SZAKDOLGOZAT

Renn Antal Gábor

Debrecen

2007

Debreceni Egyetem

Informatika Kar

Alkalmazott Matematika és Valószínûségszámítás Tanszék

MOBILTELEFONOK JAVA PROGRAMOZÁSA

Témavezető:

Bátfai Norbert

Számítástechnikai munkatárs

Készítette:

Renn Antal Gábor

Programtervező informatikus

Debrecen

2007

Tartalomjegyzék

Bevezetés

Mobilvilág, trendek, száraz adatok, statisztikák

Magyarországon 2006. november végén 100 lakosra 96,9% mobiltelefon-előfiztés jutott, az aktív SIM-kártyák száma alapján. Ha ez a növekedési tendencia tovább folytatódik, hazánk beléphet azon európai országok „elit klubjába", ahol több a mobilelőfizetések száma, mint ahány lakos van. Ilyen országok: Luxemburg, Csehország, Izland, Norvégia, Egyesült Királyság, Svédország, Olaszország, Szlovénia. Azonban a tényleges mobilhasználók száma jóval alacsonyabb egy 2006. májusi felmérés szerint, mely alapján a lakosság 76%-a rendelkezett mobiltelefonnal. Ez az eredmény az EU-átlag körüli érték, tehát Európa középmezőnyében foglal helyet hazánk.

A Föld teljes lakosságát tekintve 2004 végére a mobilhasználók száma elérte a 1,5 milliárdot. Mára ennyi a száma azon telefonkészülékeknek, melyek Java ME platformú mobil adatszolgáltatást biztosítnak!

Az adatok és felmérések igazolják, hogy a fejlett távközlési piacok a mobilértékesítés terén a telítettség felé közelítenek. Új trendek jelennek meg: az eladások további növelése már nem tartható, ezért mennyiségi helyett minőségi változásra van szükség. Új szolgáltatásokkal, és a legújabb technológiák bevezetésével próbálják meg az előfizetőket megnyerni maguknak, illetve felvenni a versenyt a konkurenciával szemben a szolgáltatók.

- világméretű hálózatok és szolgáltatások
- analóg átviteli rendszerek felváltása digitális eszközökkel
- új mozgó távközlési szolgáltatások gyors fejlődése és terjedése
- a média, az informatika és a távközlés technológiai bázisának közeledése egymáshoz

A mobilgyártók különféle célközönségnek szánt készülékeket dobnak piacra: kamerás, divat, sőt üzleti vállalkozást támogató, illetve többfunkciós mobilokat. A Nokia 2004-ben 27 új telefonnal szállt versenybe a piacon.

Mind a mobiltelefonok terjedése, a célzott előfizetők megnyerése, és az új technológiák megjelenése a mobilszoftverek szerepének növekedéséhez vezetnek. Bár egy átlagos felhasználót valószínűleg hidegen hagy, hogy milyen operációs rendszer fut a készülékén, és

hogy milyen alkalmazásokat képes futtatni, mégis a mobil teljesítményét és értékét nagyban befolyásolják ezen tulajdonságai.

A mobiltelefonok szinte már számítógéppé, médialejátszóvá, játékkonzollá, kamerává alakulnak. Rengeteg multimédiás és információ elérési lehetőséget biztosítanak: zenék, videók lejátszása, kamera, rádió, adattárolás különböző memóriakártyákon, számítógéphez való csatlakozás, internet elérés, készülékek közötti átviteli technológiák (infraport, Bluetooth) stb.

A mobil alkalmazások terén igen nagy teret hódított magának a Java. A Java ME platform segítségével rengeteg készülékre fejleszthetünk alkalmazásokat, melyek segítségével az adott eszköz képességeit is kihasználhatjuk. Java alkalmazások (főleg játékok) millió tölthetőek le a „push" technikának köszönhetően az interneten, SMS-en, WAP-on keresztül.

Java történelem, avagy hogyan lett a Tölgyfából Kávé

A Java nyelv története 1991-ben kezdődött, és James Gosling nevéhez fűződik, aki a Sun MicroSystems egy csoportjának vezetője volt. Ez a csoport dolgozott a „Green" projekten, melynek lényege, hogy egy mini nyelvet tervezzenek kommunikációs eszközök programozására. A tervezés során szembesülniük kellett az eszközök hátrányaival: nem túl gyorsak, kis memóriával rendelkeznek, ezért a nyelvnek kicsinek, valamint a kódnak igen hatékonynak kell lennie. De a legnagyobb kihívást az jelentette, hogy a különféle gyártók más és más processzorokat építenek saját termékükbe, tehát fontos volt a létrejövő nyelv architektúra-függetlensége. Niklaus Wirth ötletét használták fel, mely szerint egy közbenső kódot kell generáltatni egy virtuális gépre. Így ez a közbenső, ún. bájtkód minden olyan gépen futtatható, ahol megtalálható a megfelelő interpreter. Jelszavuk: „Write once, run everywhere" (=írd meg egyszer, futtasd mindenhol). Gosling és csapata UNIX-os „előéletük" miatt egy C++ alapú nyelvet hozott létre, mely az „Oak" nevet kapta, Gosling irodájának ablaka előtt növő tölgyfáról. De mivel „Oak" programnyelv már létezett, a nyelv az újabb névválasztása közben elfogyasztott nagy mennyiségű kávéról végül Java lett.

A Java nyelv tulajdonságai, mitől is finom a Kávé

Egyszerű

Mivel a nyelv a C++ leegyszerűsített változata, ezért azok számára, akik rendelkeznek némi programozói ismerettel, az elsajátítása nem okoz gondot. (Persze nem egy kifejezetten tanuló nyelv!) Az egyszerűségét tükrözi, hogy nincs benne például többszörös öröklődés, a C++-ban amúgy is felesleges struct, a kódot bonyolulttá tevő goto ugró utasítás, illetve a C-ben programozói hibák halmazát eredményező mutató (pointer). De például megmaradt a futásidejű hibák kezelésére a kivételkezelő, a felhasználó által már nem használt memóriaterületek automatikus felszabadítását végző szemétgyűjtő (garbage collector).

Objektum orientál

A Java OO szemléletű, majdnem tiszta OO nyelv, a C++ objektum orientált tulajdonságaival rendelkezik. Objektumokat, azaz az egyes osztályokba tartozó egyedeket hozhatunk létre. Az osztály tartalmazza az adatreprezentációt és az adatokon operáló metódusokat. Egy Java alkalmazást osztályok készítésével és újrafelhasználásával készíthetünk el. Egyszeres öröklődést vall! Láthatósági eszközrendszere négy szintű: nyilvános (public), védett (protected), privát (private), illetve csomagszintű láthatóság (package).

Architektúra-független, hordozható

A Java nem tartalmaz architektúra- vagy implementációfüggő elemeket, tehát a nyelvi elemek minden környezetben ugyanúgy vannak specifikálva. A futtatás feltétele, hogy az adott architektúrára már van implementált virtuális gép, beleértve a rendszerkönyvtárakat is. Ezek a könyvtárak valósítják meg az operációs rendszerhez kötődő feladatokat (pl. az alkalmazások grafikus kezelői felületét).

Többszálú

A processzor jobb kihasználása érdekében a programok nagy része vezérlési szálakra bontható, melyek a párhuzamos végrehajtást teszik lehetővé. A Java a többszálú programozáshoz szükséges kölcsönös kizárást nyelvi szinten valósítja meg. A szálak létrehozásához, szinkronizációjához a rendszerkönyvtár a Thread osztályt tartalmazza. A virtuális gép a szálak futtatásához prioritásos preemtív ütemezőt alkalmaz.

Az egymással kommunikáló szálakra bontott feladat áttekinthető, könnyen implementálható, és a számítógép jobb kihasználtságát eredményezi.

Dinamikus, hozzáférhető

A Java-t úgy tervezték, hogy a továbbfejlesztése könnyű legyen, az osztálykönyvtárak szabadon bővíthetőek.

Az elterjedésében nagy szerepet játszott a hozzáférhetősége. Rengeteg forráskód, specifikáció, példaprogram, egyéb dokumentum, sőt fejlesztő környezet ingyenesen elérhető.

A zárt forráskódú Java eszközök miatt sok támadás érte a Sun-t, valószínűleg ennek is köszönhető, hogy 2007-ben a Sun Microsystems a Java SE-t, Java ME-t, Java EE-t GPL (General Public License, Általános Nyilvános Licenc) licenc alatt nyílt forráskódúvá, tehát szabad szoftverré teszi.

Java verziók:

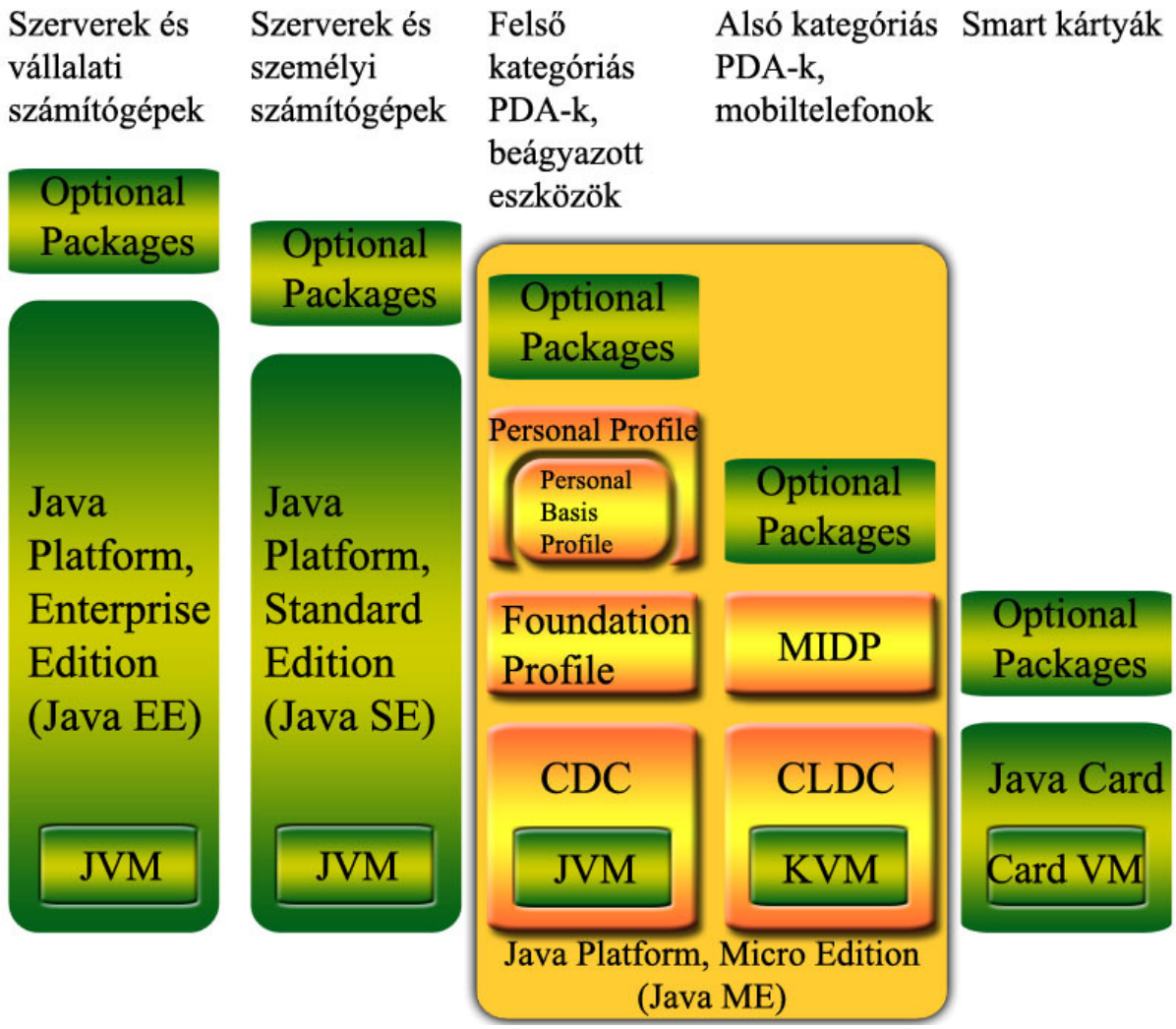
- 1.0 (1996) – Java virtuális gép és osztálykönyvtárak első verziója
- 1.1 (1997) – osztályok egymásba ágyazásnak lehetősége, belső osztály fogalma
- 1.2 (1998) – Java 2
- 1.3 (2000)
- 1.4 (2002)
- 5 (2004) – 1.5, kódneve: Tiger, új ciklusmegoldások, generikus, autoboxing
- 6 (2006) – 1.6.0, kódneve: Mustang, kiterjesztett nyomkövetés, felügyeleti megoldások, szkriptnyelvek integrációja, támogatás grafikus felület tervezéséhez
- 7 (? 2008 ?) - kódneve: Dolphin

Java Virtuális Gép (JVM)

Klasszikus esetben, amikor a számítógépen elindítunk egy programot, akkor a program utasításai olyan formában vannak tárolva, hogy azt az adott processzor megérti. Ez az ún. natív futtatható állomány. Ennek az igen nagy hátránya, hogy nem hordozható. Ha egy RISC processzorra lefordított programot akarunk egy CISC processzorral rendelkező rendszeren futtatni, az ezen a platformon nem fog működni.

A virtuális gép a Java program és az adott gép platformja közötti réteget képezi. Ez egy absztrakt számítógép, amely a valódiakhoz hasonlóan rendelkezik utasításkészlettel, kezeli a memóriaterületet futás közben, garantálja a biztonságot, vezérli a végrehajtási szálakat.
A JVM a bájtkódra lefordított Java programokat képes értelmezni. A bájtkód jelenti az átmenetet a forrásfájl és a natív kód között.
Kétféle futtatás lehetséges: a virtuális gép a bájtkódot futtatás közben értelmezi, tehát a végrehajtás pillanatában történik a gépi kódra való fordítás (JIT – Just In Time). A másik megoldás, hogy még a végrehajtás előtt fordítja le az egész programot natív kódra a JVM, majd ezután futtatja ezt (HotSpot fordítás).
Mindkettőnek hátrány van: az értelmezéses megoldás a program teljes futása közben veszít teljesítményéből. Az előfordítós megoldásnál a futtatás előtti fordítás miatt tolódik ki az indulási idő. De a mai optimalizált HotSpot f ordítónak köszönhetően ez az idő nem tűnik fel nagyon. Összességében a bájtkódú programok ezen hátrányok miatt még elmaradnak teljesítményüket tekintve a natív alkalmazások mögött.

Java Platform



Java SE

Standard Edition. Általános célú programok és hagyományos desktop alkalmazások fejlesztése, tesztelése, futtatása, valamint a Java-t támogató böngészőkben az applet-ek futtatása. Az alkalmazások speciális típusa szerverként fut, hálózati klienseket kiszolgálva. A platform megfelelően nagy memóriával és teljesítményű processzorral rendelkezik.

Java EE

Enterprise Edition. Alapját a Java SE jelenti. Vállalati, üzleti alkalmazások szerver oldali fejlesztésére szánt változat. Fontos eleme a servlet, amely a szerver oldalon fut, a szerver-futtatókörnyezet részeként, és a kliens felől érkező kérések kiszolgálását végzi.

Java Card

Kevés memóriával és kis teljesítményű processzorral rendelkező eszközökre (intelligens kártyák) való fejlesztés. Digitális azonosítási megoldások. Elvárás, hogy az alkalmazások képesek legyenek együttműködni a különböző gyártók kártyáival → Java

Java ME

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

Java Micro Edition (Java ME), a legkisebb csésze kávé

Konfigurációk, profilok, a kávé alap ízesítői

A platform alapját egy konfiguráció képezi, amely magába foglalja a virtuális gépet és az alapfunkciók ellátását biztosító könyvtárakat. Erre épül az adott eszköz működését meghatározó profil. A profilok tartalmazzák például az alkalmazás életciklusára, a felhasználói felület készítésére és az adattárolásra implementált csomagokat. Az opcionális csomagok olyan lehetőségeket biztosítanak, amelyek nem férnek bele a konfigurációba és a profilba, de az adott eszköz támogatja, így ezen csomagok segítségével kihasználhatóak a hardver nyújtotta lehetőségek.

Konfigurációk, profilok és opcionális csomagok kombinációjából áll össze a Java futási környezete. Minden kombináció egy adott eszközcsoport képességeihez van optimalizálva, így az eredmény egy olyan platform, mely a céleszközök nyújtotta lehetőségek előnyeinek kihasználását biztosítja.

A Java ME két konfigurációt definiál: CDC (Connected Device Configuration) és CLDC (Connected Limited Device Configuration).

A CDC-t fejlettebb eszközökre tervezték, például kommunikátorok, PDA-k:

- 32 bites processzor
- 2 MB RAM – felejtő memória
- 2,5 MB ROM – nem felejtő memória
- fájlrendszerrel rendelkező operációs rendszer
- nagyobb hálózati sávszélesség

CLDC - Connected Limited Device Configuration

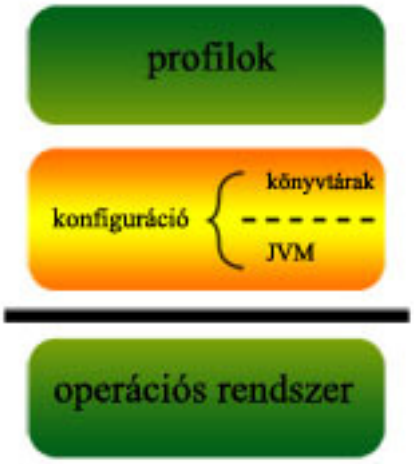
A CLDC-t korlátozott erőforrásokkal rendelkező készülékekre tervezték (pl. mobiltelefon):

- 16 vagy 32 bites processzor, minimum 16 MHz-es órajellel
- minimum 192 KB memóri
  - minimum 160 KB nem felejtő memória KVM és CLDC részére

- minimum 32 KB felejtő memória (KVM)

- alacsony energiafogyasztás (elem vagy akkumulátor)
- korlátozott átviteli sebességű hálózati kapcsolat

CLDC magas szintű architektúrája:



A konfiguráció csak azokat a legfontosabb könyvtárakat tartalmazza, és a virtuális gép csak azon legalapvetőbb szolgáltatásait nyújtja, melyeknek mindenféleképpen szerepelniük kell a Java ME környezet implementációjában. Ezek segítségével a platform méretét le lehet csökkenteni olyan szintre, hogy az minél több eszközre illeszkedjen, valamint kiegészíthető az eszköz funkcionalitását támogató API-kal.
A CLCD 1.1 a következő osztályokat tartalmazza:

- java.io – I/O
- java.lang – a Java nyelv alapját jelentő osztályok
- java.util – kollekciók, dátum és idő
- java.microedition.io – Generic Connection Framework (GFC)-ön alapuló hálózati támogatás

Kilobyte Virtual Machine (KVM)

A Java ME C nyelven írt virtuális gépe a KVM. Neve a pár 10 KB-os méretére utal. Olyan 16 vagy 32 bites processzorokra tervezték, melyek minimum 160 KB memóriával rendelkeznek a KVM és az osztálykönyvtárak részére. A kifejlesztésénél fontos volt, hogy a könyvtárak és a virtuális gép méretét minimalizálják, az alkalmazások végrehajtása közben a felmerülő memóriaigényt csökkentsék, és hogy a virtuális gép bizonyos részeit az eszközök egyéni tulajdonságai szerint konfigurálják.
Mindezek eredményeképpen a KVM alapkonfigurációja csak 50-80 KB között van, és a hatékony futáshoz mindössze pár tíz kilobájt dinamikus memóriára van szüksége. A KVM architektúrája nagymértékben hordozható, köszönhető ez annak, hogy a K virtuális gép C-ben íródott és így egyszerűen átvihető oda, ahol van C fordító.

Mobile Information Device Profile (MIDP)

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

- Motorola – specifikáció vezetője
- Nokia – felhasználói API
- Sun – időkezelés, perzisztens tárolás, hálózat, stb.

A MIDPEG által támasztott hardveres követelmények egy MID (Mobile Information Device) eszköz számára:

- **Kijelző:**
  - méret: 96*54 pixel
  - színmélység: 1 bit (legalább két színű kijelző)
  - torzítási arány: 1:1
- **Input, beviteli lehetőségek:**
  - egykezes billentyűzet (ITU-T)
  - kétkezes billentyűzet (QWERTY)
  - érintőképernyő

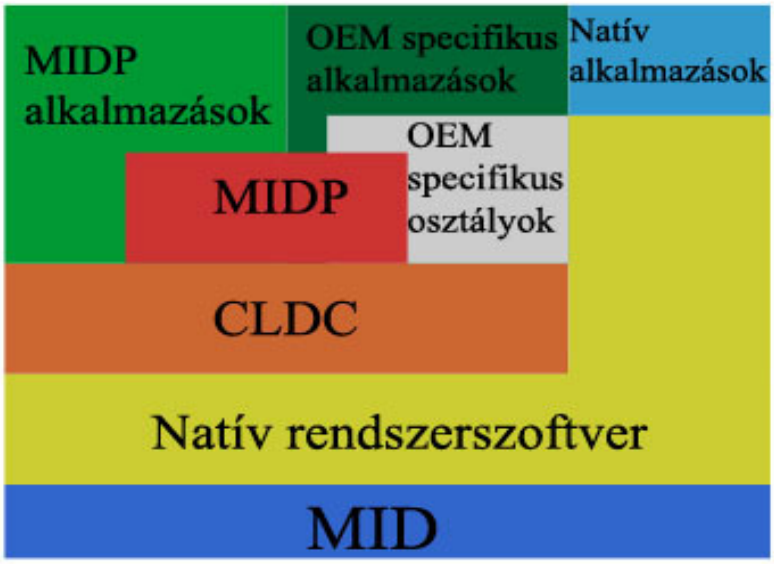
  valamelyikével kell rendelkeznie

- **Memória:**
  - 8 KB nem felejtő memória az alkalmazások által használt perzisztens adatok számára
  - 32 KB felejtő memória a Java futtatására
  - 128 KB nem felejtő memória a MIDP komponensek számára
- **Hálózat:**
  - vezeték nélküli
  - két irányú
  - korlátozott sávszélességgel

Mivel a MID eszközök operációs rendszere eltérő lehet, ezért a MIDP-nek vannak az eszköz szoftverére vonatkozó követelményei is:

- hardver kezelésére és a virtuális gép futtatására egy minimális kernel (megszakítás, kivételkezelés)
- alkalmazások számára fenntartott nem felejtő memória írása és olvasása
- az eszköz hálózati kapcsolatához való írási és olvasási hozzáférés
- a bittérképes grafikus kijelző kezelése
- felhasználói input kezelése
- az alkalmazás életciklusának kezelése

MIDP architektúra:



- a MIDP alkalmazás (MIDlet) csak a CLDC és MIDP specifikációkban megadott API-kat használja
- az OEM specifikus alkalmazások, olyan API-kat használnak, melyeket a gyártó az eszközhöz adott ki, így ezek az alkalmazások nem hordozhatóak
- a Natív alkalmazások nem Java-ban íródtak

A MIDP lefelé kompatibilis, azaz a jelenlegi 2.0-ás környezetben is futtathatóak az 1.0-ás verzió szerint készített alkalmazások. A MIDP 2.0 az 1.0 egy továbbfejlesztett, kibővített változata.

MIDP 2.0 újdonságai

Továbbfejlesztett felhasználói interface

A kibővített felhasználói felület nagyobb hordozhatóságot, bővíthetőséget biztosít. Az újdonságok interaktívabbá és könnyen kezelhetővé teszik a MIDP alkalmazásokat a felhasználók számára. A fejlesztők számára a képernyő elrendezésének jobb irányítását biztosítja a mobil eszközök saját képernyő méretén és egyéb attribútumon alapuló MIDP implementációnak köszönhetően. A képernyő egyes elemei rendelkeznek legkisebb és ajánlott mérettel. Minden MIDP implementáció használja az elrendezés, a képernyő és az elemek attribútumait az alkalmazás optimális megjelenítéséhez. Nagyobb bővíthetőséget biztosít, figyelemre méltó a Custom Item, mely segítségével a fejlesztők saját egyedi felhasználói interface komponenseket hozhatnak létre.

Média támogatás

A MIDP 2.0 tartalmazza az Audio Building Block (ABB)-t, mely a Mobile Media API (MMAPI)-nak is része, ami egy opcionális csomag a MIDP 2.0-hoz. A fejlesztők számára lehetővé teszi az audio eszközök, mint például hangok, hang szekvenciák és WAV fájlok hozzáadását a MIDP alkalmazásokhoz, a MMAPI használata nélkül. MMAPI implementációval rendelkező készülékeken lehetőségük van a fejlesztőknek további multimédiás tartalom használatára a MIDP alkalmazásokban, például videó lejátszására.

Játék támogatás

MIDP 2.0-ban megjelenő Game API a játékok fejlesztésének egy szabványos alapját képezi. A MIDP Game API játék-specifikus eszközöket tartalmaz, sprite-okat és tiled layer-eket, melyek lehetővé teszik az eszköz grafikus előnyeinek kihasználását. Ezek a komponensek egyszerűsítik a fejlesztést és a grafika és teljesítmény jobb kihasználtságát biztosíják.

Kiterjesztett összekapcsolhatóság

A MIDP 2.0 a HTTP-en kívül további vezető kapcsolati szabványokat tartalmaz, mint például HTTPS, datagram, socket, szerver socket, és a soros port kommunikáció. Ez a kiterjesztett összekapcsolhatóság egyszerű és szabványos utat biztosít a Java programozók előtt is ismert szoftver infrastruktúrák integrálásához.

Push architektúra

Lehetővé teszi, hogy a készülék egy szerverről információt, üzenetet kapjon, bizonyos események bekövetkezésékor, anélkül, hogy ezek érkezését ellenőrizni kellene. MIDP 2.0 push technológiája leegyszerűsíti a válasz küldését a különféle üzenetekre, illetve lehetővé teszi az adatállományok, vagy alkalmazások csatolását az üzenetekhez.

Over-the-air (OTA) ellátás

A MIDP 2.0-ban kötelező specifikációként szerepel. A MIDP specifikáció definiálja, hogyan lehet a MIDlet készleteket „felfedezni”, telepíteni, frissíteni és eltávolítani a mobil információs eszközökről.

End-to-end biztonság

A MIDP 2.0 egy robusztus, nyílt szabványú end-to-end biztonsági modellt tartalmaz, mely védi a hálózatot, az alkalmazást és mobil információs eszközt. A MIDP 2.0 támogatja a

HTTPS-t és kihasználja a már létező szabványokat, mint az SSL és a WTLS a titkosított adatok átviteléhez. MIDP 2.0-ban biztonsági tartományok védik az adatokat, az alkalmazásokat, a hálózatot és az eszköz erőforrásait a jogosulatlan hozzáféréssel szemben. A MIDlet készletek alapértelmezésben nem megbízhatóak, és nem megbízható tartományokhoz vannak hozzárendelve, melyeknek nincs hozzáférésük a kiváltságos funkcionalitásokhoz. A privilegizált eszközökhöz való hozzáféréshez a MIDlet készletet az adott mobil eszközön definiált, speciális tartományhoz kell hozzárendelni, valamint megfelelően aláírni az X.509 PKI biztonsági szabvány használatával. Azért, hogy egy aláírt MIDlet készeletet letöltsünk, telepítsünk és megszerezzük a hozzárendelt engedélyeket, megfelelően hitelesített kell hogy legyen.

Néhány érdekes opcionális kiegészítés

- Mobile Media API (JSR-135): hang és videó fájlok lejátszása és vezérlése
- Mobile 3D API (JSR-184): 3D-s grafikus alkalmazások készítése
- Bluetooth API (JSR-82): Bluetooth szerver és kliens kapcsolatok létrehozása
- Wireless Messaging API (WMA) (JSR 120, JSR 205): SMS küldése és fogadása MIDlet-ekből

MIDlet, MIDlet suite

A MID profilban készült alkalmazások a MIDlet-ek. A MIDlet-eket egy alkalmzásvezérlő rendszer (AMS – Application Management Software) menedzseli: betölti, aktiválja és deaktiválja. Valamint a felhasználónak lehetősége van a MIDlet törlésére, ha az szükségtelenné vált.

A MIDlet-eket szabványos JAR (Java Archive Resource) fájlba szokták csomagolni, ha több MIDlet van egy ilyen fájlban, akkor MIDlet készletről (MIDlet suite) beszélünk. Ezek többnyire együttműködő vagy hasonló feladatot ellátó MIDlet-ek, melyek osztozhatnak az erőforrásokon és hozzáférhetnek egymás információihoz, de más készletben (suite) lévőkhöz nem! A JAR fájlról egy ún. JAD leíró fájl (Java Application Descriptor) biztosít információkat a készülék számára. Ez alapján az AMS eldönti, hogy képes-e letölteni a JAR

fájlt, illetve, hogy az alkalmas-e az eszközön való futtatásra. A leíró fájlban szerepelnek kötelező és szerepelhetnek opcionális információk.

Kötelező információk:

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

Opcionális információk:

- **MIDlet-Description:** a MIDlet készlet leírása
- **MIDlet-Icon:** a MIDlet készlethez rendelhető ikon (.png kiterjesztésű fájl)
- **MIDlet-Info-URL:** a MIDlet készletről információt tartalmazó cím
- **MIDlet-Data-Size:** a MIDlet perzisztens adatainak tárolásához szükséges minimális memóriaigény bájtokban, alapértelmezésben ez 0

Minden MIDlet tartalmaz egy osztályt, mely a *javax.microedition.midlet.MIDlet* osztály leszármazottja (ezt kell kiterjesztenie), illetve további osztályokat, melyekre a MIDlet-nek szüksége van. A MIDlet nem rendelkezik *public stativ void main()* metódussal, az alkalmalmazásvezérlő elindítja a MIDlet egy egyedét, tehát létrehozza annak egy példányát.

A legegyszerűbb MIDlet:

```
import javax.microedition.midlet.*

public class egyszeruMIDlet extends MIDlet {

     public egyszeruMIDlet()
     public void startApp()
     public void pauseApp()
     public void destroyApp(boolean unconditional)
}
```

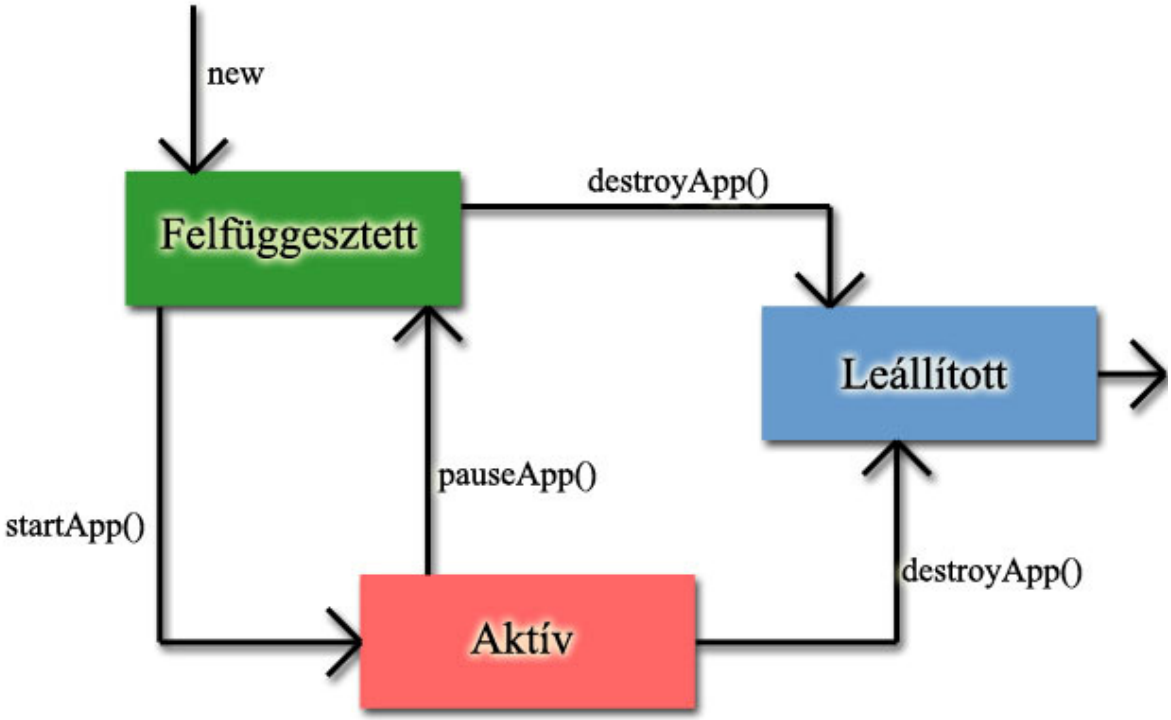
public egyszeruMIDlet() – a MIDlet konstruktora

public void startApp() – a MIDlet Aktív állapotba lép, ha nem sikerült elindítani, akkor a kiváltódik a MIDletStateChangeException kivétel. Ezt a metódust csak Felfüggesztett állapotból lehet meghívni

public void pauseApp() – a MIDlet Felfüggesztett állapotba kerül, bármilyen hiba hatására a MIDlet megszűnik. Csak Aktív állapotból lehet meghívni

public void destroyApp(boolean unconditional) – true argumentum esetén felszabítja a lefoglalt memóriaterületet és a futása mindenféleképpen befejeződik; false estén kiváltja a MIDletStateChangeException kivételt

A MIDlet életciklusa



Egy MIDlet három állapotban lehet:

- Felfüggesztett (Paused)
- Aktív (Active)

- Leállított (Destroyed)

A MIDlet osztály metódusainak segítségével értesíti az AMS a MIDletet az állapotváltozásokról.

- A MIDlet alapértelmezésben Leállított állapotban van, azaz nincs betöltve a JVM-be
- Aktiváláskor a MIDlet Felfüggesztett állapotba kerül, ilyenkor a MIDlet osztály betöltődik a JAR-ból a JVM-be, ekkor még nem foglal le erőforrásokat
- Az AMS a MIDlet startApp() metódusának meghívásával aktiválja azt. A metódus sikeres lefutás után, ha nem váltódott ki a MIDletStateChangeException kivétel, a MIDlet Aktív állapotba kerül
- A MIDlet futásának szüneteltetéséhez az AMS a pauseApp() metódust hívja meg, hatására a MIDlet Felfüggesztett állapotba kerül. Akkor is ebbe az állapotba kerül, ha a startApp() metódus hívása sikertelen volt, tehát kiváltódik a MIDletStateChangeException kivétel. Az állapot implementációját a különböző gyártók másképpen valósítják meg. (Pl. Nokia: csak akkor szünetelteti a futást, ha azt a MIDlet maga kéri. Motorola: ha a MIDlet eltűnik a képernyőről, például hívásnál, akkor Felfüggesztett állapotba kerül)
- A MIDlet befejezi a futását, az AMS meghívja a MIDlet destroyApp(boolean unconditional) metódusát. Ha ennek a paramétere false, kiváltódik a MIDletStateChangeException kivétel. Ha a paraméter true, az AMS mindenképpen törli a MIDlet-et

Egy egyszerű játék bemutatása

Miért pont egy játék? Talán legtöbben az informatikához nem értők közül a Java szóval, mint programozási nyelvvel, akkor találkoztak, amikor Java-t támogató mobiltelefont kezdtek el használni. Bizonyára a készülék nyújtotta eme szolgáltatását a játékok miatt kedvelik, a legtöbben ugyanis játékokat töltenek le rá. Persze a Java-nak ezt a változatát sok más dologra is lehet használni, a határt a képzeletünk és a kezünkben vagy zsebünkben lévő készülék korlátai szabják meg. És a legfontosabb amiért a középpontban egy játék áll: a leghasznosabb valamit úgy megtanulni, elsajátítani, hogy az közben szórakoztató is!

Manapság egy igényes, piacképes játék fejlesztése nem egy egyszemélyes, hanem komoly, több hónapos, sőt akár éves munka. De mobilra egy személy, viszonylag rövid idő alatt el tud készíteni egy szórakoztató kis játékot.

Ha megfigyeljük a mobiltelefonokon lévő játékokat, az embernek kissé olyan érzése támad, mintha már látta volna azokat, valahol korábban. Nem csoda, hiszen rengeteg pl. Commodore 64-re írt játék feldolgozása jelenik meg Java ME környezetben, hiszen ami egyszer siker volt, az valószínűleg most is az lesz. Továbbá a PC-khez viszonyítva, egy mobiltelefon multimédiás képessége és számítási kapacitása messze elmarad. Találkozhatunk logikai, felülnézetes kaland és logikai játékokkal.

A mobil játékok manapság jelentős piacra tettek szert. Mobiltelefonnal szinte mindenki rendelkezik, játszani, kikapcsolódni mindenki szeret. Gyakran unatkozik az ember, pl. a buszon iskolába vagy munkába menet, ilyenkor kellemes kikapcsolódást nyújthat egy ilyen kis egyszerű játék, mivel a telefonunk mindig kéznél van. Az áruk sem elhanyagolandó, míg egy PC-s vagy konzol játék ára több mint 10000 forint, addig egy mobil játék pár száz forint, és nem is kell érte a város másik felébe utazni, hogy megvegyük, egyszerűen csak letöltjük a szolgáltatónk WAP hálózatán keresztül.

A legtöbb játék pixelgrafikus, ezért a leggyakrabban tetszetős, rajzfilmszerű megjelenéssel találkozhatunk. Az eszköz képernyőjének felbontása meglehetősen kicsi, emiatt a játékelemek (pl. sprite) mérete csak pár tíz pixel. Ez is kedvet adhat azoknak, akik nem rendelkeznek különösebb grafikai érzékkel, egy ilyen pici elem elkészítése nem okoz gondot senkinek. Ha 3 dimenziós megjelenítést akarunk használni, figyelni kell, hogy a telefonban nincs erre a grafikai megoldásra szánt célhardver. De mivel ez, mint az informatika többi területe

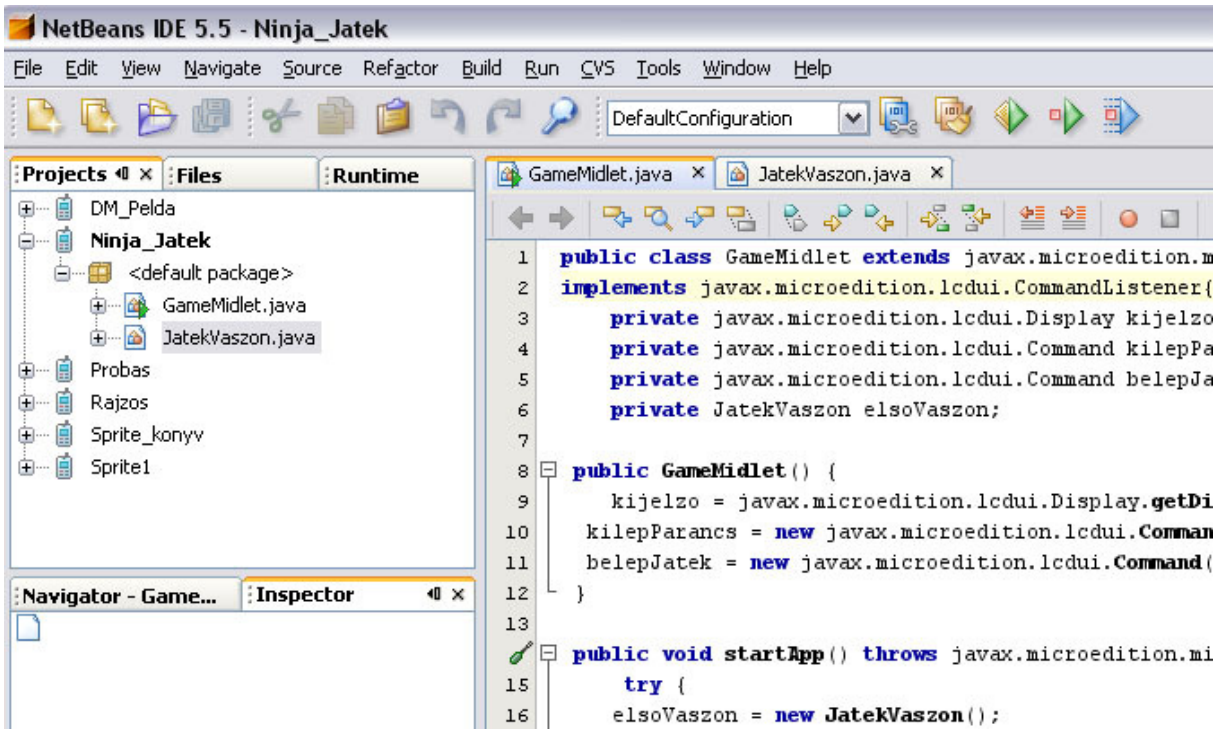
erőteljesen fejlődik, elképzelhető, hogy néhány éven belül a mai PC-s játékok mobilos változataival fogunk játszani a munkahelyi ebédszünetben.

A fejlesztéshez használt eszközök

NetBeans 5.5

A Sun által támogatott, ingyenes integrált fejlesztői környezet, megfelel akár kezdő akár profi programozóknak egyaránt. Logikus felépítésű, pluginek segítségével bővíthető.
A többi fejlesztői környezethez hasonlóan rendelkezik azok előnyével:

- forráskód kiegészítése, automatikus formázás, színkiemelés
- felbukkanó ablakok dokumentációs segítsége
- kód írása közbeni ellenőrzések, hibakövetés interaktív eszközökkel
- refactoring, varázslókkal történő átalakítás
- csoportmunka támogatása, CVS, Subversion
- tesztelés támogatása



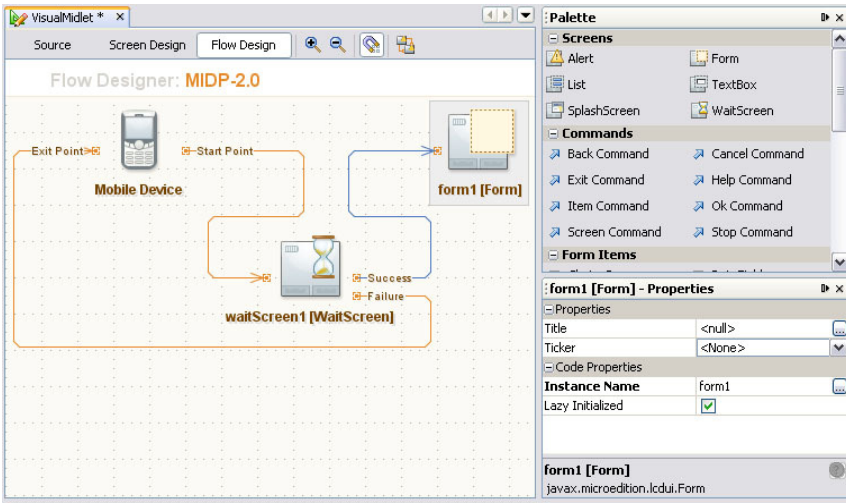
Letölthető innen: http://www.netbeans.info/downloads/index.php

NetBeans Mobility Pack 5.5

A Mobility Pack a NetBeans-hez elérhető csomag, mely segítséget nyújt a Java ME platformú alkalmazások fejlesztéséhez. A benne található emulátor megfelelően szimulálja egy valódi mobil eszköz működését. Tartalmaz egy Visual Mobile Desnigner nevű eszközt, segítségével egyszerűen csak rá kell húznunk a kiválasztott desing elemet a szerkesztett Form-unkra. A Flow Designer lehetővé teszi alkalmazásunk nézeteinek (Form-jainak) létrehozását és a közöttük lévő navigáció szerkesztésében is segítséget nyújt.

Az emulátor:

Flow Designer:



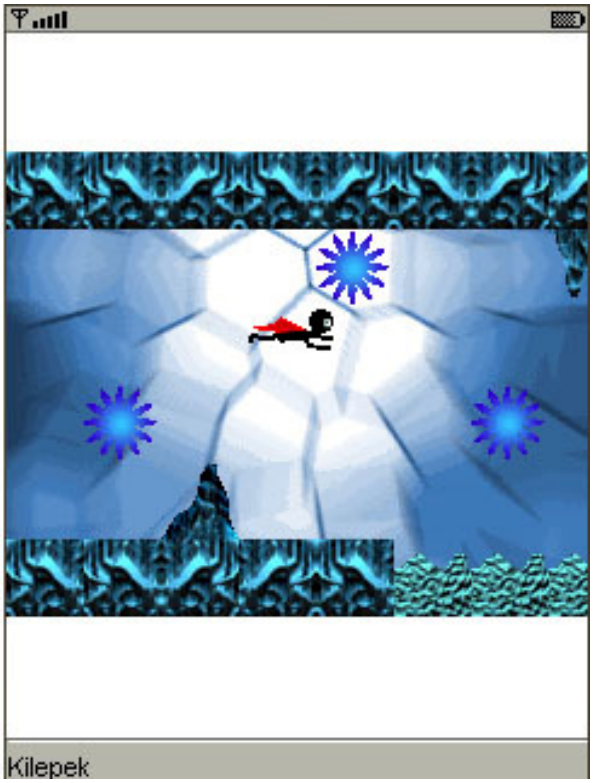
Lehetőség van méretezhető vektorgrafika (SVG - Scalable Vector Graphics) alkalmazására is. Ha esetleg nehéznek véljük az elindulást a Java ME egy bizonyos része felé, új projektet kezdve kiválaszthatjuk a beépített példák közül a számunkra hasznosat, ami egy jó kiindulási pontot adhat az adott témához.

Letöltése: http://www.netbeans.org/products/mobility/

A játék

Mivel egy egyszerű játékról van szó, ezért a játék célja is egyszerű: egy repülő figurával kell végighaladni a barlangszerű pályán, úgy hogy ne ütközzünk sem a barlang falának, sem a benne található forgó akadályoknak. Az előre való mozgás automatikusan történik, nekünk csak fel-, lemozgatásra, illetve az irány módosítására van lehetőségünk.

Néhány kép a játékról:



A játék felépítése

A játék egy MIDlet –ből (JatekMidlet) és egy osztályból (JatekVaszon) áll. Értelemszerűen először a MIDlet-et elemezzük ki, a forrásban sorba haladva, mi mire való. A játék teljesen a beépített emulátorra lett optimalizálva!

JatekMidlet elemzése

Mint azt a Java programoknál megszoktuk, az osztályt az *import* utasítással kezdjük, hogy a későbbiekben megkönnyítsük a kódolást.

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

A javax.microedition.midlet csomagra azért van szükség, mert ez tartalmazza a MIDlet osztályt, melyet minden MIDlet-nek ki kell terjesztenie. Ez az osztály rendelkezik három absztakt metódussal (startApp(), pauseApp(), destroyApp()), amelyeket implementálnia kell a MIDlet-nek. Az AMS-nek van erre szüksége, mert ezekkel kap értesítést a MIDlet az állapotváltozásokról. Mint azt később látjuk, a pauseApp() és destroyApp() törzsét üresen hagyjuk.
A javax.microedition.lcdui csomag az alkalmazásokban létrehozható felhasználói felületek készítésére és vezérlésére tartalmaz osztályokat.

```
public class JatekMidlet extends MIDlet implements CommandListener{
   private Display kijelzo;
   private Command kilepParancs;
   private Command belepJatek;
   private JatekVaszon jatekVaszon;
```

A JatekMidlet kiterjeszti a szükséges MIDlet osztályt, valamint implementálja javax.microedition.lcdui csomag CommandListener interfészét. Az interfész segítségével magas szintű eseményeket fogadhat az alkalmazásunk, ehhez a

void commandAction(Command c, Displayable d)

metódust kell implementálni, melyben lekezelhetjük az eseményeket. Ez minden esemény bekövetkezésekor lefut!

- Command c paraméter az aktuális esemény
- Displayable d azaz objektum, ahol az esemény bekövetkezett

Az adattagok:

- Display kijelzo – az adott eszköz képernyőjének és input eszközeinek a kezelését reprezentáló osztály
- Command kilepParancs, Command belepJatek – két parancs az üdvözlőkép számára
- JatekVaszon jatekVaszon – a játékot ténylegesen megvalósító osztály

Konstruktor:

```
public JatekMidlet() {
    kijelzo = Display.getDisplay(this);
    kilepParancs = new Command("Kilepek", Command.EXIT, 10);
    belepJatek = new Command("Jatek!", Command.ITEM, 10);
}
```

Az osztály konstruktora. ☺ Először a kijelzo adattagnak a Display.getDisplay(this) segítségével átadjuk a képernyőt reprezentáló objektumot, melynek paramétere az adott MIDlet. Majd inicializáljuk a Command típusú objektumokat (ezek tartalmazzák a magasszintű események tulajdonságait), a kijelzőn megjelenő névével, a hozzájuk rendelt parancstípussal és prioritással.

```
public void startApp() throws MIDletStateChangeException {
    try {
    jatekVaszon = new JatekVaszon();
    } catch(java.io.IOException e) {}
        jatekVaszon.addCommand(kilepParancs);
        jatekVaszon.addCommand(belepJatek);
        jatekVaszon.setCommandListener(this);
        kijelzo.setCurrent(jatekVaszon);
}
```

A kötelezően implementálandó metódusok egyike. Kötelező volt használni kivételkezelést, mert JatekVaszon-ban kiváltódhatnak kivételek a képek megnyitásakor. Hozzáadjuk a jatekVaszon-hoz a előbb inicializált parancsokat, beállítjuk a parancsfigyelőt (CommandListener), és MIDlet képernyőjét a jatekVaszon-ra állítjuk.

```
public void commandAction(Command command, Displayable displayable) {
   if (command == belepJatek) {
      jatekVaszon.removeCommand(belepJatek);
      jatekVaszon.betoltJatek();
   }

   if (command == kilepParancs) {
      kilep();
   }
 }
```

A CommandListener interfészhez szükséges commandAction() metódus implementációja. A két parancs használatakor végrehajtandó eseményeket állítjuk be.

```
        public void kilep() {
           jatekVaszon.fut = false;
           kijelzo.setCurrent(null);
           destroyApp(true);
           notifyDestroyed();
        }
```

A kilepParancs alkalmazásakor hívódik meg ez a metódus, beállítjuk a játék leállításához szükséges feltételt (jatekVaszon.fut = false;), mivel a kijelzőre nem lesz tovább szükségünk, ezért null-nak állítjuk be a megjelenítendő objektumot. A destroyApp(true) segítségével a MIDlet futasá mindenképpen befejeződik, és a legfoglalt erőforrások felszabadulnak. A

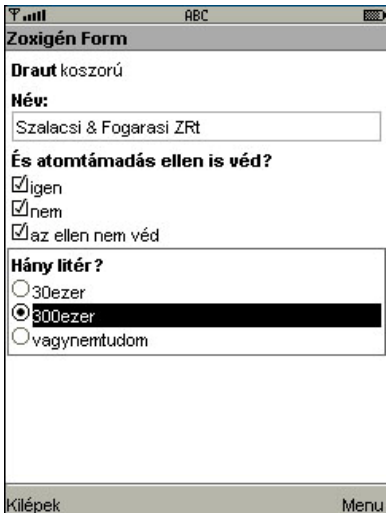
notifyDestroyed() jelzi az AMS-nek, hogy Destroyed állapotba került a MIDlet, ezzel zárjuk be azt.

```
public void pauseApp() {
 }

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

A MIDlet kiterjesztéséhez szükséges másik két metódus implementációja, mivel nincs rájuk szükségünk, de kötelező a szerepeltetésük, ezért üres törzzsel definiáljuk őket.

A játék indításakor egy egyszerű képernyőt látunk, csak egy Kilepek és egy Jatek! parancs jelenik meg. Persze lehetőségünk van teljes menürendszert felépíteni a Command-ok segítségével. A MIDP alkalmazások felhasználói felülete, kezelését tekintve nagyban különbözik az asztali gépekétől. Fontos, hogy egyértelműek, egyszerűen kezelhetőek legyenek azok számára is, akik nem jártasak a számítástechnikában, és akár egy kézzel is könnyen navigálhatóak legyenek. Ami a legnagyobb különbséget jelenti, hogy a telefonok nem rendelkeznek mutató eszközzel, mint a számítógép egérrel, vagy a PDA ceruzával. A felhasználói felület megtervezéséhez elég nagy szabadságot biztosít a Form osztály. Érdemes figyelni az eszközök megjelenítési korlátaira, ne legyen átláthatatlan a nézet! Az osztály segítségével interaktív űrlap felépítésű képernyő hozható létre, elhelyezhetőek rajta az Item osztály leszármazottjai: ChoiceGroup, CustomItem, DateField, Gauge, ImageItem, Spacer, StringItem, TextField. Lehetőség van teljesen új megjelenésű és kezelésű elem létrehozására, ehhez a CustomItem nevű absztrakt osztályt kell kiterjesztenünk.



Egy Form néhány elemmel

JatekVaszon elemzése, középpontban a Game API

Talán a legtöbb alkalmazás, amit a MIDP 1.0 óta Java ME platformra írtak, játék program. És mivel a játék a legnépszerűbb alkalmazás a Java-t támogató mobiltelefonok körében, ezért egy játékfejlesztést nagyban segítő API megjelenése várható volt. A MIDP 2.0-ban újdonságként megjelenik, a teljesen játék-specifikus Game API. A benne található néhány osztály segítségével szép megjelenésű, akár animált játék is készíthető.
A forrás elemzése közben, minden megjelenő eszköz funkcióját kifejtem.

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

A szükséges import-ok. Amit a *javax.microedition.lcdui* csomagból használunk:

- Image osztály – segítségével tölthetünk be képeket a MIDlet-ünkbe. Fontos, hogy csak PNG kiterjesztésű fájlokat használhatunk a platformon. A PNG-t (Portable Network Graphics) a W3C fejlesztette ki, azzal a céllal, hogy egy olyan tömörítési eljárást (deflation nevű algoritmus) alkalmazzanak benne, amely nem áll jogvédelem alatt (ellentétben a GIF formátummal, amely az LZW algoritmust alkalmazza). Nagy előnye, hogy lehetőség van átlátszó képek megjelenítésére!
- Graphics osztály – lehetővé teszi az egyszerű 2D-s rajzolást. A Canvas-okra tudunk vele rajzolni, ehhez szükséges implementálni a paint(Graphics g) metódusát, így a paraméterben megadott objektumra rajzolhatunk.
- Font osztály – reprezentálja a megjelenítendő fontokat és azok tulajdonságait. Néhány alapértelmezett beállítás segítségével állíthatjuk össze a nekünk tetsző betűtípust.

A *javax.microedition.lcdui.game* csomag használt osztályai:

- GameCanvas – ez az osztály adja az alapját a játék felületének. Az úgynevezett Double Buffer technikát valósítja meg, mely egyenletesen mozgó animációt eredményez. Lényege, hogy a *getGraphics()* által visszaadott grafikai objektum segítségével az off-screen buffer-be rajzolhatunk, így a bekövetkezett változás is itt történik meg, de ezek a változások csak a *flushGraphics()* metódus híváskor jelennek

meg a képernyőn. A megjelenítési feladatokon kívül az osztály a gomb események kezeléséért is felelős. Ezt az osztályt terjeszti ki a JatekVaszon.

- Layer – absztrakt osztály, a játék egy vizuális elemét reprezentálja. Tulajdonképpen egy réteg, működését tekintve hasonlít a grafikus programok rétegeihez. A programban mi ennek a leszármazottait használjuk: Sprite, TiledLayer.
- Sprite – alap vizuális elem. Segítségével hozhatunk létre animációt, ugyanis az animáció képkockáit egy kép tartalmazza.
- TiledLayer – szintén vizuális elem. Főként játékok hátterének készítéséhez, a pálya megrajzolásához használják. A Sprite-hoz hasonlóan ennek is egy darab kép adja az alapját, tulajdonképpen egy táblázat, ahol a cellákat a kép a részeivel töltjük fel, így meglehetősen nagy területet tölthetünk ki viszonylag kis képpel.
- LayerManager – több Layer-t összefogó és kezelő osztály. Nem kell külön minden Layer-t kirajzolni, elég ha a LayerManager rajzoló metódusát meghívjuk, ezzel miden olyan Layer kirajzolásra kerül, amely az adott LayerManager-hez hozzátartozik. Fontos szerepe van a Layer-ek kirajzolási sorrendjének, amit alapesetben a hozzáadás sorrendje határoz meg. Persze lehetőség van egy adott indexű helyre való beszúrásra.

Sprite

A játékok szempontjából az egyik legfontosabb eszköz, ezért is szánok rá egy külön kis fejezetet. A Sprite alapját egy kép adja, ez a kép tartalmazza az animáláshoz szükséges összes képkockát. A képkockák a következő elrendezésekben szerepelhetnek a képben:

1	2	3	4

1
2
3
4

1	2
3	4

A programban felhasznált Sprite az első elrendezési megoldással szerepel, ami egy 5 képmezőből álló kép:

Minden képkocka azonos méretű, automatikusan egy sorszám rendelődik hozzájuk A sorszámozás 0-tól indul és értelemszerűen képkockák száma-1–ig tart. Később ennek segítségével hivatkozhatunk az adott számú frame-re. Alapértelmezésben az alkalmazás sorszám szerinti sorrendben játssza le az animációt, de lehetőség van rá, hogy saját sorrendet definiáljunk a setFrameSequence(int[] sequence) metódus segítségével. Hogy mikor jelenjen meg a következő vagy előző képkocka, arról nekünk kell gondoskodnunk a nextFrame() vagy prevFrame() metódusokkal.

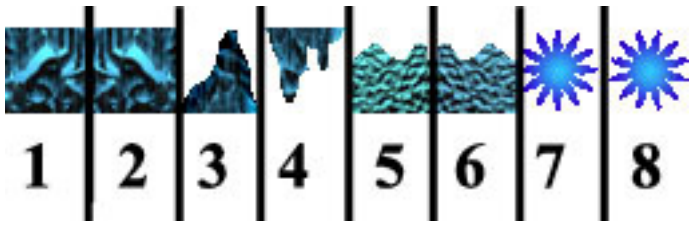
A setTransform() használatával végrehajthatunk a képkockákon különféle transzformációkat, tükrözhetjük, forgathatjuk, tükrözve forgatjuk, illetve visszaállíthatjuk eredeti állapotába a Sprite-unkat. Ezek alapját a referencia pixel képezi, melyhez viszonyítva a transzformáció végrehajtódik. Alapértelmezésben ez a frame (0,0) pontján, azaz a bal felső sarokban helyezkedik el, de módosítható a defineReferencePixel(int x , int y) segítségével a kívánt pozícióra.

A legnagyobb könnyítést a programozók számára talán az ütközések detektálása jelenti. Nem kell a képekre egyenként ütközést ellenőrző algoritmust írni, nincs szükség bonyolult koordinátákkal való számításokra! Helyettük egyszerűen csak az adott Sprite collidesWith() metódusát kell meghívni, mely true értékkel tér vissza, ha az első paraméterében megadott másik Sprite-tal (collidesWith(Sprite s, boolean pixelLevel)), TiledLayer-rel (collidesWith(TiledLayer t, boolean pixelLevel)) , képpel (collidesWith(Image image, int x, int y, boolean pixelLevel)) ütközés történt. Tehát az első paraméter tartalmazza azt az

objektumot, amellyel való ütközést ellenőrizni akarjuk. Az utolsó, logikai értékű argumentum az teszi lehetővé, hogy true érték esetén csak azokra a pixelekre történjen a detektálás, amelyek nem átlátszóak, azaz amit nem látunk, az nem is ütközik. Kép esetében meg kell adni még a kép bal felső sarkának a koordinátáit.

TiledLayer

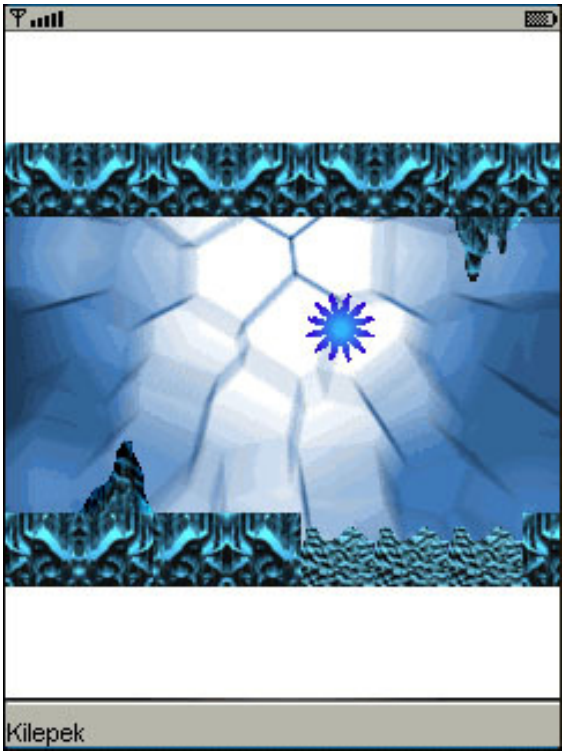
A játékok megrajzolásának másik fő komponense. Legtöbbször a háttér és a pálya elkészítéséhez használják. Tulajdonképpen egy táblázat, cellák rácsa, amiket egy kép részeivel töltünk fel. Hasonlóan a Spite-hoz, itt is a kép azonos méretű egységekből áll, csak a sorszámozás nem nullától, hanem egytől indul. A táblázat megadásánál a nulla értékű cellák üresen maradnak. A kép kockáinak elrendezése ugyanúgy történhet mint a Sprite-ok esetében.



Az itt látható képből egyszerűen fel lehet építeni a játék környezetét.
Például egy pályát reprezentáló táblázat:

2	1	2	1	2	1	1	2
0	0	0	0	0	0	4	0
0	0	0	0	-2	0	0	0
0	0	0	0	0	0	0	0
0	3	0	0	0	0	0	0
2	1	1	2	-1	-1	-1	2

És a táblázat alapján felépülő kép (a levegő háttér nélkül):



A táblázatban feltűnhet, hogy tartalmaz negatív számokat is. Lehetőség van a kép részeinek animálására. Ehhez a createAnimatedTitle(int staticTileIndex) metódust kell használni, ami egy negatív indexet ad vissza. Generál egy virtuális sorszámot, amit a paraméterben megadott képkocka sorszámához rendel. Ezt a hozzárendelést a setAnimatedTile(int animatedTileIndex, int staticTileIndex) metódussal tudjuk megváltoztatni, ami az összes azonos indexű elem változását eredményezi. A programban ezt a megoldást a víz és a forgó akadályok animálásánál alkalmazom. Sprite-oknál található ütközésdetektálás meghívható TiledLayer-re is, gondoljunk csak bele mennyi számítás lekódolására lenne szükség, ha ez nem lenne elérhető beépített funkcióként!

A játékkészítés alapját jelenti a Game API-n át történő programozás. A fejlesztők rendelkezésére állnak olyan osztályok, és bennük megtalálható metódusok, melyek segítségével rendkívül egyszerűen, viszonylag rövid idő alatt képesek piacképes programok előállítására.

A kis kitérő után nézzük tovább a JatekVaszon osztályunkat:

```
public class JatekVaszon extends GameCanvas implements Runnable {
```

Az osztály a játék alapját jelentő GameCanvas osztályt terjeszti ki és implementálja Runnable interfészt. A kiterjesztés különösebben nem szorul magyarázatra, ez adja az egész alapját. A Runnable interfészt azért használjuk, mert szálkezeléssel egyszerűen megvalósítható a játék folyamata, így mindent könnyedén csak a run() metódusban kell megvalósítanunk, és az automatikusan elvégzi majd a szükséges eseménykezelést, rajzolást.

```
Sprite ninjaSprite;
Sprite levegoSprite;
Image ninja_bej;
Image ninjaImg;
Image levegoImg;
Image hatterKep;
LayerManager lm;
TiledLayer barlang_TL;
int[] repulesSequence = {0, 1};
int[] halalSequence = {2, 3, 4};
int animatedIndex, animatedIndex_forgo;
```

Az osztály adattagjai: a Sprite-ok, TiledLayer, a hozzájuk szükséges képek (Image), a kezelésükhöz a LayerManager, a Sprite mozgásszekvenciái (repulesSequence, halalSequence), az animált részek indexei (animatedIndex, animatedIndex_forgo), és a futás feltétele (fut).

```
public JatekVaszon() throws java.io.IOException{
  super(true);
  fut = true;
   try {
      ninjaImg = Image.createImage("/ninja_rep.png");
      levegoImg = Image.createImage("/levego.png");
      ninja_bej = Image.createImage("/ninja_b.png");
      hatterKep = Image.createImage("/tiles_barlang.png");
   } catch(java.io.IOException e) {}

  lm = new LayerManager();
  lm.setViewWindow(0, getHeight() - 192, getWidth(), 192);

 ninjaSprite = new Sprite(ninjaImg, 40, 40);
 ninjaSprite.setFrameSequence( repulesSequence );
 ninjaSprite.defineReferencePixel(20, 20);
 ninjaSprite.setRefPixelPosition(getWidth()/2, getHeight()-32-20);
 ninjaSprite.defineCollisionRectangle(0, 10, 36, 17);

 levegoSprite = new Sprite(levegoImg);
 levegoSprite.defineReferencePixel(levegoImg.getWidth()/2, levegoImg.getHeight()/2);
 levegoSprite.setRefPixelPosition(getWidth()/2, getHeight()/2);
 lm.append(levegoSprite);
 createHatter();
 udvozloKep();
```

Konstruktor a szükséges kivételkezeléssel. Felhasznált képek betöltése, LayerManager inicializálása, és a megjelenítési ablak beállítása (csak az ezen belül megjelenő objektumok lesznek láthatóak). Az animált, repülő figurához tartozó Sprite létrehozását, a mozgó képkockák sorrendjének megadását, az említett referencia pixel pozícionálását, és az ütközési téglalap definiálását végezzük el. A következő Sprite csak egy statikus képet tartalmaz, ez adja a játék hátterét (levegő). Ezt hozzáfűzzük a LayerManagar-ünkhöz append() metódussal, így van már egy rétegünk.

```
void udvozloKep() {
   Graphics g = getGraphics();
   g.drawImage(ninja_bej, getWidth()/2, getHeight()/2, Graphics.VCENTER |
Graphics.HCENTER);
}
```

A metódus meghívásának eredményét pillantjuk meg először a képernyőn. A getGraphics() által visszaadott grafikai objektumra kirajzoljuk a képernyő közepére pozícionált képet.

```
void betoltJatek() {
   new Thread(this).start();
}
```

Az üdvölőkép Jatek! parancsához tartozó gomb lenyomása esetén hívódik meg, létrehoz egy szálat, majd elindítja azt, a Java Virtuális Gép meghívja a szálhoz tartozó run() módszert.

```
private void createHatter() {

    int[][] map = {{2, 1, 2, 1, 2, 1, 1, 2, …},
                   {0, 0, 0, 0, 0, 0, 4, 0, …},
                   {0, 0, 0, 0, 0, 0, 0, 0, …},
                   {0, 0, 0, 0, 0, 0, 0, 0, …},
                   {0, 3, 0, 0, 0, 0, 0, 0, …},
                   {2, 1, 1, 2, -1, -1, -1, -1, …}};

   barlang_TL = new TiledLayer(48, 6, hatterKep, 32, 32);
   barlang_TL.setPosition(0, getHeight() - 192);

   animatedIndex = barlang_TL.createAnimatedTile(5);
   animatedIndex_forgo = barlang_TL.createAnimatedTile(7);
```

```
    for (int i = 0; i < barlang_TL.getRows(); i++) {
        for (int j = 0; j < barlang_TL.getColumns(); j++)
            barlang_TL.setCell(j, i, map[i][j]);
    }
    lm.insert(barlang_TL, 0);
  }
```

A metódus segítségével építjük fel a pályát (hátteret). Megadjuk a TiledLayer táblázatának felépítését egy kétdimenziós tömb segítségével. Inicializáljuk a szükséges adatokkal, majd beállítjuk a pozícióját, és lekérdezzük az animált részek indexét. Az egymásba ágyazott for-ciklusokkal töltjük fel a TiledLayer celláit. Majd ezt a réteget beszúrjuk a LayerManager 0. helyére. Most már két réteget tartalmaz, hátul a levegőt, előtte pedig magát a játék pályáját.

Most nézzük a játék színterének felépítése után a játék lényegét adó run() metódust:

```
public void run() {
    Graphics g = getGraphics();

    int ir = -4, animValt = 1;
    boolean vesztett = false;

    Font f =
```

A Runnable interfészhez szükséges metódus, a szál indításakor hívódik meg. Először lekérjük a rajzolásra szánt grafikus objektumot, majd a játék állapotához szükséges változókat állítjuk be: ir = -4 – melyik irányba haladjon majd a figura, animValt = 1 – a TiledLayer animált részeinek váltásához szükséges, vesztett = boolean – a neve magaért beszél, vesztett-e a játékos. Az f Font típusú objektumba lekérünk a fenti beállításokkal egy betűtípust, majd ezt állítjuk be a grafikus objektum írási betűtípusának. A LayerManager-hez egy újabb réteget fűzünk, ez a figura Sprite-jából áll, és mint a sorszámán is látszik, ez lesz a legfelső.

```
while(fut) {
```

Indítunk egy ciklust, melynek feltétele a fut változó logikailag igaz értéke. Ebben a ciklusban valósul meg a játék.

```
    g.setColor(0xffffff);
    g.fillRect(0, 0, getWidth(), getHeight());
    g.setColor(0x000000);
```

Hogy mozgóképként érzékeljük majd a pálya megrajzolását, minden ciklusismétlődéskor le kell törölnünk a képernyőt.

```
int billentyu = getKeyStates();

    if( ((billentyu & GameCanvas.LEFT_PRESSED) != 0) ) {
        ir = 4;
        ninjaSprite.setTransform(Sprite.TRANS_MIRROR);
    }
    else if( ((billentyu & RIGHT_PRESSED) != 0) ) {
        ir = -4;
        ninjaSprite.setTransform(Sprite.TRANS_NONE);
    }
    else if( ((billentyu & GameCanvas.UP_PRESSED) != 0) ) {
 ninjaSprite.setRefPixelPosition(ninjaSprite.getRefPixelX(), ninjaSprite.getRefPixelY()-5);
    }
    else if( ((billentyu & GameCanvas.DOWN_PRESSED) != 0) ) {
 ninjaSprite.setRefPixelPosition(ninjaSprite.getRefPixelX(),ninjaSprite.getRefPixelY()+5);
    }
```

A következő feladatunk a gombesemények kezelése. Egy int típusú változóba bekérjük a lenyomott gomb kódját, majd az elágazás feltételében látható módon ellenőrizzük, melyik gomb került lenyomásra. Számunkra csak a négyirányú mozgást reprezentáló gombesemények a fontosak: ha balra vagy jobbra akarunk tovább menni, szükség lehet a figura tükrözéséhez, hogy ne háttal a menetiránynak repüljön. A fel, le mozgáshoz a figura Sprite-jának a referencia pixelét kell a megadott irányba pozícionálni.

```
ninjaSprite.nextFrame();
```

A Sprite animálást biztosítjuk, átléptetjük a következő képkockára, a ciklus futása adja az animálás folyamatosságát.

```
if( ((barlang_TL.getX()+ir <= 0) && (ir > 0)) || ( (ir<0) &&
(barlang_TL.getX()+barlang_TL.getColumns()*barlang_TL.getCellHeight()-ir) >=
getWidth()) ) {
      barlang_TL.move(ir, 0);
   }
```

A figura mozgatása csak fel-le történik, vízszintes irányú elmozdulást nem végezünk vele. Nem a figura Sprite-ja mozog előre, hanem a pályát megvalósító réteget mozgatjuk, emiatt tűnik úgy, mintha a kis alak repülne. Ez a feltétel azért szükséges, hogy a pálya ne szaladjon ki a képernyőből egyik irányba sem, ugyanis csak akkor végezzük el a mozgatást, ha az még a pálya megfelelő láthatóságát eredményezi a készüléken.

```
if(ninjaSprite.collidesWith(barlang_TL, false)) {
        ninjaSprite.setFrameSequence( halalSequence );
      for(int i=0 ; i < 3 ; i++ ) {
         ninjaSprite.setFrame(i);
         lm.paint(g, 0, (getHeight()-192)/2);
         flushGraphics();
         try {
            Thread.sleep(100);
         } catch(InterruptedException e) {}
      }
     vesztett = true;
      fut = false;
}
```

Mozgatás után ellenőrizzük, hogy a figuránk nekiütközött-e a TiledLayernek, azaz a pályának. A második paraméter false értéke miatt az ütközés ellenőrzését csak az ütközési téglalapra hajtjuk végre. Először próbáltam az átlátszó pixelek figyelmen kívül hagyását, de ez egy kivételt és a játék leállását eredményezte. Sajnos nem sikerült kideritenem mi váltotta ezt ki, sejtésem szerint a mozgó részekkel történő ütközést nem tudta rendesen kezelni, mivel itt folyamatosan változik az átlátszó részek helyzete az animálás miatt.
Ütközés esetén átállítjuk a képkockák sorrendjét. Az új sorrend a figura halálát animálja. Ennek a lejátszását valósítja meg a következő for-ciklus: beállítjuk a megfelelő képkockát, a LayerManager paint() metódusával egyszerűen megrajzoljuk az összes, hozzá kapcsolódó réteget, a flushGraphics()-val pedig a kirajzoljuk az off-screen buffert, majd hogy ne legyen a lejátszás menete érzékelhetetlenül gyors, a szálat „elaltatjuk” 100 millisecundum ideig. Ütközés miatt a játék állapotát tükröző logikai változókat beállítjuk.

```
if( animValt == 1) {
      barlang_TL.setAnimatedTile(animatedIndex, 6);
      barlang_TL.setAnimatedTile(animatedIndex_forgo, 8);
    } else {
      barlang_TL.setAnimatedTile(animatedIndex, 5);
      barlang_TL.setAnimatedTile(animatedIndex_forgo, 7);
 }
animValt = -animValt;
```

A TiledLayer mozgó részeinek animálását végezzük el. Az animValt változó használatával váltogatjuk az animált képrészeket. Majd, hogy a következő ciklusfutásban a másik képkocka jelenjen meg, negáljuk a segédváltozót. Persze tetszés szerint akár logikai változóval is megoldható a két állapot váltogatása.

```
lm.paint(g, 0, (getHeight()-192)/2);
```

Egyetlen metódushívással megrajzoljuk a teljes játékot.

```
if(
((barlang_TL.getX()+barlang_TL.getColumns()*barlang_TL.getCellHeight()-ir)
<= getWidth()) && !vesztett ) {
        g.setColor(0xff0000);
        g.drawString("NYERTEL!", getWidth()/2, getHeight()/2, Graphics.TOP |
Graphics.HCENTER);
      fut = false;
```

Itt ellenőrizzük, hogy a pálya végére értünk-e már, tehát nyertünk-e. Ha igen, beállítunk egy pirost színt a rajzolás színének, majd kiírjuk a képernyő közepére a NYERTEL! feliratot. Mivel nyertünk, beállítjuk a játék befejezésének megfelelően a fut változót.

```
if(vesztett) {
        g.setColor(0xff0000);
        g.drawString("VESZTETTEL!", getWidth()/2, getHeight()/2,
Graphics.TOP | Graphics.HCENTER);
  }

    flushGraphics();
```

Vesztés esetén szintén piros színnel kiírjuk a VESZTETTEL! szöveget a kijelzőre. Hogy az off-screen bufferben megtörtént változások megjelenjenek, meghívjuk a flushGraphics() metódust.

```
try {
    Thread.sleep(100);
   } catch(InterruptedException e) {}
  }
 }
}
```

Hogy a játék menete ne csak egy gyorsan falnak ütközés legyen, késleltetjük a szál futását, ügyelve az esetlegesen kiváltódó kivételre.

Ezzel elérkeztünk a program elemzésének a végére. Látható, hogy nem nehéz játékprogramot írni erre a platformra, hiszen rengeteg segítséget nyújtanak a beépített lehetőségek. Aki már ismeri a Java nyelvet, annak a platform-specifikus elemek elsajátítása nem okoz gondot.

Összegzés

A mobiltelefon talán napjaink legkedveltebb és legtöbbet használt kommunikációs eszköze. A készülékeken elérhető Java ME támogatás igen nagy szabadságot biztosít az eszköz-specifikus programok fejlesztéséhez. Ennek a szabadságnak a határait csak a készülékek fizikai korlátai szabják meg.

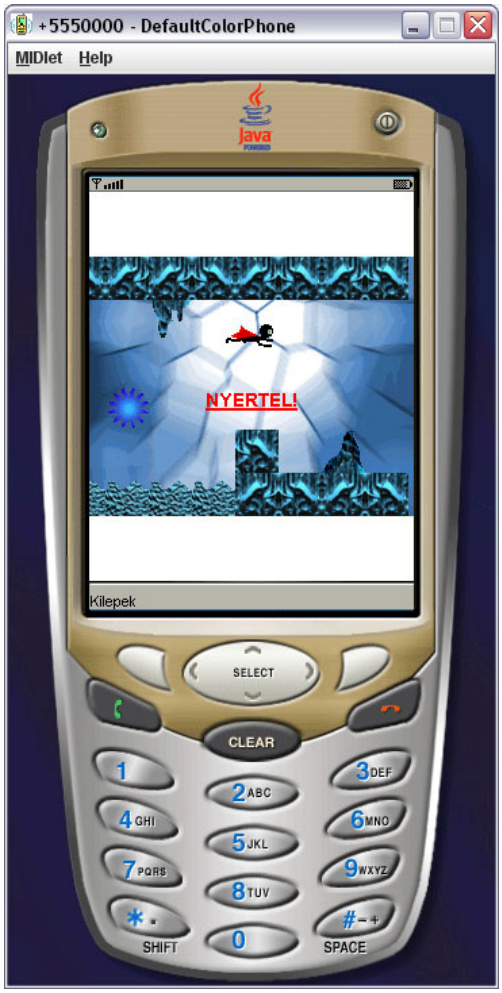
Mivel a legelterjedtebb, mobiltelefonokon elérhető Java alkalmazások a játékok, ezért állítottam egy egyszerű játékot a szakdolgozat témájának középpontjába. Remélem megfelelően jó kiindulási alapul szolgál majd azoknak, akik érdeklődnek a téma iránt. Talán könnyebb megkedvelni úgy egy programozási platformot, hogy nem unalmas, már megszokott példákon keresztül próbáljuk elsajátítani azt, hanem egy játékot veszünk alapul, aminek mind a grafikai, mind pedig a szórakoztató jellege leköti az érdeklődő figyelmét, és nem csak tanul, hanem élvezettel veti bele magát a játékkészítés, ezáltal a programozás rejtelmeibe.

Persze nem minden csak a játékról szól. De ha már eljutottunk egy bizonyos szintre, akkor már bátran belevághatunk a Java ME platform nyújtotta további lehetőségek megismerésébe.

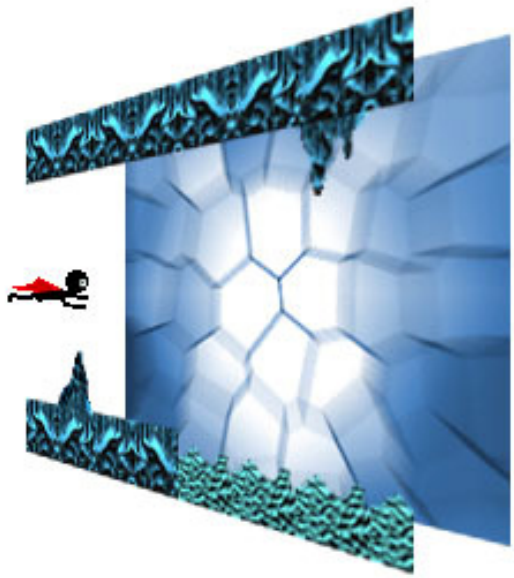
Sok minden múlik a kezdésen, és ha már a dolgok elején megkedveljük azt, amivel foglalkozunk, akkor sokkal könnyebb elsajátítani az új ismereteket.

Függelék

Néhány kép magáról a játékról az emulátoron történő tesztelés közben:



A játék belépő képernyője, indulása, Megnyerése, elvesztése



Egy kis segítség rétegek elképzeléséhez ☺

Irodalomjegyzék

- Angster Erzsébet: Objektumorientált tervezés és programozás, 2003, Akadémiai Nyomda
- Vartan Piroumian: Wireless J2ME Platform Programming, 2002, Prentice Hall
- http://www.ittk.hu/web/docs/ITTK_MITJ_2006.pdf
- http://www.ittk.hu/web/docs/mwdr2004_ittk.pdf
- http://java.sun.com/javame/technology/index.jsp
- http://java.sun.com/products/cdc/overview.html
- http://java.sun.com/products/cldc/overview.html
- http://java.sun.com/products/midp/whatsnew.html
- http://pallergabor.uw.hu/hu/java-app/
- http://java.sun.com/javame/reference/apis/jsr118/