

Java 2 ME

Alkalmazásfejlesztés mobiltelefonokra



Pekk Roland



© 2006

Bevezetés

A vezeték nélküli kommunikáció igencsak elterjedt és népszerű az egész világon, hiszen mindenki a nap 24 órájában elérhető akar lenni, és ő is el akar érní mindenkit (kivéve amikor nem). A mobiltelefon tehát senkinek sem újdonság manapság, de egy mezei mobillal senki sem elégszik meg. Hatalmas pénzek vannak a mobilpiacban, így hát minden gyártó egyre jobb, és „okosabb” készülékeket dob a piacra, amelyek színes kijelzővel, kamerával vannak felszerelve. Ki hinné, hogy az ember szeret a mobiljával is játszani? Tekintve a számítógépes játékipiac méreteit, egyáltalán nem csoda.

Nosza, fejlesszünk mobiltelefonra! A kérdés már csak az, hogy hogyan, és mégis mivel? A Java programnyelv már régóta létezik a számítógépes világban, egyre elterjedtebb, és mivel platformfüggetlen, logikus választásnak tűnik.

Java Virtual Machine

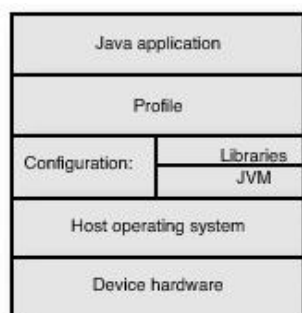
A JVM, vagyis a Java virtuális gép egy működtető motor minden Java program mögött (vagy applet, servlet, stb.). A JVM lefordítja az osztály (class) fileokat az aktuális platform gépi kódjaivá, amin a JVM fut. Garantálja a biztonságot, kezeli a memóriát (lefoglal és felszabadít), vezérli a végrehajtási szálakat.

Ez mind szép és jó lenne, de azért egy mobiltelefon teljesítménye mégsem hasonlítható össze egy PC-ével. A JVM túl sok erőforrást igényel ahhoz, hogy egy mobiltelefonon futtatható legyen. A Javában egy konfiguráció van definiálva az azonos tulajdonságú eszközöknek. A Java 2 ME két ilyen definiál, a CDC-t és a CLDC-t.

Konfigurációk (CDC, CLDC)

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

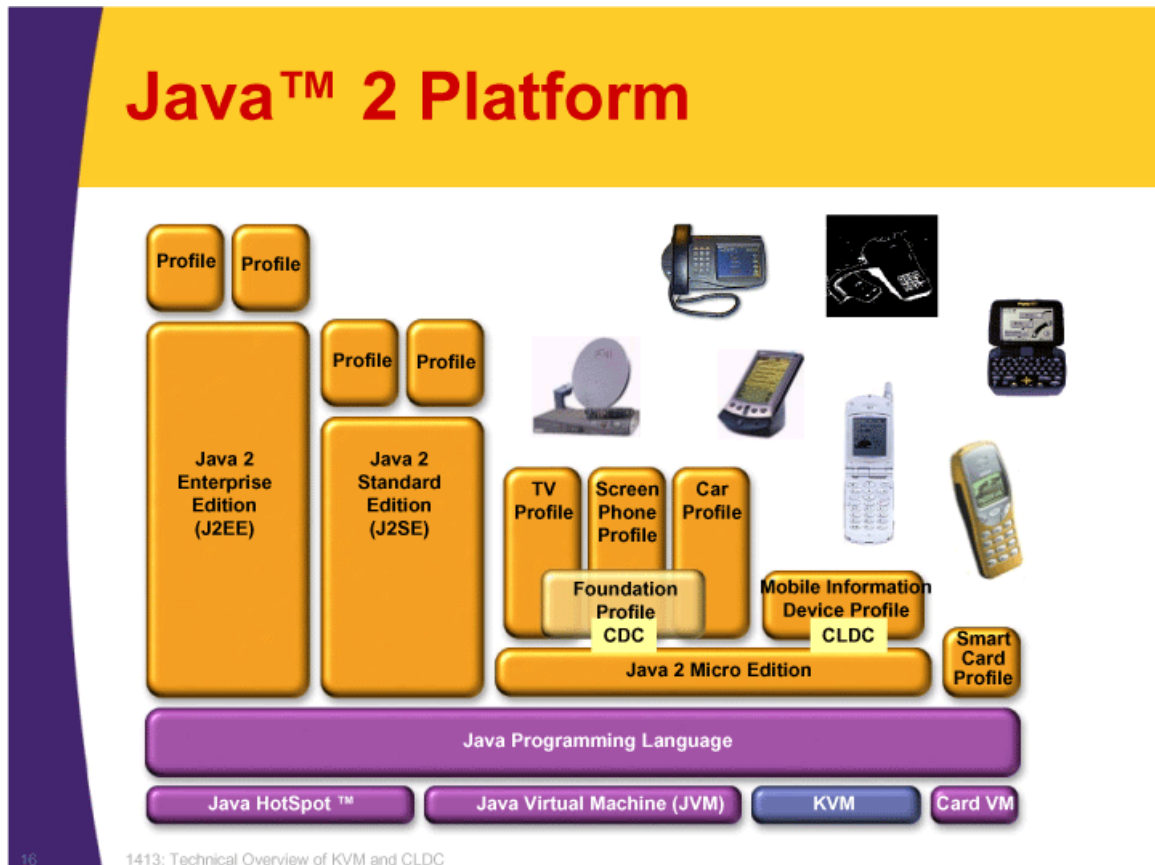
A CLDC-nek a Sun kifejlesztette a virtuális gép referencia implementációját, ami a K Virtual Machine nevet kapta, vagy simán KVM. Tehát a CLDC tulajdonképpen a KVM és az alapvető API-k összessége. Mi most természetesen csak a CLDC-t használó eszközökkel fogunk foglalkozni.



1. ábra Jól látszik a konfiguráció összetétele

K Virtual Machine (KVM)

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



2. ábra

CLDC 1.0 (JSR-30)

Connected, Limited Device Configuration

Célok

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

Hatásköre

Java nyelv és VM sajátosságok

Java könyvtárak magja: java.lang.*, java.util.*

Input/Output

Hálózat

Biztonság

Többnyelvűvé tétel

Nem biztosítja a következő funkciókat, ezeknek a területeknek a kezelése a profilok feladata:

- Alkalmazások életciklus modellje (telepítés, betöltés, törlés)
- Kapcsolattartás a felhasználóval
- Eseménykezelés

Biztonság

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

Kihagyott tulajdonságok:

- Java Native Interface (JNI)
- lebegőpontos ábrázolás
- felhasználó-definiált JAVA osztálybetöltő (class loader)
- Reflection támogatás (komponensek felderítése)
- Szálcsoportok és démon szálak
- Finalization (nem tartalmazza az Object.finalize() metódust)
- Gyenge hivatkozások
- Korlátozott hibakezelés (A java.lang.Error legtöbb leszármazottja nem támogatott)

CLDC 1.0 API

- java.io (része a J2SE-nek)
- java.lang (része a J2SE-nek)
- java.util (része a J2SE-nek)
- Hálózatkezelés (javax.microedition.io)

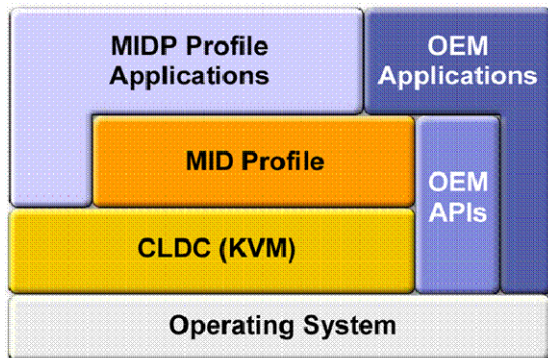
CLDC 1.1 (JSR-139)

- legalább 192 kB teljes memória a Java platformnak
 - legalább 160 kilobyte "nem felejtő memória" a virtuális gépnek és a CLDC könyvtáraknak.
 - legalább 32 kilobyte „felejtő memória” alkalmazásfuttatáshoz (pl. objektum heap)
- 16 vagy 32-bites processzor
- alacsony energiafelvétel, gyakran elemmel működik

- Csatlakoztathatóság valamilyen hálózathoz, gyakran vezeték nélkülihez, szakaszos kapcsolat korlátozott sávszélességgel

MIDP – Mobile Information Device Profile

Egy J2ME MID profil definiálja a kiegészítő API-k halmazát és jellemzőket egy speciális készülék kategória (egy családba tartozó készülékek) számára. Egy készülék akár több profilt is támogathat. Az MIDP és a CLDC együtt alkotja a Java 2 ME alkalmazás futási környezetét.



3. ábra

A 3. ábrán láthatjuk, hogy nemcsak MIDP alkalmazásokat készíthetünk, hanem ún. OEM alkalmazásokat is. Ezek annyiban térnek el a többitől, hogy olyan eszközspecifikus API-kat is használnak, amiket a gyártók az adott eszközhöz adtak ki, így ezek az alkalmazások nem vihetők át másik eszközre. Célszerű tehát MIDP alkalmazásokat írni, amiket több típuson is lehet futtatni.

MIDP 1.0 (JSR-37)

Minimum hardverkövetelmények:

- Kijelző:
 - Képernyőméret: 96x54
 - színmélység: 1-bit
 - torzítási arány: 1:1
- Input:
 - egykezes billentyűzet (ITU-T); pl. szokványos mobiltelefon készülék
 - kétkézes billentyűzet (QWERTY); pl. PalmTop
 - érintőképernyő
- Memória: memória követelmények cited here are for MIDP components only. CLDC and other system software memory requirements are beyond the scope of this specification and therefore not included.
 - 128 kilobyte non-volatile memória (tartalmát megőrzi a készülék) a MIDP komponenseknek
 - 8 kilobyte non-volatile memória az alkalmazás által létrehozott perzisztens adatoknak
 - 32 kilobyte volatile memória a Java futtatásának (pl. Java heap)
- Hálózat:
 - kétirányú, vezeték nélküli, esetleg időszakos, korlátozott sávszélességgel

MIDP 1.0 API

- Alkalmazás életciklusa (javax.microedition.**midlet**)
- Felhasználói felület (javax.microedition.**lcdui**)
- Perzisztens tárolás (javax.microedition.**rms**)
- Hálózatkezelés (javax.microedition.**io**)

MIDP 2.0 (JSR-118)

- Game API (javax.microedition.lcdui.**game**)
 - GameCanvas (fullscreen lehetőség)
 - RGB képek
 - TiledLayer, Layer, Sprite (transzformációkkal és ütközés detektálással)
- Media API (javax.microedition.**media**)
- UI
- Háttérvilágítás, vibrálás
 - Formok
- Biztonság
- Hálózatkezelés

Secure Networking https

Form Enhancements Form, Item, Spacer, CustomItem

Rövidítések

API – Application Programming Interface

CLDC - Connected Limited Device Configuration

KVM - K Virtual Machine

MIDP – Mobile Information Device Profile

IDE - Integrated Development Environment: Integrált fejlesztői környezet

SDK – Software Development Kit

A fejlesztéshez használt alkalmazások

Szerencsére a használni kívánt alkalmazások mindegyike ingyenes, legálisan fejleszthetünk.

- NetBeans IDE 5.0 (58 Mb) (<http://www.netbeans.org>)
- NetBeans Mobility Pack 5.0 (22.4 Mb) (<http://www.netbeans.org>, <http://java.sun.com>)
 - emulátor
 - telepítés, beállítás a későbbiekben
- Java 2 SE JDK 5.0 Update 8 (49.5 Mb) (<http://java.sun.com/>)
- Egyéb telefon SDK (pl. Siemens Mobility Toolkit (SMTK), Sony Ericsson SDK for Java ME)
 - emulátor
 - OEM API-k

MIDLet, JAD, JAR

Telepítés

Először is a Java 2 SDKt kell feltelepítenünk, ha még nincs fenn a gépen. A telepítés pár Next gomb-nyomogatásból, és a telepíteni kívánt összetevők kiválasztásából áll, nem okozhat problémát senkinek sem.

Ezután jöhet a NetBeans, aminek a telepítése szintén egyszerű, ha minden jól megy, a Java 2 SDK könyvtárát automatikusan megtalálja, ha mégsem, akkor adjuk meg neki kézzel.

Végül a NetBeans Mobility Pack-et telepítsük, ami keresni fogja a NetBeans telepítési könyvtárát, ha nem találja, adjuk meg kézzel.

Ha elindítottuk a NetBeanst, Tools | Java Platform Manages -> Add Platform, Java Micro Edition Platform Emulator, itt tudunk más emulátorokat is telepíteni.

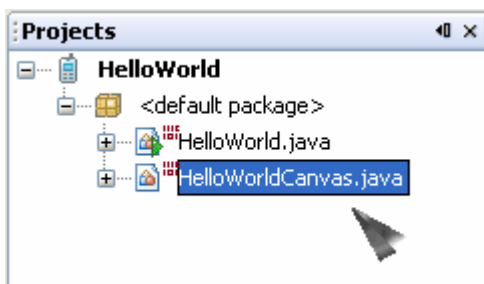
Hello World

File | New Project -> Categories: Mobile | Projects: Mobile Application

Project Name: A projektünk neve | Project Location: Hol tárolódik | A Project Folder automatikusan létrejön a Project locationön belül, neve megegyezik a Project Name-el. A Create Hello MIDlet ne legyen bepipálva, különben létrehoz nekünk 1 HelloWorld programot, formmal, mindennel együtt, de mi ezt magunknak szeretnénk megcsinálni, és nem formmal.

Ezután kiválaszthatjuk, hogy milyen emulátor-csomagot szeretnénk használni (ha többet is telepítettünk), milyen készülék legyen emulálva, beállíthatjuk a konfigurációt (CLDC) és a profilt (MIDP) is, ha többet támogat a kiválasztott készülék, ezután Finish. Ezzel létrehoztunk egy üres projektet, ami a bal oldali ablakban látszik is.

Először is szükségünk van egy MIDlet-re: File | New File -> Categories: MIDP | File Types: MIDlet. Adjuk meg a MIDlet Name-t: HelloWorld, MIDlet Class Name: HelloWorld. Szükségünk lesz még 1 "vászonra", amire kiírhatjuk a szövegünket: File | New File -> Categories: MIDP | File Types: MIDP Canvas. MIDP Class Name: HelloWorldCanvas. Ezután ez az osztályunk is megjelenik a projekt-listában a default package-en belül, de természetesen kedvünkre csomagokba (package) is szervezhetjük az osztályainkat.



4. ábra

Legyünk túl gyorsan a forráson, aztán szépen kielemezzetjük.

```

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

public class HelloWorld extends MIDlet {
    private Display kepernyo;
    private HelloWorldCanvas vaszon;

    public HelloWorld() {
        vaszon = new HelloWorldCanvas(this);
    }

    public void startApp() {
        if (kepernyo==null) {
            kepernyo=Display.getDisplay(this);
        }
        kepernyo.setCurrent(vaszon);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }
}

```

A javax.microedition.midlet csomag tartalmazza a MIDlet osztályt, melyet a mi HelloWorld osztályunk kiterjeszt. A MIDlet osztály három absztrakt metódusát (startApp(), pauseApp(), destroyApp()) a származtatott osztályunknak implementálnia kell. Definiáltunk egy Display típusú, és egy HelloWorldCanvas típusú objektumot. (a HelloWorldCanvas-t is mindjárt megírjuk) A HelloWorld konstruktorában létrehozuk a HelloWorldCanvas típusú vaszon-t, a HelloWorldCanvas konstruktorának átadjuk paraméterben a MIDletet.

A javax.microedition.lcdui csomagban a Display osztály található. A Display reprezentálja a kijelző és a beviteli eszközök vezérlőjét. Hozzárendeljük a kepernyo-höz a kijelzőt (ha még nem tettük meg) a getDisplay(this)-el, ezután a setCurrent-el megjelenítjük a vaszon-unkat, ami ugye HelloWorldCanvas típusú.

Lássuk tehát a HelloWorldCanvas-t:

```

import javax.microedition.lcdui.*;

public class HelloWorldCanvas extends Canvas implements CommandListener {
    private Command exitCommand=new Command("Exit", Command.EXIT, 1);
    private HelloWorld helloworld;

    public HelloWorldCanvas(HelloWorld midlet) {
        helloworld=midlet;
        addCommand(exitCommand);
        setCommandListener(this);
    }

    public void paint(Graphics g) {
        g.setColor( 0xffffffff );
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor( 0x000000 );
        g.drawString("Hello World", 0, 0,Graphics.TOP|Graphics.LEFT);
    }
}

```

```

    public void commandAction(Command command, Displayable displayable) {
        if (command.equals(exitCommand) ) {
            helloworld.notifyDestroyed();
        }
    }
}

```

A Canvas osztályból származtatjuk a HelloWorldCanvas-unkat, és implementáljuk a CommandListener-t, ami a javax.microedition.lcdui csomagban található. Definiáltunk egy exitCommand nevű Commandot, és egy HelloWorld típusú objektumot is, ebben tároljuk el a MIDletünk, amit paraméterben adtunk át, ez ahhoz kell, hogy be tudjuk majd zárni. A setCommandListener(this)-el állítjuk az aktuális Displayable-re a parancsfigyelést. Erről egyelőre annyit kell tudnunk, hogy a Canvas a Displayable-ből van származtatva (a Screen-el együtt). Az addCommand(exitCommand)-al hozzáadjuk a Displayable-nkhoz az elején definiált parancsunk, aminek Exit a felirata.

Implementálnunk kell a Canvas paint(Graphics g) metódusát. Ezt nekünk sosem kell meghívunk, az implementáció hívja meg. A paraméterben kapott Graphics objektumot használhatjuk rajzolásra, beállítjuk a színt a setColor-al (RGB szín hexában), letöröljük a képernyőt a FillRect-el (négyzetrajzolással), a getWidth() és getHeight() szintén Displayable metódusok, az elérhető terület szélességét, és magasságát adják meg pixelben. Az értékek függhetnek a címkétől, tickertől, és a commandok jelenlététől. Beállítjuk a színt, és a drawString-el kiiratjuk a Hello World szöveget. Mivel implementáljuk a CommandListener-t, meg kell írunk a commandAction metódust. A paraméterben megkapott paranccsot megvizsgálva eldönthetjük, hogy mit kell végrehajtanunk, igaz most csak az exitCommandunk van. Értesítjük a midletünk, hogy záruljon be.

Na, ha már nagyjából értjük a Hello Worldöt, akkor térjünk ki részletesebben a használt osztályok metódusaira.

MIDlet

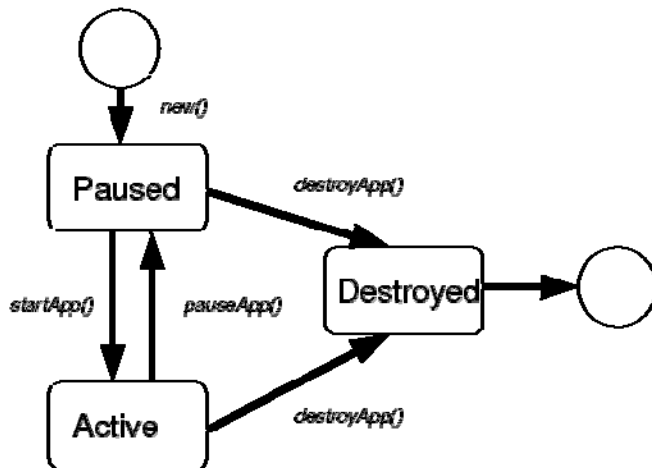
javax.microedition.midlet

A MIDlet egy MID Profilú program.

MIDletcsomag összeállításának módja

- Java Archive (JAR)
- Application Descriptor (JAD)
 - Alkalmazás neve, verziója, gyártó, JAR URL, JAR méret

A programunk életciklusa:



5. ábra

- Az Application Management Software (AMS) létrehozza a MIDletünk (`new()`).
- Az AMS aktiválja a MIDletünk, meghívódik a `startApp()`. A MIDlet Active állapotba kerül.
- Az AMS felfüggeszti a MIDlet futását, meghívódik a `pauseApp()`. A MIDlet Paused állapotba kerül.
- Az AMS bezárja a MIDletet, meghívódik a `destroyApp()`. A MIDlet Destroyed állapotba kerül, befejeződik a futása.

protected abstract void destroyApp(boolean unconditional)

a MIDlet bezárásakor hívódik meg, ezután a MIDlet Destroyed állapotba kerül. , ha a paraméter true, akkor itt felszabadíthatjuk a lefoglalt erőforrásokat

protected abstract void pauseApp()

Ha a MIDletünk Paused állapotba kerül, akkor meghívódik a metódus.

protected abstract void startApp()

Ha a MIDletünk Active állapotba kerül, akkor meghívódik a metódus. Mivel a `startApp()` mindig meghívódik, amikor a MIDlet Paused-ből Active-ba vált, ide nem érdemes egyszeri beállításokat írni, ezt a MIDlet konstruktorában tegyük meg.

void notifyDestroyed()

jelzi az AMS-nek, hogy Destroyed állapotba került (nincs `destroyApp()` hívás), magyarul ezzel zárhatjuk be a MIDletünk.

void notifyPaused()

jelzi az AMS-nek, hogy Paused állapotba került a MIDlet (nincs `pauseApp()` hívás)

void resumeRequest()

jelzi az AMS-nek, hogy újra Active állapotba szeretne kerülni a MIDlet (meghívódik a `startApp()`)

MIDletStateChangeException

javax.microedition.midlet

A következő két metódus dobhatja:

startApp()

Akkor keletkezik, ha a MIDlet valamiért nem tud elindulni. Ezután meghívódik a `destroyApp()`, és befejeződik a MIDlet futása.

destroyApp(boolean unconditional)

Akkor keletkezik, ha a MIDletünk még futni akar. A kivétel figyelmen kívül hagyódik, ha az `unconditional` paraméter `true`.

Display

`javax.microedition.lcdui`

Az adott eszköz képernyőjét és input eszközeit reprezentáló osztály.

static Display getDisplay(MIDlet midlet)

visszaadja a `Display` objektumot, ami a képernyőt reprezentálja a paraméterben megadott MIDletnek.

void setCurrent(Displayable nextDisplayable)

Ezzel jeleníthetjük meg a paraméterben megadott `Displayable` (pl. `Screen`, `Canvas`) objektumot a képernyőn

void setCurrent(Alert alert, Displayable nextDisplayable)

Ezzel jeleníthetünk meg egy `Alert`-et a képernyőn, az `Alert` után a `nextDisplayable` kerül megjelenítésre

Displayable getCurrent()

Az éppen megjelenített `Displayable` objektumot adja vissza

boolean isColor()

A kijelző színes-e

int numColors()

Hány színű a kijelző (pl. 65536), ha az `isColor()` `true`, ha `false`, akkor a szürkeértékek számát (fekete-fehér kijelzőnél 2)

boolean flashBacklight(int duration)

A háttérvilágítást kapcsolja be a paraméterben megadott ideig (millisec). A hívás azonnal visszatér, tehát nem állítja meg a program futását. A visszatérési érték `true`, ha az alkalmazás kontrollálhatja a háttérvilágítást, egyébként `false`. Csak akkor működik, ha a `Displayable` a képernyőn van, egyébként `false`-t fogunk visszakapni, és nem történik semmi.

A háttérvilágítást kikapcsolhatjuk még idő előtt, ha meghívjuk 0 paraméterrel. Csak MIDP 2-től működik.

boolean vibrate(int duration)

A vibrátort (a telefonban...) kapcsolja be a paraméterben megadott ideig

(millisec). A flashBacklightnál leírtak itt is érvényesek (visszatérési érték true/false, 0 paraméter) Csak MIDP 2-től működik.

CommandListener

javax.microedition.lcdui

Interfész osztály, ami magasszintű eseményeket fogad.

void commandAction(Command c, Displayable d)

Ezt az egy metódust kell implementálnunk hozzá, amelyben lekezelhetjük az eseményeket. Minden eseménykor meghívódik. A c paraméter az aktuális esemény, a d Displayable paraméter pedig visszaadja azt az objektumot, ahol az esemény történt.

Command

javax.microedition.lcdui

Egy objektum, ami egy magasszintű esemény tulajdonságait tartalmazza.

Használata:

Command myCommand = new Command(label, type, priority);

String label

A parancs szövege, címkéje, ez fog megjelenni a képernyőn.

int type

a parancs típusa, hogy minek van szánva a parancs.

A definiált típusok: BACK, CANCEL, EXIT, HELP, ITEM, OK, SCREEN és STOP. Hivatkozás rájuk pl. Command.BACK

int priority

a program prioritást használ a parancsok egymáshoz viszonyított fontosságához a képernyőn. A kisebb érték nagyobb prioritást jelent.

Na jó, de mik ezek a típusok, és minek is a prioritás?

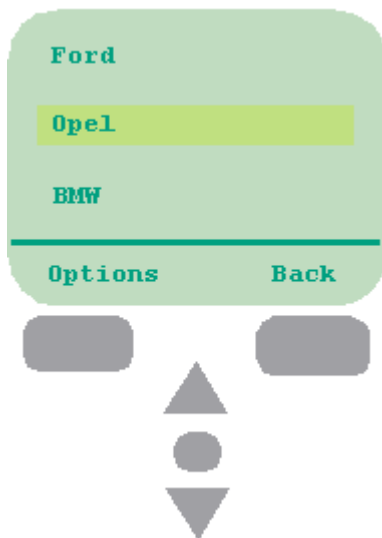
A telefonokon ugye van – általában 2 – multifunkciós billentyű, ezek a Soft Key-ek. Az aktuális rendeltetésük pedig a képernyőn olvasható. Ezeket a rendeltetéseket definiálják a most ismertetett Commandok. Az implementáció először kiválaszja a parancs típusa szerint, hogy hová, melyik helyre kellene kerülnie a parancsnak, aztán prioritásuk szerint helyezi el őket. Hogy micsoda? Parancsot többet is létrehozhatunk, és ami még fontosabb, megjeleníthetjük egyszerre a kijelzőn, de mivel pl. csak 2 Soft Key van, ezért az egy helyre eső több parancsnak létrehoz egy menüt, a Soft Key lenyomásakor megjelenik a lista, és onnan választhatunk. A képernyőn lehetnek egyszerre azonos típusú és prioritású parancsok is.

Egy egyszerű példa: (a Javadoc-ból)

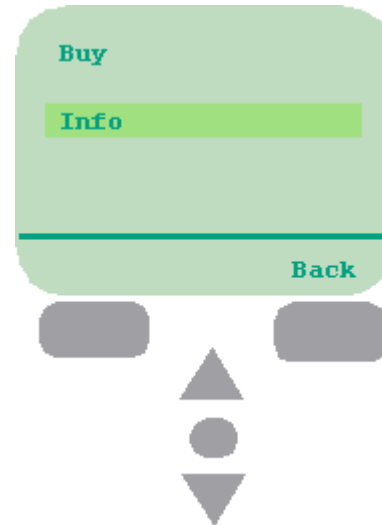
Van három parancsunk, amiket egyszerre szeretnénk megjeleníteni a képernyőn, és így definiáltuk őket:

```
new Command("Buy", Command.SCREEN, 1);
```

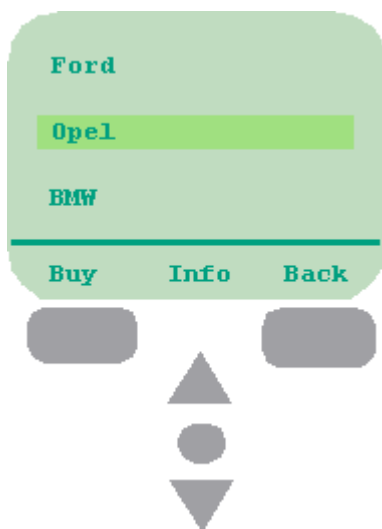
```
new Command("Info", Command.SCREEN, 1);  
new Command("Back", Command.BACK, 1);
```



6. ábra a Buy és Info a bal oldara kerülnének, ezért létrejön nekik egy menü



7. ábra Ha a bal oldali Soft Key megnyomjuk, akkor megjelenik a menü, benne a Buy és Info



8. ábra Ha a telefonnak három Soft Key-e is van, akkor mind a három parancs megjeleníthető

Ahol lehet őket használni:

- Canvas
- Form
- List
- Textbox

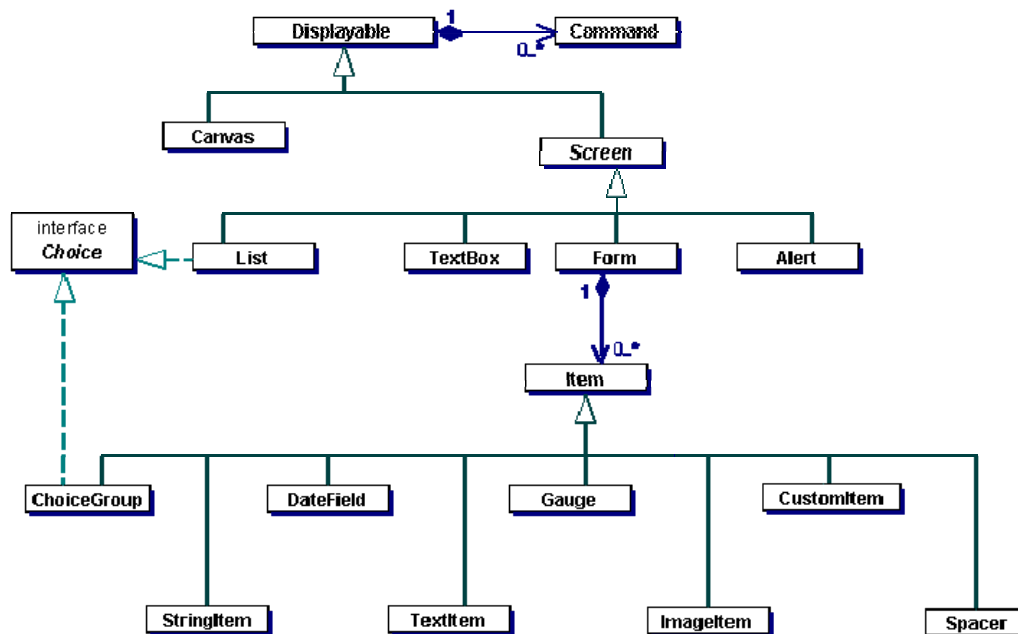
Ahol nem lehet használni:

- Alert
- FullCanvas (Nokia UI API)

Ha mégis be akarnánk állítani a parancsfigyelés a `setCommandListener(...)`-el, `IllegalStateException` kivétel fog keletkezni.

User Interface

Mivel jónéhány megjeleníthető osztály következik, ezért tekintsük át a repertoárt, és hogy hogyan is kapcsolódnak egymáshoz:



9. ábra

Displayable

javax.microedition.lcdui

Ez az alap osztályunk, ami megjeleníthető a képernyőn, absztrakt osztály. Ahogy az ábrán is látni, a Canvas és a Screen az alosztályai, de megemlíteném még a FullCanvas (Nokia UI API, a Canvasból van származtatva, valamint a GameCanvas, ami szintén a Canvas-ból van származtatva, MIDP 2-től létezik, és szintén lehet Fullscreen-ben kezelni)

void addCommand(Command cmd)

Hozzáad egy parancsot a Displayable objektumhoz.

void removeCommand(Command cmd)

Eltávolít egy parancsot a Displayable objektumból.

void setCommandListener(CommandListener l)

Beállítja az aktuális parancsfigyelőt a Displayable objektumhoz, lecseréli az előző CommandListenerert.

void setTitle(String s)

beállítja a Displayable címkéjét (mint windowsban az ablak-fejlécek)

String getTitle()

visszaadja az aktuális címkét

void setTicker(Ticker t)

beállítja a Displayable Tickerjét. A Ticker egy kis scroll-sáv a képernyőn, sokmindent nem is tudunk rajta állítani, a scrollozás iránya és sebessége implementáció-függő.

Létrehozása: **Ticker t=new Ticker("Ez itt a scrollozandó szöveg");**

Ticker getTicker()

az aktuális Ticker lekérdezése

int getHeight()

a Displayable magassága pixelekben

int getWidth()

a Displayable szélessége pixelekben

protected void sizeChanged(int width, int height)

A méret megváltozásakor meghívódó event, a méret a paraméterekben kapott értékekre változott. Az objektum létrehozásakor nem hívódik meg, de pl. a Ticker, Title befolyásolják a méretét.

Canvas

javax.microedition.lcdui

A Canvas osztály hozzáférést biztosít alacsony szintű eseményekhez (billentyűleütések) és grafikus hívásokhoz, amikkel rajzolhatunk a képernyőre.

protected void keyPressed(int keyCode)

Egy billentyű lenyomásakor hívódik meg

protected void keyReleased(int keyCode)

A billentyű felengedésekor hívódik meg

protected void keyRepeated(int keyCode)

Egy billentyű nyomvatartásakor hívódik meg

protected abstract void paint(Graphics g)

rajzol a Canvas-ra a paraméterben megadott Graphics objektum segítségével

void paint(Graphics g)

Lerendereli a Canvas-t, a paraméterben megadott Graphics objektummal tudunk rajzolni

void repaint()

Újrajzolja az egész Canvas-t

void repaint(int x, int y, int width, int height)

Újrajzolja a képernyőt a paraméterben megadott koordinátákon belül

protected void showNotify()

Az implementáció meghívja a **showNotify()-t** amint a Canvas látható lesz a kijelzőn

protected void hideNotify()

Az implementáció meghívja a **hideNotify()-t** miután a Canvas el lett távolítva a kijelzőről

String getKeyName(int keyCode)

Egy informatív szöveget ad vissza a paraméterben megadott billentyűhöz. Pl. KEY_NUM0-9 (a 0-9-es gombok kódjai 48-57-ig mennek sorba, a Soft Key-ek általában negatívak)

Graphics

javax.microedition.lcdui

Egyszerű 2D-s rajzolási lehetőséget biztosít. Rajzolni a Canvasra, GameCanvasra, FullCanvasra tudunk, amiknek implementálnunk kell a paint(Graphics g) metódusát, a paraméterben megadott Graphics objektumot használhatjuk rajzolásra.

void drawString(String str, int x, int y, int anchor)

Kírja a megadott szöveget az x,y koordinátára az aktuális színnel és fonttal, a megadott igazítással.

void drawLine(int x1, int y1, int x2, int y2)

Vonalat húz a megadott 2 koordináta között

void drawRect(int x, int y, int width, int height)

Téglalapot rajzol a beállított színnel és vonalstílussal

void fillRect(int x, int y, int width, int height)

Kitölti a megadott téglalapot a beállított színnel

void setColor(int RGB)

Beállítja az aktuális RGB színt

void setColor(int red, int green, int blue)

Beállítja az aktuális RGB színt

void setStrokestyle(int style)

beállítja az aktuális vonalstílust, ami SOLID vagy DOTTED

Igazítások

Az igazítások a függőleges és vízszintes igazításból állnak össze

Vízszintes:

- RIGHT
- HCENTER
- LEFT

Függőleges:

- BOTTOM
- BASELINE/VCENTER (szövegnél/képnél)
- TOP

Használata: `Graphics.TOP | Graphics.LEFT`

Ez a szöveget/képet úgy helyezi el, hogy a megadott koordináta a kép/szöveg bal felső sarkában van.

System Properties

`java.lang`

Az eszközünktől különböző beállításokat kérhetünk le `System.getProperty(String key)`-el, aminek a visszatérési értéke is `String`. A lehetséges paraméterek, jelentésük, és a default értékük:

microedition.configuration

A J2ME konfigurációjának verziószáma
Default: CLDC-1.0

microedition.profiles

A J2ME profiljának verziószáma
Default: MIDP-1.0

microedition.platform

a jelenleg használt platform
Default: j2me

microedition.locale

a jelenlegi felszín az I18N internacionalizációs szabványhoz
Default: en_US

Ez utóbbi igen jól jön nekünk, ha többnyelvű programot szeretnénk írni. Íme egy lista a visszakapható értékekről:

American English en-US	Finnish fi-FI	Portuguese pt-PT
American Spanish es-US	French fr	Romanian ro-RO
Arabic ar	German de	Russian ru-RU
Brazilian Portuguese pt-BR	Greek el-GR	Serbian sr-YU
Bulgarian bg-BG	Hebrew he-IL	Slovak sk-SK
Canadian French fr-CA	Hindi hi-IN	Slovene sl-SL
Chinese Hong Kong zh-HK	Hungarian hu-HU	Spanish es-ES
Chinese Simplified zh-CN	Icelandic is-IS	Swedish sv
Chinese Traditional zh-TW	Indonesian id-ID	Tagalog tl-PH
Croatian hr-HR	Italian it	Thai th-TH
Czech cs-CZ	Latvian lv-LV	Turkish tr-TR
Danish da-DK	Lithuanian lt-LT	Ukrainian uk-UA
Dutch nl-NL	Malaysian ms-MY	Vietnamese vi-VN
English en	Norwegian no-NO	
Estonian et-EE	Polish pl-PL	

Használata: `String locale = System.getProperty("microedition.locale");`

Alert

Az Alert egy képernyő, ami valamilyen információt közöl a felhasználóval, vár egy meghatározott ideig, aztán a következő képernyőre vált.

Alert(String title)

Egy új, üres Alertet hoz létre a megadott címkével

Alert(String title, String alertText, Image alertImage, AlertType alertType)

Egy új Alertet hoz létre a megadott címkével, szöveggel, képpel és a megadott alert típussal

void setTimeout(int time)

Beállítja az Alert várakozási idejét, ennyi ideig látható az Alert (milliszekundumokban), értéke lehet még a speciális FOREVER

static int FOREVER

FOREVER jelzi, hogy az Alert addig marad a képernyőn, amíg a felhasználó nem nyugtázza le

Használata: myAlert.setTimeout(Alert.FOREVER);

Display display;

display.setCurrent(displayable);

Ha Alert a paraméter, ha lejár, az előző Displayable kerül visszaállításra

display.setCurrent(alert, nextDisplayable);

Ha az Alert lejár, a nextDisplayable kerül a képernyőre

AlertType

static AlertType ALARM

Az ALARM olyan figyelmeztetés, amire a felhasználó előre kérte hogy figyelmeztessék

static AlertType CONFIRMATION

A CONFIRMATION egy megerősítő figyelmeztetés valamire, amit a felhasználó hajtott végre

static AlertType ERROR

Az ERROR valamilye hibára való figyelmeztetés

static AlertType INFO

Az INFO valamilyen nem-fenyegető információra figyelmezteti a felhasználót

static AlertType WARNING

A WARNING valamilyen potenciálisan veszélyes dologra figyelmezteti a felhasználót

DataInputStream

Loading an arbitrary type of file.

Note that the length of the file is 0!

```
InputStream is = midlet.getClass().getResourceAsStream( filename );
DataInputStream dis = new DataInputStream( is );
int data;
while ( ( data = dis.read() ) != -1 ) {
    switch ( data ) {
        ...
        case 13:
            break;
        case 10:
            break;
        default:
    }
}
dis.close();
```

Screen

List

TextBox

Form

Form(String title)

Létrehoz egy új, üres Formot a megadott címkével

int append(Image image)

Egy Image elemet ad a Formhoz

int append(Item item)

Egy Item elemet ad a Formhoz

int append(String str)

Egy szöveges elemet ad a Formhoz

void setItemStateListener(ItemStateListener iListener)

Beállítja az ItemStateListenert a Formhoz, lecserélve az előzőt

javax.microedition.lcdui.game

GameCanvas

A GameCanvas osztály szolgáltatja az alapot a játék felülethez

Layer

A Layer egy absztrakt osztály, ami a játék egy vizuális elemét reprezentálja, mint a Sprite, és a TiledLayer

LayerManager

A LayerManager több Layert fog össze, és kezel

Sprite

A Sprite egy vizuális alapja a játéknak, aminek megjeleníthető 1 kockája a sok frame-ből, amit egy Image-ben tárolunk. A frame-ek változtatásával animálhatjuk a Spriteot

TiledLayer

A TiledLayer is egy vizuális elem, egy cellákból álló rács, ahol a cellákat feltölthetjük különböző képekkel

GameCanvas

void flushGraphics()

Az off-screen buffert a képernyőre másolja

void flushGraphics(int x, int y, int width, int height)

Az off-screen buffer megadott területét a képernyőre másolja

protected Graphics getGraphics()

Visszaadja a Graphics objektumot, amivel a GameCanvasra rajzolhatunk

int getKeyStates()

Lekérdezi a fizikai játékgombok állapotát.

Használata:

```
int keyState = getKeyStates();  
if ((keyState & LEFT_KEY) != 0) {...}
```

Tehát egy ÉS-el megvizsgálhatjuk, hogy az adott gomb le van-e nyomva

void paint(Graphics g)

Megrajzolja a GameCanvas, nekünk kell implementálnunk

Layer

Közvetlen alosztályai: Sprite, TiledLayer

int getHeight(); getWidth(); getX(); getY()

A layer aktuális magassága, szélessége, x, y koordinátája

boolean isVisible()

Visszaadja, hogy látható-e a layer

void move(int dx, int dy)

A Layert a megadott dx, dy távolságra mozgatja

void setPosition(int x, int y)

Beállítja a Layert úgy, hogy annak bal-felső sarka az (x,y) koordinátára esik

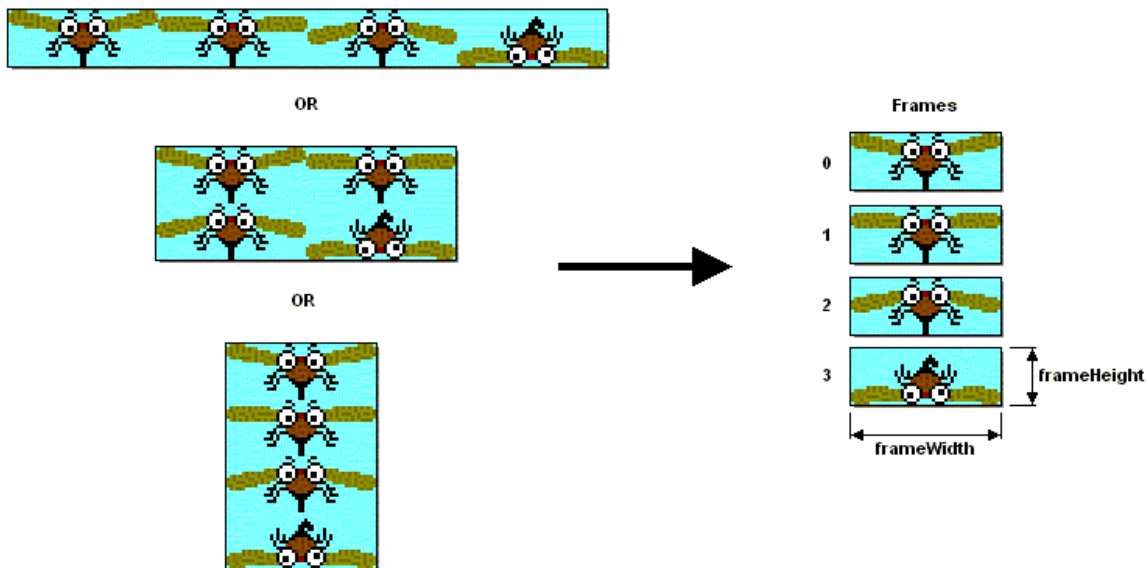
void setVisible(boolean visible)

A Layer láthatóságát állítja

Sprite

Sprite frame-ek

A Sprite framejei egy Imageben tárolódnak. Ha több mint egy frame-et használunk, akkor az Image egyenlő méretű frame-ekre lesz osztva. Ahogy a kép is mutatja, az elrendezés különböző is lehet. A frame-ek indexelve lesznek 0-tól kezdve (10. ábra)



10. ábra

Sprite(Image image, int frameWidth, int frameHeight)

Létrehoz egy új Spriteot a megadott képből, a megadott szélességű és magasságú frame-ekre osztva a képet (10. ábra)

boolean collidesWith(Sprite s, boolean pixelLevel)

Megvizsgálja, hogy van-e ütközés a Sprite és a paraméterben megadott Sprite között.

Ha a pixelLevel true, csak akkor van ütközés, ha nem átlátszó pixelek ütköznek.

Csak a Collision Rectangle-be eső pixeleket vizsgálja.
Ha a pixelLevel false, akkor a Collision Rectangle-k fedését nézi.

void defineCollisionRectangle(int x, int y, int width, int height)

A Spritehoz egy behatároló téglalapot definiál, amit az ütközés-detektáláshoz használ

void setImage(Image img, int frameWidth, int frameHeight)

Megváltoztatja a képet, ami a Sprite framejeit tartalmazza

void defineReferencePixel(int x, int y)

A Sprite referencia pixelét definiálja (13. ábra)

void setRefPixelPosition(int x, int y)

Beállítja a Sprite pozícióját úgy, hogy a referencia pixel az (x,y) koordinátákra esik (14. ábra)

void setTransform(int transform)

Beállítja a Sprite transzformációját. (15-16. ábra)

void setFrameSequence(int[] sequence)

Beállítja a frame-ek sorrendjét a Spritehoz (11-12. ábra)

void setFrame(int sequenceIndex)

Beállítja az aktuális frame-et a frame sequence-ből

void prevFrame()

Az előző frame-re ugrik a frame sequence-ből

void nextFrame()

A következő frame-re ugrik a frame sequence-ből

int getFrameSequenceLength()

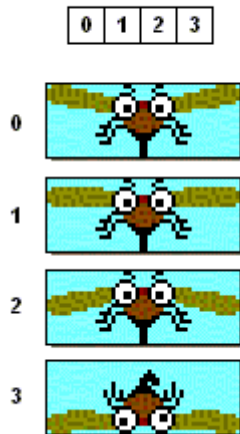
Lekérdezi a frame sequence hosszát

int getFrame()

Lekérdezi az aktuális frame index-ét a frame sequence-ből

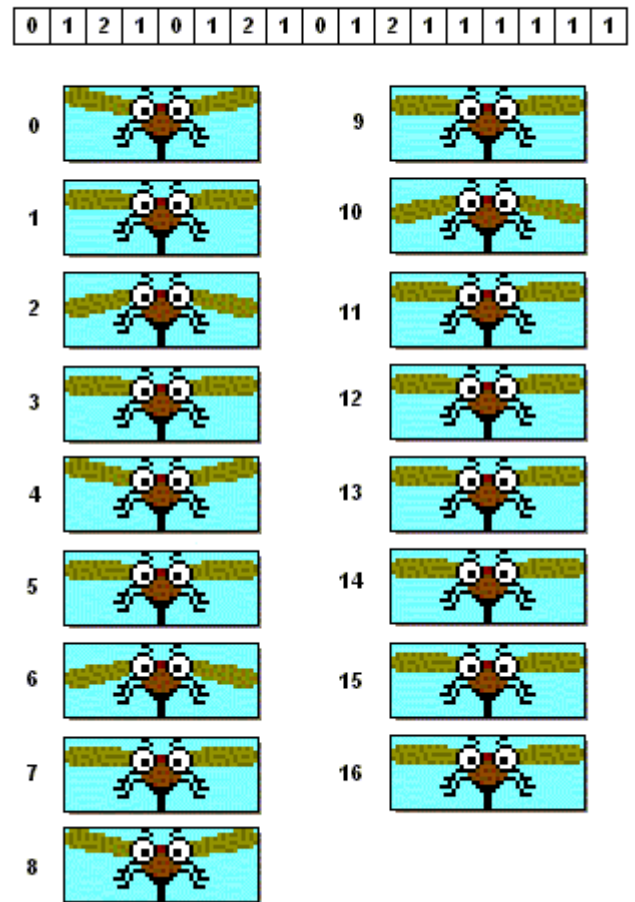
Frame Sequence

Default Frame Sequence



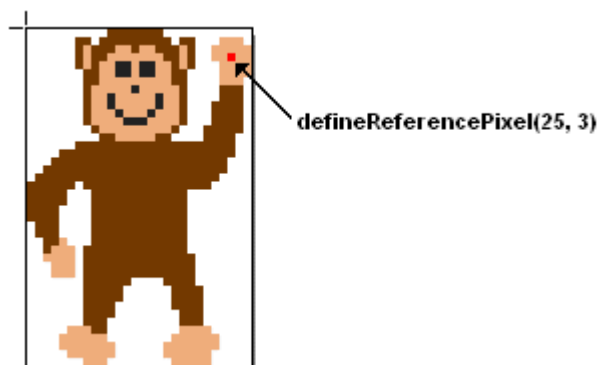
11. ábra

Special Frame Sequence

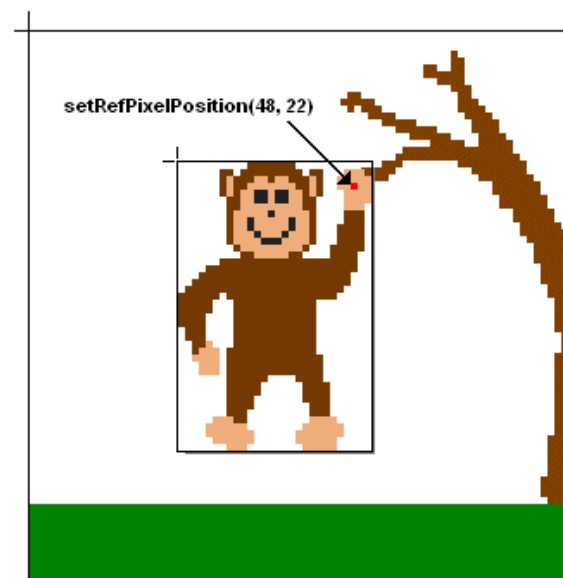


12. ábra

Reference Pixel

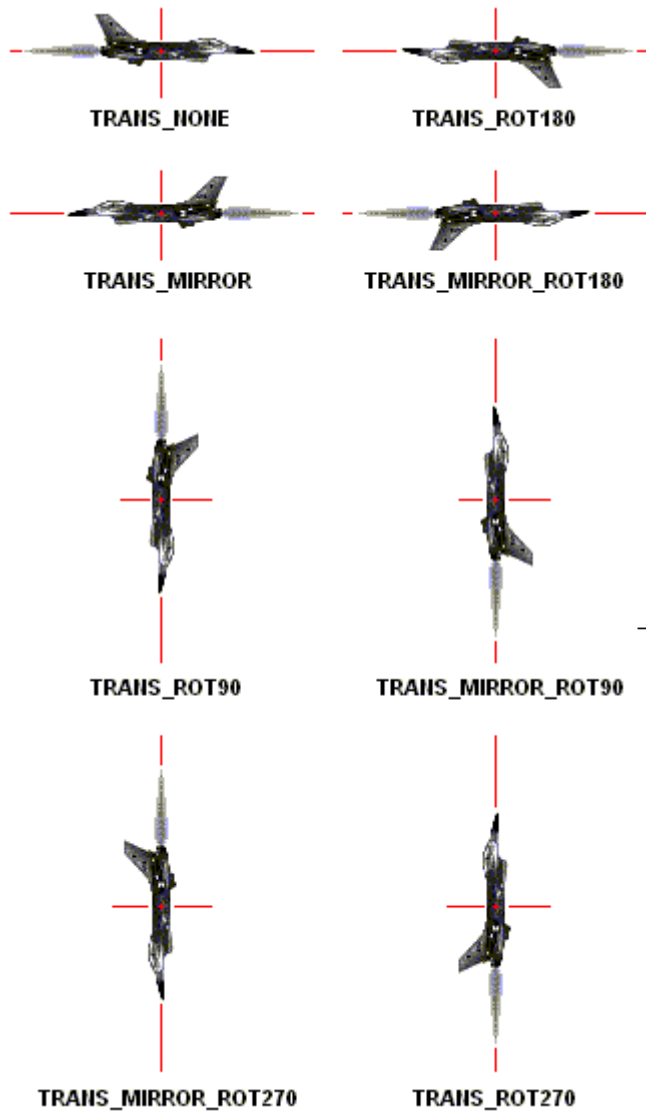


13. ábra

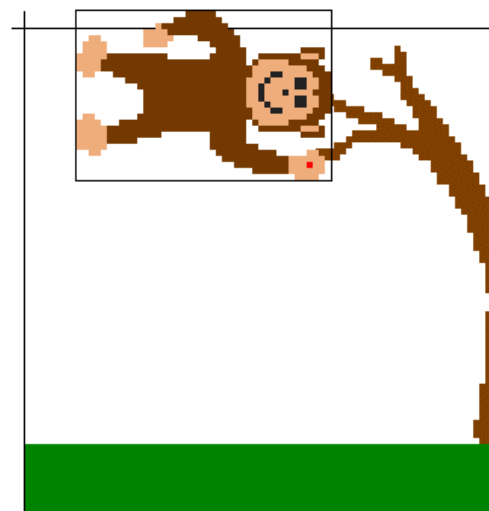


14. ábra

Transforms



15. ábra



16. ábra

TiledLayer

TiledLayer(int columns, int rows, Image image, int tileWidth, int tileHeight)

Egy új TiledLayert hoz létre a paraméterben megadott képből, amit a paraméterben megadott méretű képekre oszt

void setAnimatedTile(int animatedTileIndex, int staticTileIndex)

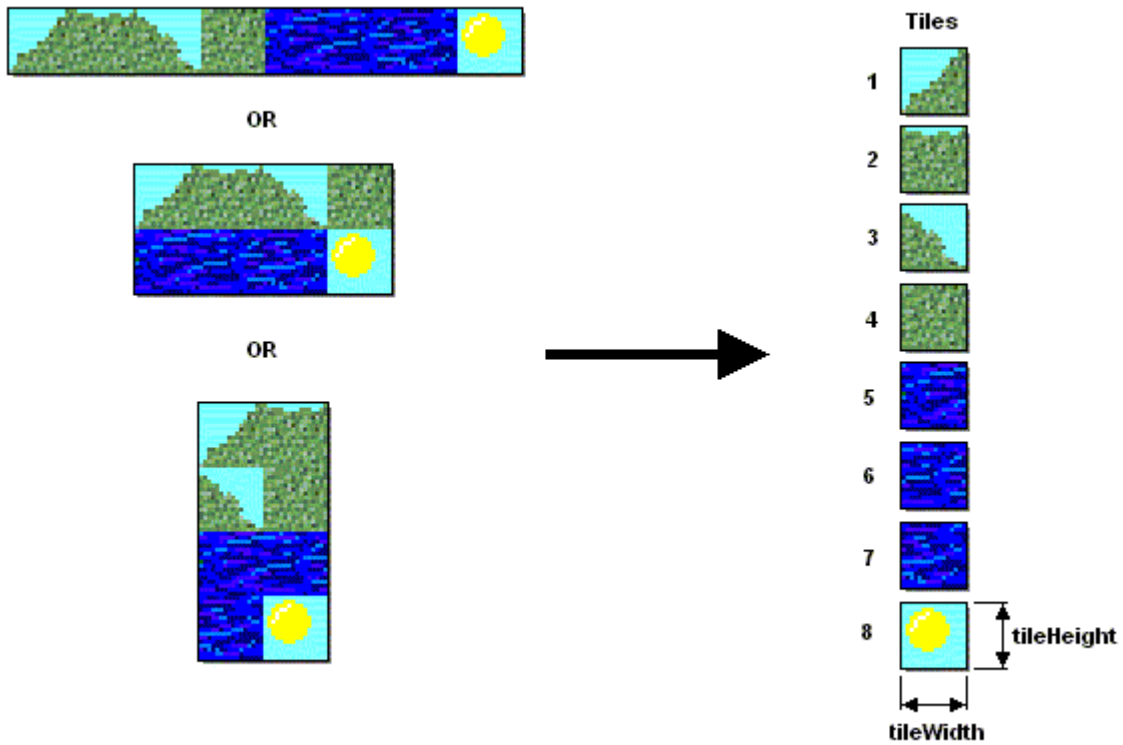
Egy animált cellához rendel egy statikus értéket

void setCell(int col, int row, int tileIndex)

Beállítja egy cella tartalmát a megadott indexre

void setStaticTileSet(Image image, int tileWidth, int tileHeight)
 Megváltoztatja a képet, és paramétereit, amiből a cellákat töltjük fel

Tiles



17. ábra

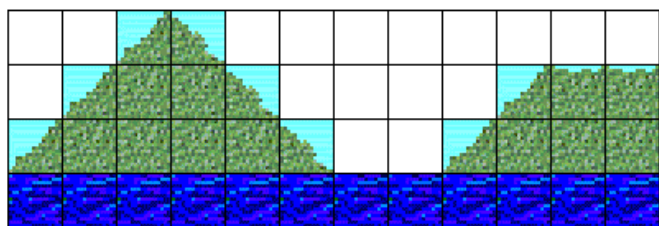
Cells

Cells

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

Animated Tiles

-1 = 5



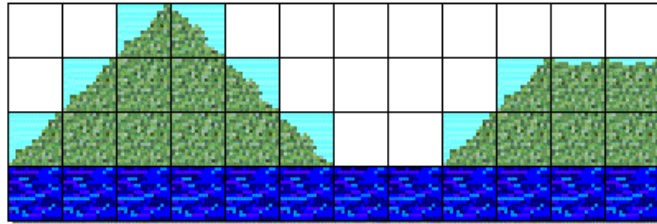
18. ábra

Cells

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

Animated Tiles

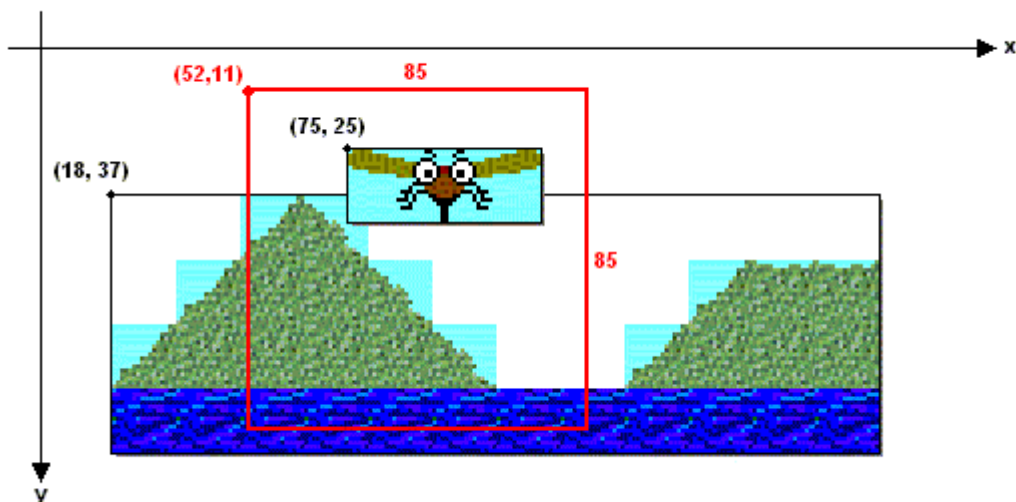
-1 = 7



19. ábra

LayerManager

A LayerManager több Layert kezel, automatikusan lerendereli őket a megfelelő pozíciójukban, automatikusan a meghatározott sorrendjükben. Ahogy a Layer-eket hozzáadjuk, beszurjuk vagy töröljük a LayerManagerhez, mindig van egy folyamatos sorrend közöttük. A legkisebb, 0 indexű van legfelül, a felhasználóhoz legközelebb, a nagyobb indexű pedig távolabb. Beállítható a LayerManagerhez egy megjelenítési ablak, ez az a terület, ami megjelenítődik a képernyőn. Az ábrán egy 85*85-ös terület jelenik meg a képernyőn (az (52, 11) koordinátáknál). A Layerok a LayerManagerhez relatívan jelennek meg, tehát nem a megjelenítési ablakhoz viszonyítva.



20. ábra



21. ábra

void append(Layer l)

Hozzáad egy Layert a LayerManagerhez

void insert(Layer l, int index)

Beszúr egy Layert a megadott indexre

void remove(Layer l)

Eltávolítja a LayerManagerből a megadott Layert

Layer getLayerAt(int index)

Visszaadja a Layert a megadott indexen

int getSize()

Visszaadja a Layerek számát a LayerManagerben

void paint(Graphics g, int x, int y)

A LayerManager aktuális nézetét a megadott koordinátán kirajzolja a megadott Graphics objektumra (a GameCanvas paint-jében) (21. ábra)

void setViewWindow(int x, int y, int width, int height)

Beállítja a megjelenítési ablakát a LayerManagernek a megadott szélességgel és magassággal a megadott koordinátánál (20. ábra, pirossal)

http://www.developer.com/ws/article.php/10927_1475521_1

<http://bdn1.borland.com/article/borcon/files/4136/paper/4136.html>