ByteArrayOutputStream   bs = new ByteArrayOutputStream();
        DataOutputStream        ds = new DataOutputStream(bs);
        ds.writeUTF(név);
        ds.writeInt(ár);
        ds.writeUTF(pénznem);
        ds.writeInt(mennyiség);
        ds.writeUTF(leírás);
        ds.writeInt(prioritás);
        ds.flush();
        byte[]  bytes = bs.toByteArray();
        ds.close();
        bs.close();
        return bytes;
    }

    public void feltölt(byte [] b) throws IOException
    {
        ByteArrayInputStream    bs = new ByteArrayInputStream(b);
        DataInputStream         ds = new DataInputStream(bs);
        név = ds.readUTF();
        ár = ds.readInt();
        pénznem = ds.readUTF();
        mennyiség = ds.readInt();
```

[3]Ebben az esetben a törlés elhanyagolható, ugyanis nem tudunk elemet törölni a listából. Ezért a megfelelő elemek felülírása, és az extra elemek hozzáfűzése megfelelő lenne.

```
        leírás = ds.readUTF();
        prioritás = ds.readInt();
        ds.close();
        bs.close();
    }
}
```

A `Cikkek` osztály változásai a következők.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.Vector;
import javax.microedition.rms.*;

public class Cikkek extends MIDlet implements CommandListener
{
    private static final String rsnév = "CikkekRS";

    ...

    public Cikkek()
    {
        ...
        cikkekTöltése();
    }


    private void exit()
    {
        cikkekMentése();
        show(null);
        destroyApp(true);
        notifyDestroyed();
    }

    private void cikkekTöltése()
    {
        RecordStore rs = null;
        Cikk         áru;
        try
        {
            rs = RecordStore.openRecordStore(rsnév, true);
            for ( int i = 1; i <= rs.getNumRecords(); i++ )
            {
```

```
                áru = new Cikk();
                áru.feltölt(rs.getRecord(i));
                áruk.addElement(áru);
                lista.append(áru.neve(), null);
            }
            rs.closeRecordStore();
        }
        catch (Exception e) { e.printStackTrace(); }
    }

    private void cikkekMentése()
    {
        RecordStore rs = null;
        try
        {
            RecordStore.deleteRecordStore(rsnév);
            rs = RecordStore.openRecordStore(rsnév, true);
            byte[]  b;
            for ( int i = 0; i < áruk.size(); i++ )
            {
                b = ((Cikk)áruk.elementAt(i)).kiír();
                rs.addRecord(b, 0, b.length);
            }
            rs.closeRecordStore();
        }
        catch (Exception e) { e.printStackTrace(); }
    }
}
```

Az `Adatlap` osztály nem változik, hiszen az egy áru megjelenítésére, szerkesztésére szolgál, ezért a tárolással kapcsolatos tevékenységek nem érintik.

4.2. Külső kapcsolatok

Az előzőekben tárgyalt RMS nem használható arra, hogy alkalmazások egymásnak adatot továbbítsanak. A programok közötti külső kapcsolatok kezelésére vezették be a `javax.microedition.io` csomagot, amelyben az általános kapcsolat leírására szolgál a `Connnection` interfész. Ez teszi lehetővé például http kapcsolat létrehozását, illetve a `javax.microedition.io.file` csomag segítségével a telefon fájlrendszerének elérését és használatát.

A `Connection` interfész egyetlen műveletet definiál, a `close()` műveletet, amellyel egy kapcsolatot bezárhatunk. Az egyes konkrét kapcsolatokra jellemző műveletet az

ebből származtatott interfészek adják meg[4]. Ezek közül az internetes kapcsolatok kezelésére szolgál a `HttpConnection` interfész, illetve fájlokat kezelhetünk az opcionális JSR-75 modulban található `FileConnection` interfész segítségével[5].

Kapcsolatokat a `Connector` gyártó osztály statikus műveletei segítségével hozhatunk létre, nyithatunk meg. A kapcsolatot egy URL segítségével adhatjuk meg, amelynek formátuma a következő:

`protokoll:cím;paraméterek`.

Ebből a paraméterek rész (sőt a cím is) elmaradhat. A protokoll szerint kerül kiválasztásra a kapcsolat jellege, ennek megfelelő típusúként kell kezelni a létrejött kapcsolatot. Internetes kapcsolat esetén a protokoll értéke `http`, fájl esetén `file`.

Az egyik alapvető művelet a kapcsolat létrehozása az `open` művelet segítségével. Ennek paramétere az URL, amit követhet az elérési mód szabályozása (olvasás, írás, olvasás és írás), illetve a lejárati idő kezelésének figyelése. Alapértelmezett elérési mód az olvasás és írás (`Connector.READ_WRITE`), az ettől eltérő módot meg kell adni (`READ` vagy `WRITE`). A létrehozott kapcsolatot a konkrét interfész műveletei segítségével kezelhetjük.

A `Connector` osztály lehetőséget ad a kapcsolatban szereplő elemek adatainak közvetlen elérésére is. Erre szolgálnak az `openInputStream`, `openOuputStream`, illetve `openDataInputStream`, `openDataOutputStream` műveletek, amelyeknél csak az URL szerepel paraméterként. Ekkor csak adatok olvasása, írása lehetséges, az esetleges ellenőrzések, létrehozások elmaradnak. (Az `open` műveletben létrehozott kapcsolat azonos műveleteivel ez két lépésben tehető meg, ha egyéb tevékenység is szükséges.)

4.2.1. Fájlkezelés

A JSR-75-ös kiegészítés tartalmazza a fájlok kezeléséhez szükséges FileConnection opcionális csomagot[6]. Ennek használatához a `javax.microedition.io` csomag mellett importálni kell a `javax.microedition.io.file` csomagot.

Fájlok elérésére az előzőek alapján a `Connector` osztály `open` művelete szolgál. Például a telefon gyökérkönyvtárában[7] (`C:`) található `adat.txt` állományt elérhetjük a következőképpen:

[4] Az összes fajtát nem áll módunkban áttekinteni, azokat az olvasó megismerheti a súgón keresztül elérhető dokumentáció segítségével.

[5] A JSR-75 nem standard eleme a Java ME-nek, ezért például a NetBeans környezetben automatikusan nem érhető el a (külön letölthető) dokumentációja. Mielőtt valaki a programban `FileConnection` típusú elemeket használna, meg kell győződnie, hogy az eszköz támogatja-e a JSR-75 modult. Ezt a 2.5. pontban megadott módon ellenőrizheti.

[6] Ezen kívül ebben szerepel a PIM opcionális csomag is, amelynek segítségével a készüléken található személyes adatok (névjegyzék, naptár, ...) érhetők el. (Feltéve, hogy a telefon ezt engedi.)

[7] A gyökérkönyvtárak elnevezései telefononként eltérhetnek, ezért annak a nevéről nem tehetünk fel semmit sem, viszont lekérdezhető, amint később látni fogjuk.

`file:///C:/adat.txt`

Az általunk használt emulátor esetén a gyökérkönyvtár neve `root1`, ami a számítógépen a

home`/j2mewtk/2.5.2/appdb/DefaultColorPhone/filesystem`

könyvtárban található, ahol *home* a felhasználó "home könyvtára".

Amint azt az előzőekben láttuk, fájlok kezelése során fontos a gyökérkönyvtár lekérdezése. Az is előfordulhat, hogy több ilyennel rendelkezik egy készülék (például kártya is van benne), ekkor a szükséges a megfelelő elem kiválasztása is.

A gyökérkönyvtárak felsorolására szolgál a `FileSystemRegistry` osztály statikus `listRoots()` művelete. Ez egy felsorolást (`Enumeration`) ad vissza, amit a szokásos `nextElement` és `hasMoreElements` műveletek segítségével járhatunk be, ahogy azt a következő kódrészlet szemlélteti.

```
Enumeration e = FileSystemRegistry.listRoots();
while ( e.hasMoreElements() )
{
    String root = (String)e.nextElement();
    ...
}
```

Fájlok kezelése során figyelnünk kell a kivételek kezelésére. Majdnem minden `Connection` típusú objektumra vonatkozó művelet dobhat kivételt, ezért ezeket `try catch` blokkokban használjuk.

Az `open` művelettel megnyitott kapcsolatot `FileConnection` típusúvá kell alakítanunk (ahogy azt az alábbi kódrészlet mutatja), annak érdekében, hogy a 4.2. táblázat műveletei alkalmazhatóak legyenek arra.

```
try
{
    FileConnection fc = (FileConnection)Connector.open(
                                        "file:///C/adat.txt");
    if ( !fc.exists() ) fc.create();
    ...
}
catch (IOException ioe)
{
    ...
}
finally
{
    ...
    fc.close();
}
```

`void create()`	: Létrehozza a fájlt.
`void mkdir()`	: Létrehozza a könyvtárt.
`void delete()`	: Eltávolítja a fájlt, könyvtárat.
`boolean exists()`	: Ellenőrzi, hogy létezik-e.
`boolean isDirectory()`	: Ellenőrzi, hogy könyvtár-e.
`Enumeration list()`	: Megadja a könyvtár látható tartalmát.
`InputStream openInputStream()`	: Megnyitja a stream-et.
`OutputStream openOutputStream()`	: Megnyitja a stream-et.
`DataInputStream openDataInputStream()`	: Megnyitja a stream-et.
`DataOutputStream openDataOutputStream()`	: Megnyitja a stream-et.

4.2. táblázat. A `FileConnection` interfész fontosabb műveletei

A hibák elkerülése végett a legritkább esetben adjuk meg közvetlenül stringként a fájl nevét. A fájlokat rendszerint egy böngésző segítségével választjuk ki. Ilyen böngészőt viszonylag könnyű készíteni, de ezt sem kell megtenni, ha valaki a NetBeans által biztosított `FileBrowser` osztályt használja. Ennek érdekében importálni kell az `org.netbeans.microedition.lcdui.pda` csomagot, és a megfelelő könyvtárat (`jar` állomány) be kell állítanunk a fejlesztőkörnyezet könyvtárai közé[8].

4.2.2. Példa fájlkezelésre

Egy egyszerű feladaton keresztül mutatjuk be a `FileConnection` típusú kapcsolatok használatát. A példában egy határidő naplót készítünk mobiltelefonra. Ebben lehetőségünk lesz egy adott időponthoz emlékeztető szöveget rendelni, és ezeket tárolni, időpont alapján kikeresni egy feljegyzést[9].

Hozzunk létre a szokásos módon egy Java ME alkalmazást, amelyben legalább MIDP 2.0 profilt válasszunk. Ehhez vegyünk fel egy `Jegyzet` MIDletet, és a bejegyzések kezeléséért felelős `Elem` osztályt.

Egy `Elem` típusú objektumban az időpontot és a leírást kell tárolnunk, és ezeket kell lekérdezni, beállítani. Kijelzésre az időpontot fogjuk használni, és ennek karakteres, listába kerülő formáját adja meg a `toString` művelet. Így a következő osztályhoz jutunk.

[8] A `FileBrowser` osztály mellett egyebeket (PIM kezelés) is tartalmaz a csomag. Miután a futtatható `jar` állományba mobil alkalmazások esetén a könyvtárakat is be kell fordítani, kisebb kódhoz jutunk egy saját böngésző elkészítésével. Ebben a kiválasztáson kívül további műveleteket (új fájl létrehozása, törlés) is megvalósíthatunk.

[9] Természetesen a feladat megoldható RMS segítségével is, azonban mi most a szemléltetés érdekében választjuk ezt az utat. Ezzel a megközelítéssel, az is megvalósítható, hogy másik alkalmazás is használja a napló adatait, illetve máshova (pl. számítógép) is el tudjuk azt juttatni.

```
import java.util.Date;

public class Elem
{
    private Date    idő;
    private String  leírás;

    public Elem()
    {
        idő = new Date();   leírás = "";
    }

    public Elem(Date idő, String leírás)
    {
        this.idő = idő; this.leírás = leírás;
    }

    public Date ideje()             { return idő; }
    public String leírása()         { return leírás; }
    public void újIdő(Date idő)     { this.idő = idő; }
    public void újLeírás(String s)  { leírás = s; }
    public String toString()        { return idő.toString(); }
}
```

Az egyszerűség érdekében a napló adatait a gyökérkönyvtár `jegyzet.dta` nevű állományában fogjuk tárolni. A gyökérkönyvtár lekérdezésére szolgál a `gyökér()` művelet, ami az azonos nevű attribútumban tárolja a könyvtár nevét. Csak abban az esetben kérdezi le a könyvtár nevét az előzőekben megismertek szerint, ha az még a programban nem ismert. Ha több gyökér is rendelkezésre áll, akkor az első választjuk ki.

A napló bejegyzéseit egy `Vector` objektumban tároljuk, és egy `List` elemben jelenítjük meg. A listából választhatunk elemet (`választ` parancs), amit megtekinthetünk, módosíthatunk, törölhetünk. Az `új` parancs segítségével vehetünk fel új bejegyzést.

Egy bejegyzés megjelenítésére és szerkesztésére szolgál az `adatok Form`. Ebben egy `DateField` típusú elemmel kezelhetjük az időpontot, a hozzá tartozó szöveget pedig egy `TextField` segítségével. Ehhez a formhoz kell a törlési parancsot is felvennünk a jóváhagyás és a visszalépés mellé.

Az alkalmazás indulásakor a `töltés` eljárás segítségével beolvassuk a napló adatait. Kilépéskor elindítunk egy szálat (a MIDlet ezért valósítja meg a `Runnable` interfészt), amelynek segítségével kiírjuk az adatokat. Miután a kilépés egy parancs kezelését jelenti, kötelező szálat indítanunk, hogy elkerüljük a felhasználói felület esetleges blokkolódását.

A naplóból történő választás, illetve az elemek módosítása a cikkek kezelését végző programmal analóg módon történik. (A form most sokkal egyszerűbb, ezért nem kellett külön osztály létrehoznunk a megjelenített elemek és adatok összekötésére.) Ezek alapján adódik a következő MIDlet.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;
import javax.microedition.io.*;
import javax.microedition.io.file.*;
import java.io.*;

public class Jegyzet extends MIDlet implements CommandListener, Runnable
{
    private boolean     paused = false;
    private List        lista;
    private Command     új;
    private Command     kilép;
    private Command     választ;
    private Form        adatok;
    private DateField   idő;
    private TextField   leírás;
    private Command     ok;
    private Command     törlés;
    private Command     vissza;
    private Vector      események;
    private int         index;
    private String      gyökér;

    public Jegyzet()
    {
        lista = new List("Események", List.IMPLICIT);
        új = new Command("Új", Command.SCREEN, 0);
        kilép = new Command("Vége", Command.EXIT, 0);
        választ = new Command("Választ", Command.ITEM, 0);
        lista.addCommand(új);
        lista.addCommand(kilép);
        lista.setSelectCommand(választ);
        lista.setCommandListener(this);
        adatok = new Form("Esemény");
        idő = new DateField("Időpont", DateField.DATE_TIME);
        leírás = new TextField("Esemény", "", 256, TextField.ANY);
        adatok.append(idő);
```

```
        adatok.append(leírás);
        ok = new Command("OK", Command.OK, 0);
        törlés = new Command("Töröl", Command.OK, 0);
        vissza = new Command("Vissza", Command.BACK, 0);
        adatok.addCommand(ok);
        adatok.addCommand(törlés);
        adatok.addCommand(vissza);
        adatok.setCommandListener(this);
        események = new Vector();
        gyökér = null;
        töltés();
    }

    public void startApp()
    {
        if ( !paused )  show(lista);
        paused = false;
    }

    public void pauseApp()  { paused = true; }

    public void destroyApp(boolean unconditional)   {}

    private void show(Displayable d)
    {
        Display.getDisplay(this).setCurrent(d);
    }

    private void exit()
    {
        new Thread(this).start();
        show(null);
        destroyApp(true);
        notifyDestroyed();
    }

    public void commandAction(Command cmd, Displayable d)
    {
        if ( cmd == új )            új();
        else if ( cmd == kilép )    exit();
        else if ( cmd == választ )  szerkeszt();
        else if ( cmd == ok )       felvesz();
        else if ( cmd == törlés )   töröl();
```

```
    else if ( cmd == vissza )   show(lista);
}

private void új()
{
    index = -1;
    idő.setDate(null);
    leírás.setString("");
    show(adatok);
}

private void szerkeszt()
{
    index = lista.getSelectedIndex();
    Elem    e = (Elem)események.elementAt(index);
    idő.setDate(e.ideje());
    leírás.setString(e.leírása());
    show(adatok);
}

private void felvesz()
{
    try
    {
        if ( index == -1 )
        {
            Elem    e = new Elem(idő.getDate(), leírás.getString());
            események.addElement(e);
            lista.append(e.toString(), null);
        }
        else
        {
            Elem    e = (Elem)események.elementAt(index);
            e.újIdő(idő.getDate());
            e.újLeírás(leírás.getString());
            események.setElementAt(e, index);
            lista.set(index, e.toString(), null);
        }
        show(lista);
    }
    catch (Exception e) {}
}
```

```
    private void töröl()
    {
        if ( index != -1 )
        {
            események.removeElementAt(index);
            lista.delete(index);
        }
        show(lista);
    }

    public void run()   { mentés(); }

    private void töltés()
    {
        FileConnection  fc = null;
        String          fn = "file:///" + gyökér() + "jegyzet.dta";
        try
        {
            fc = (FileConnection)Connector.open(fn, Connector.READ);
            DataInputStream is = fc.openDataInputStream();
            boolean         vanelem = true;
            while ( vanelem )
                try
                {
                    Elem    e = new Elem(new Date(is.readLong()),
                                                 is.readUTF());
                    események.addElement(e);
                    lista.append(e.toString(), null);
                }
                catch ( EOFException eof )  { vanelem = false; }
            is.close();
            fc.close();
        }
        catch ( Exception e )   {}
    }

    private void mentés()
    {
        FileConnection  fc = null;
        String          fn = "file:///" + gyökér() + "jegyzet.dta";
        try
        {
            fc = (FileConnection)Connector.open(fn, Connector.READ_WRITE);
```

```
            if ( fc.exists() )  fc.delete();
            fc.create();
            put(fc.openDataOutputStream());
            fc.close();
        }
        catch ( Exception e )   {}
    }

    private void put(DataOutputStream os) throws Exception
    {
        for ( int i = 0; i < események.size(); i++ )
        {
            Elem    e = (Elem)események.elementAt(i);
            os.writeLong(e.ideje().getTime());
            os.writeUTF(e.leírása());
        }
        os.close();
    }

    private String gyökér()
    {
        if ( gyökér == null )
        {
            Enumeration  roots = FileSystemRegistry.listRoots();
            if ( roots.hasMoreElements() )
                gyökér = (String)roots.nextElement();
        }
        return gyökér;
    }
}
```

4.2.3. HTTP kapcsolatok

A kapcsolatok egységes kezelése biztosítja, hogy az internetes (http) kapcsolatok kezelése lényegében megegyezik a fájlok esetében megismertekkel. Itt további műveletek szolgálnak a kapcsolat jellegéből adódó speciális igények kielégítésére. Ebben az esetben meg lehet (és kell is) vizsgálni, hogy sikerült-e a kapcsolatot felépíteni, azaz van-e visszajelzés, illetve, hogy megfelelő típusú elemhez kapcsolódunk-e.

A http kapcsolatok leírására szolgál a `HttpConnection` interfész, így értelemszerűen ekkor a `Connector.open` műveletével megkapott értéket ilyen típusúvá kell alakítanunk. Egy interneten keresztüli XML fájl elérését szemlélteti a következő kódrészlet, amelyben `parse` végzi az XML elemzést.

```
HttpConnection  con = null;
InputStream     is = null;
try
{
    con = (HttpConnection)Connector.open(xmlfile);
    if ( con.getResponseCode() == HttpConnection.HTTP_OK )
    {
        if ( con.getType().equals("application/xml") )
        {
            is = con.openInputStream();
            parse(is);
        }
    }
}
catch (Exception e) {}
finally
{
    try
    {
        if ( is != null )   is.close();
        if ( con != null )  con.close();
    }
    catch (Exception e) {}
}
```

A kapcsolat jellegéből adódóan ebben az esetben fájlok létrehozására, törlésére, illetve könyvtárak kezelésére nincs lehetőség. Az adatok elérésén (stream-ek) kívül a kapcsolat kezelésére, ellenőrzésére szolgáló műveletek állnak rendelkezésre (4.3. táblázat), illetve a válaszok kódértékeit megadó konstansok.

`int getResponseCode()`	: HTTP válasz státusz kódja.
`String getType())`	: A tartalom típusa.
`String getEncoding()`	: A kódolás leírója.
`long getLength()`	: A fájl hossza.
`String getHost()`	: Az URL host része.
`InputStream openInputStream()`	: Megnyitja a stream-et.
`OutputStream openOutputStream()`	: Megnyitja a stream-et.
`DataInputStream openDataInputStream()`	: Megnyitja a stream-et.
`DataOutputStream openDataOutputStream()`	: Megnyitja a stream-et.

4.3. táblázat. A `HttpConnection` néhány művelete

5. XML kezelés

Alkalmazások igen széles köre használja adatok tárolására, továbbítására, kommunikációra az XML-t. Éppen ezért Java ME platformon is képesnek kell lennünk XML állományokat feldolgozására, azaz XML formátumban tárolt információ kinyerésére. Ennek alapvető eleme, az XML fájl elemzése, amelyhez egy elemző, *parser* szükséges.

Jellegük szerint háromféle XML elemzőt különböztethetünk meg.

Modell alapú Ezek a feldolgozók a teljes állományt beolvassák, és felépítenek egy fát. Ezt a fát lehet a rendelkezésre álló műveletekkel bejárni, és így kinyerni a szükséges információt. Ehhez a teljes fát tárolni kell, ami meglehetősen erőforrásigényes, ezért az ilyen feldolgozók nem használhatóak Java ME környezetben.

Push típusú Működés közben beolvassák az XML fájlt, és a feldolgozás közben eseményeket hoznak létre, amelyeket a feldolgozó alkalmazásnak figyelnie (értelmeznie, kezelnie) kell. Ezek az elemzők egyszerűen használhatóak, ellenben még mindig jelentős memóriára van szükségük nagyobb XML fájl esetén.

Pull típusú Ezek a feldolgozók az állomány kis részét olvassák csak be egyidejűleg, ugyanis csak addig olvasnak, amíg egy még fel nem dolgozott elemet (*tag*et) találnak. Ezért ebben az esetben csak lépésenként dolgozzuk fel az XML állományt, és az egyes lépések során le lehet kérdezni az aktuális elemet és annak tulajdonságait. Az előzőekkel ellentétben, ebben az esetben az alkalmazás kér új adatokat a feldolgozótól. Értelemszerűen ez a megközelítés illeszkedik legjobban a mobileszközök lehetőségeihez.

5.1. Push típusú XML elemzés

Annak ellenére, hogy ez a megközelítés jóval nagyobb erőforrásigénnyel bír, mint a pull típusú elemzés, kifejlesztettek Java ME platformra ilyen típusú elemzőt[1]. A megfelelő API minden XML feldolgozásához szükséges elemet tartalmaz.

[1] Valószínűleg történeti és kompatibilitási okokból, ugyanis a halózati szolgáltatások igénybevétele XML-en keresztül történik, és ahhoz push típusú elemzőt használtak. Java ME-ben a JSR 172 kiegészítés tartalmazza a hálózati szolgáltatások kezelését, és annak része a JAXP (Java API for XML processing).

`void startDocument()`	: A dokumentum eleje.
`void endDocument()`	: A dokumentum vége.
`void startElement(String uri, String ln, String qName, Attributes attr)`	: Egy elem (tag) kezdete. :
`void endElement(String uri, String ln, String qN)`	: Egy elem vége.
`void characters(char[] ch, int start, int hossz)`	: Egy elem szöveges része.

5.1. táblázat. A `DefaultHandler` osztály fontosabb műveletei

A `javax.xml.parsers` csomagban található a SAX, push típusú elemző. Egy SAX feldolgozót (`SAXParser`) a `SAXParserFactory` osztály példányának segítségével hozhatunk létre. Egy példányt az osztály statikus `newInstance` műveletével kaphatunk, amelynek `newSaxParser` művelete állítja elő az elemzőt. Az elemző `parse` műveletével dolgozhatjuk fel az XML állományt[2]. Ennek első paramétere az XML állomány (InputStream vagy InputSource típusú objektumként), második paramétere a SAX eseményeket (push típusú elemzés eseményeit) feldolgozó `DefaultHandler` objektum. Egy ilyen objektum alapértelmezett megvalósítását adja az események kiváltásakor meghívott műveleteknek, csak azokat fel felüldefiniálnunk, amelyekre szükségünk van. Néhány fontosabb műveletet tartalmaz az 5.1. táblázat.

Látható, hogy ebben a megközelítésben egy megfelelő `DefaultHandler` objektumot kell csak létrehozni. Ez a szükséges műveletek újradefiniálásán kívül rendszerint egy vermet is tartalmaz, amelyben a tagek (`qName`) neveit tároljuk. Ez a verem teszi lehetővé az aktuális helyzet vizsgálatát.

Egy egyszerű példán szemléltetjük a JAXP API használatát. Ebben egy XML állományban tároljuk diákok adatait: nevüket, ETR azonosítójukat és egy jegyet. Ezt a fájlt töltjük be, a diákok neveit egy listába tesszük, ahonnan kiválasztás után megtekinthetjük, és módosíthatjuk egy diák adatait. Az adatokat ugyanabba az XML állományba visszaírhatjuk. Az XML fájl kiválasztásához a NetBeans megfelelő kiegészítő csomagjában található `FileBrowser` osztályt használjuk.

A feldolgozandó XML fájl formátuma a következő:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <student>
        <name>Fa Ede</name>
        <etr>FAEOAAT.ELTE</etr>
        <mark>5</mark>
    </student>
```

[2]Külön szálban indítsuk, hogy ne blokkoljuk a programot.

```
    <student>
        <name>Kis Pál</name>
        <etr>KIPBBAAT.ELTE</etr>
        <mark>3</mark>
    </student>
    <student>
        <name>Nagy Éva</name>
        <etr>NAECXAT.ELTE</etr>
        <mark>4</mark>
    </student>
</root>
```

Hozzunk létre a szokásos módon NetBeans alatt egy Java ME alkalmazást CLDC 1.1 konfigurációval és MIDP 2.0 (vagy magasabb) profájllal. A projekthez vegyük hozzá a fájl kiválasztáshoz szükséges könyvtárat. A projekt tulajdonságaiban válasszuk a `Libraries & Resources` pontot, itt az `Add Library...` gombot, és a listából vegyük fel a `NetBeans MIDP Components PDA` elemet. A `FileBrowser` osztály használatához a megfelelő helyen (esetünkben a MIDlet osztályában) importálni kell az `org.netbeans.microedition.lcdui.pda` csomagot.

A projekthez vegyük fel a hallgatók adatait tároló `Student` osztályt. Ennek megvalósítása a következő.

```
public class Student
{
    private String      name;
    private String      etr;
    private int         mark;

    public Student()
    {
        name = etr = "";     mark = 0;
    }

    public String getName()             { return name; }
    public String getEtr()              { return etr; }
    public int getMark()                { return mark; }
    public void setName(String name)    { this.name = name; }
    public void setEtr(String etr)      { this.etr = etr; }
    public void setMark(String str)
    {
        try { mark = Integer.parseInt(str); }
        catch (Exception e) { mark = 0; }
    }
}
```

A fájlból beolvasott hallgatók adatait egy `Vector` segítségével tároljuk. A minősítések elkerülése érdekében a tömböt a `StudentData` osztályon belül helyezzük el.

```
import java.util.Vector;

public class StudentData
{
    private Vector      students;

    public StudentData()
    {
        students = new Vector();
    }

    public void clear()         { students.removeAllElements(); }

    public int size()           { return students.size(); }

    public void add(Student s)  { students.addElement(s); }

    public Student get(int i)   { return (Student)students.elementAt(i); }
}
```

Ezután készítsük el az XML elemzéshez szükséges `DefaultHandler` feladatra szabott változatát a `StudentXMLHandler` osztályt. EZ hivatkozik a MIDletre és a hallgatók adataira. Tartalmazza ezen kívül az aktuálisan olvasott hallgató adatait, és egy vermet a tag-nevek tárolására. Az 5.1. táblázatban szereplő műveleteket kell megvalósítani.

Egy XML dokumentum kezdetén az adatokat kell törölni, a dokumentum végén a vermet, és értesíteni kell a MIDletet az elemzés befejeződéséről. Egy tag kezdetén tárolni kell a veremben a nevét, és ha diák kezdődik, fel kell venni egy új elemet aktuálisként. Egy tag zárásakor a verem tetejét el kell távolítani, és diák esetén az aktuális elemmel bővíteni a tömböt. Szöveges rész feldolgozása csak akkor szükséges, ha van diák, és a verem tetején egy diákra vonatkozó tag szerepel.

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import java.util.Stack;

public class StudentXMLHandler extends DefaultHandler
{
    private Students        midlet;
    private StudentData     data;
```

```
private Student         student;
private Stack           stack;

public StudentXMLHandler(Students midlet, StudentData data)
{
    this.midlet = midlet;
    this.data = data;
    student = null;
    stack = new Stack();
}

public void startDocument() throws SAXException
{
    data.clear();  student = null;
}

public void startElement(String uri, String ln, String qn,
                         Attributes attr) throws SAXException
{
    if ( qn.equals("student") ) student = new Student();
    stack.push(qn);
}

public void endElement(String uri, String ln, String qn)
                                throws SAXException
{
    if ( qn.equals("student") )
     {
        data.add(student);
        student = null;
    }
   stack.pop();
}

public void characters(char[] ch, int start, int hossz)
                                throws SAXException
{
    String  str = new String(ch, start, hossz).trim();
    if ( str.length() == 0 )    return;
    if ( stack.empty() || student == null )        return;
    String  tag = (String)stack.peek();
    if ( tag.equals("name") )       student.setName(str);
    else if ( tag.equals("etr") )   student.setEtr(str);
```

```
        else if ( tag.equals("mark") )  student.setMark(str);
    }

    public void endDocument() throws SAXException
    {
        stack.removeAllElements();
        midlet.parseComplete();
    }
}
```

Ezek után a `Students` típusú MIDlettel foghatjuk össze az alkalmazás eddigi elemeit. Az osztály az eddigi ismeretek alapján elkészíthető, csak arra kell figyelnünk, hogy a fájlokra vonatkozó parancsok kezelését (XML elemzése, illetve kiírása) külön szálban indítsuk el elkerülendő a felhasználói felület blokkolását. (Ugyanazt a visszalépési parancsot – `back` – rendeljük a formhoz és a listához is, ezért a parancs kezelése során az eddigiektől eltérően nem csak a parancsot, hanem a forrását figyelnünk kell.)

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import org.netbeans.microedition.lcdui.pda.FileBrowser;
import javax.xml.parsers.*;
import javax.microedition.io.file.*;
import java.io.*;

public class Students extends MIDlet implements CommandListener, Runnable
{
    private boolean     paused;
    private FileBrowser browser;
    private Command     exit;
    private Command     back;
    private Command     ok;
    private Command     select;
    private Command     save;
    private List        list;
    private Form        form;
    private TextField   name;
    private TextField   etr;
    private TextField   mark;

    private StudentXMLHandler   handler;
    private StudentData         data;
```

```
public Students()
{
    paused = false;
    browser = new FileBrowser(Display.getDisplay(this));
    exit = new Command("Vége", Command.EXIT, 0);
    browser.addCommand(exit);
    browser.setCommandListener(this);
    list = new List("Diákok", List.IMPLICIT);
    back = new Command("Vissza", Command.BACK, 0);
    ok = new Command("OK", Command.OK, 0);
    select = new Command("Választ", Command.ITEM, 0);
    save = new Command("Mentés", Command.OK, 0);
    list.addCommand(select);
    list.addCommand(save);
    list.addCommand(back);
    list.setCommandListener(this);
    name = new TextField("Név: ", "", 64, TextField.ANY);
    etr = new TextField("ETR: ", "", 16, TextField.ANY);
    mark = new TextField("Jegy: ", "", 1, TextField.NUMERIC);
    form = new Form("Diák adatai", new Item[] { name, etr, mark});
    form.addCommand(ok);
    form.addCommand(back);
    form.setCommandListener(this);
    data = new StudentData();
    handler = new StudentXMLHandler(this, data);
}

public void startApp()
{
    if ( !paused )  show(browser);
    paused = false;
}

public void pauseApp()  { paused = true; }

public void destroyApp(boolean unconditional)   {}

private void show(Displayable d)
{
    Display.getDisplay(this).setCurrent(d);
}
```

```
private void exit()
{
    show(null);
    destroyApp(true);
    notifyDestroyed();
}

public void commandAction(Command cmd, Displayable disp)
{
    if ( cmd == FileBrowser.SELECT_FILE_COMMAND )
        new Thread(this).start();
    else if ( cmd == List.SELECT_COMMAND || cmd == select )
        view();
    else if ( cmd == exit )         exit();
    else if ( cmd == back )
    {
        if ( disp == list )         show(browser);
        else                        show(list);
    }
    else if ( cmd == ok )           update();
    else if ( cmd == save )         new SaveThread().start();
}

public void parseComplete()
{
    list.deleteAll();
    for ( int i = 0; i < data.size(); i++ )
        list.append(data.get(i).getName(), null);
    show(list);
}

public void run()
{
    try
    {
        SAXParser   parser =
                SAXParserFactory.newInstance().newSAXParser();
        FileConnection  fc = browser.getSelectedFile();
        InputStream     is = fc.openInputStream();
        parser.parse(is, handler);
        is.close();
        fc.close();
    }
```

```
        catch (Exception e) { e.printStackTrace(); }
    }

    private void view()
    {
        int     index = list.getSelectedIndex();
        if ( index == -1 )  return;
        Student s = data.get(index);
        name.setString(s.getName());
        etr.setString(s.getEtr());
        mark.setString("" + s.getMark());
        show(form);
    }

    private void update()
    {
        int     index = list.getSelectedIndex();
        Student s = data.get(index);
        s.setName(name.getString());
        s.setEtr(etr.getString());
        s.setMark(mark.getString());
        list.set(index, s.getName(), null);
        show(list);
    }

    private class SaveThread extends Thread
    {
        public void run()
        {
            try
            {
                FileConnection  fc = browser.getSelectedFile();
                fc.truncate(0);
                OutputStreamWriter  osw = new OutputStreamWriter(
                                          fc.openOutputStream(), "UTF-8");
                osw.write("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n");
                osw.write("<root>\n");
                for ( int i = 0; i < data.size(); i++ )
                {
                    Student s = data.get(i);
                    osw.write("    <student>\n");
                    osw.write("        <name>" + s.getName() + "</name>\n");
                    osw.write("        <etr>" + s.getEtr() + "</etr>\n");
```

```
                    osw.write("        <mark>" + s.getMark() + "</mark>\n");
                    osw.write("    </student>\n");
                }
                osw.write("</root>\n");
                osw.flush();
                osw.close();
                fc.close();
            }
            catch (Exception e) { e.printStackTrace(); }
        }
    }
}
```

5.2. Pull típusú XML elemzés

Létezik szabadon letölthető pull típusú elemző. Ilyen például a kXML2, amelyet kifejezetten mobileszközökre fejlesztettek ki. Ez megvalósítja az XML Pull API-ban található XmlPullParser interfészt, ami a pull típusú elemzők közös felületét adja meg. Használatához le kell tölteni a megfelelő könyvtárat[3]. A kXML2 csomaggal kapcsolatos dokumentumok, anyagok letölthetők a
`http://kxml.sourceforge.net/kxml2/`
címről.

A kXML2 használatához be kell állítani a NetBeans könyvtárai közé a letöltött `jar` fájlt, a kódban pedig importálni kell az `org.kxml2.io` csomagot. A csomag elemzésre szolgáló osztálya a `KXmlParser` osztály. Létre kell hozni egy ilyen típusú objektumot. Ennek `setInput` műveletével adhatjuk meg az elemzendő XML fájlt. A legegyszerűbb bejárást a `next` művelet biztosítja, aminek visszaadott értéke az utolsó olvasáshoz tartozó esemény típusa. Az osztály definiál konstansokat a megfelelő típusokhoz. (Az esemény típusa lekérdezhető a `getEventType` művelettel is.) Néhány fontosabb műveletet ad meg az 5.2. táblázat.

A kXML2 használatát az előző példával szemléltetjük. Ebben az esetben a projekthez fel kell még vennünk a kXML2 könyvtárat is. A `Libraries & Resources` párbeszédablakban az `Add Jar/Zip...` gomb megnyomása után válasszuk ki a kXML2 könyvtár jar állományt (`kmxl2-min-2.3.0.jar`) onnan, ahová elhelyeztük[4].

Értelemszerűen az előző program nagy része megmarad, csak az XML elemzést kell megváltoztatni. Ennek érdekében az előző megoldásban használt `StudentXMLHandler` osztály helyett vezessük be a `StudentXMLParser` osztályt, amely a kXML2 pull típusú elemzőt használja majd. Ekkor a MIDletben szereplő `handler` objektumot cseréljük a

[3] `kxml2-min-#.jar`, illetve a teljes könyvtár `kxml2-#.jar`, ahol # a verziószámot jelenti (Ez jelenleg 2.3.0.

[4] Ez és a `FileBrowser`-hez szükséges könyvtár szerepel ezután a listában.

`void setInput(...)`	: Az XML fájl megadása.
`int next()`	: A következő elemet olvassa.
`int nextTag()`	: A következő tag-et olvassa.
`int getEventType()`	: Az utolsó esemény típusa.
`char[] getTextCharacters(...)`	: A szöveget adja meg.
`String getName()`	: Az aktuális tag neve.
`int getAttributeCount()`	: A tag attribútumainak száma.
`String getAttributeName(int i)`	: Az i. attribútum neve.
`String getAttributeValue(int i)`	: Az i. attribútum értéke.
`void skipSubTree()`	: Az aktuális részfa átugrása.

5.2. táblázat. A `KXmlParser` osztály fontosabb műveletei

megfelelő típusú `parser` objektumra. A konstruktorban értelemszerűen ezt az objektumot hozzuk létre. Az objektum konstruktorának paraméterei megegyeznek az előző megoldásban megismerttel. Miután nem használjuk, a `javax.xml.parsers` csomag importálását is elhagyhatjuk a MIDletből.

Az eddigieken kívül a MIDlet XML elemzésért felelős `run` művelete változik meg a következők szerint.

```
public void run()
{
    try
    {
        FileConnection  fc = browser.getSelectedFile();
        InputStream     is = fc.openInputStream();
        parser.parse(is);
        is.close();
        fc.close();
    }
    catch (Exception e) { e.printStackTrace(); }
}
```

A `StudentXMLParser` osztály megvalósítása az előző megoldásnak megfelelően zajlik, csak most egy pull típusú elemző műveleteit kell használni. Az elemzéshez csak az `org.kxml2.io` csomag importálása szükséges. Esetünkben a fájlt is használjuk, ezért a `java.io` csomagot is használjuk.

A karakterek helyes kezelése érdekében az input megadásakor meg kell adnunk a kódolást is. (Ezt a push típusú elemző esetén nem kellett külön megtenni.) További eltérés, hogy nem vezetünk be külön műveletet a dokumentum végének kezelésére, azt a feldolgozási ciklus után tesszük meg.

```
import org.kxml2.io.*;
import java.io.*;
import java.util.Stack;

public class StudentXMLParser
{
    private Students        midlet;
    private StudentData     data;
    private Student         student;
    private Stack           stack;

    public StudentXMLParser(Students midlet, StudentData data)
    {
        this.midlet = midlet;   this.data = data;   student = null;
        stack = new Stack();
    }

    public void parse(InputStream is) throws Exception
    {
        int     eventtype;
        KXmlParser   parser = new KXmlParser();
        parser.setInput(is, "UTF-8");
        eventtype = parser.getEventType();
        while ( eventtype != KXmlParser.END_DOCUMENT )
        {
            switch ( eventtype )
            {
                case KXmlParser.START_DOCUMENT: startDocument();      break;
                case KXmlParser.START_TAG:      startElement(parser); break;
                case KXmlParser.END_TAG:        endElement(parser);   break;
                case  KXmlParser.TEXT:          textElement(parser);  break;
                default:                        break;
            }
            eventtype = parser.next();
        }
        stack.removeAllElements();
        midlet.parseComplete();
    }

    public void startDocument()
    {
        data.clear();  student = null;
    }
```

```
    public void startElement(KXmlParser parser)
    {
        String  tag = parser.getName();
        if ( tag.equals("student") )    student = new Student();
        stack.push(tag);
    }

    public void endElement(KXmlParser parser)
    {
        if ( parser.getName().equals("student") )
        {
            data.add(student);
            student = null;
        }
        stack.pop();
    }

    public void textElement(KXmlParser parser)
    {
        String  str = parser.getText().trim();
        if ( str.length() == 0 )    return;
        if ( stack.empty() || student == null )        return;
        String  tag = (String)stack.peek();
        if ( tag.equals("name") )       student.setName(str);
        else if ( tag.equals("etr") )   student.setEtr(str);
        else if ( tag.equals("mark") )  student.setMark(str);
    }
}
```

Látható, hogy a megoldás csak abban különbözik a push típusú elemzéstől, hogy most nekünk kell gondoskodni az XML fájl bejárásáról. Azonban pont ez ad lehetőséget az erőforrások kíméletesebb használatára, hiszen a vezérlés az ellenőrzésünk alatt marad. (Ebben az egyszerű példában nem látszanak az ebből adódó lehetőségek, mert az egész állományt egyszerre feldolgoztuk. Remélhetőleg az olvasó számára is nyilvánvaló, hogy nagyobb XML fájl feldolgozása esetén a pull típusú elemzővel egy-egy rész feldolgozása után meg lehet állni, míg ez a push típusú feldolgozás esetén nem lehetséges.)