SZAKDOLGOZAT

Király Edit

Debrecen

2008

Debreceni Egyetem

Informatikai Kar

Mobil alkalmazás fejlesztése Javaban

Témavezető:

Bátfai Norbert

egyetemi tanársegéd

Készítette:

Király Edit

programozó matematikus

Tartalomjegyzék

Ábrajegyzék

I. Bevezetés

Hogy miért pont egy mobiltelefonos játék fejlesztése?
A mobiltelefon mostanra életünk szerves részévé vált. És mivel napjainkban egyre jobban terjednek az olyan készülékek, amik több funkciót is képesek egyszerre ellátni, ezért a legújabb mobiltelefonokat is egyre sokoldalúbbra tervezik. Gondoljunk csak bele, hogy egy telefonnal mennyi mindent lehet csinálni. Míg kezdetben csak telefonálásra volt alkalmas, addig most már az emberek közötti kommunikáció mellett, az egyik legkisebb és legpraktikusabb szórakoztató elektronika lett. Bárhol és bármikor lehet vele zenét hallgatni, fényképezni, kamerázni, világhálóra csatlakozni és nem utolsósorban játszani is. A játék korosztálytól függetlenül mindenkit leköt, például egy hosszabb utazás alkalmával. Ezért van fontos szerepe a mobiltelefonos játékoknak. Az emberek egyre több játékot szeretnének telefonjukra és ahhoz, hogy a legkülönfélébb emberek összes igényét ki lehessen elégíteni, muszáj játékokat kifejleszteni, továbbfejleszteni.

II. Mobiltelefonok világa

II.1 Mobiltelefonok múltja

Mielőtt azonban elmélyülnénk az mobil alkalmazásokban, először kicsit tekintsük át magának a mobiltelefonnak a történetét.1983-ban a Motorola DynaTAC 8000X volt az első hordozható telefon, az 1 kilogrammos súlyával 33x4,5x9 cm-es méretével, 1 órányi beszélgetési és 8 órányi készenléti idejével. De a mai szemmel nézve nagy méret és súly és persze magas ár ellenére több ezres várólistája volt a forradalmi újításnak. A vevőknek tetszett az a koncepció, hogy bárhol bármikor elérhetőek. 10 év kellett ahhoz, hogy a mobiltelefonok infrastruktúrája megépüljön, kiépüljön a hálózat és a licence-díjak is tisztázva legyenek. Ezután már csak a

következő utasításnak kellett eleget tennie a fejlesztőnek: "Megcsináltad, most már nézzen is ki jól!"Ennek eredménye lett 1989-ben a Microtac készülék, mely kisebb méretével és flipes kivitelével sokak kedvencévé vált. Az 1980-as években kiépült először a 150 MHz-es frekvencián működő, majd később a 900MHz-en működő mobiltelefon hálózat.1990-ben mutatták be az első digitális mobilkészüléket az Egyesült Államokban. Egy évvel később Európa bevezette a GSM szabványt. Ezek a hálózatok a korábbinál gyorsabb és erősebb hálózati kap

csolódást tettek lehetővé. Európában általánossá vált a 900MHz frekvencia az 1G-s és 2G-s készülékek esetében, viszont később kezdték lejjebb építeni, hogy helyet biztosítsanak a GSM rendszerek fejlesztésének. A mobiltelefongyártók ebben az időszakban próbálták mobiltelefonjaik méretét egyre kisebbre csökkenteni. Ezt úgy érték el, hogy egyre energiatakarékosabb alkatrészeket építettek bele, így elég volt kisebb akkumulátort is belerakni. Valamint több funkcióval is ellátták a készülékeket. Egyre nagyobb adatátviteli sebességű hálózatokat valósítottak meg, mint például a GPRS és EDGE. Majd következett 2001-ben a 3G-s mobilok forradalma. Az első ilyen hálózatot Japánban mutatták be. Itt már más a működési frekvencia, mely kifejezetten nagy átviteli sebességű és multimédiás szolgáltatások végeláthatatlan sorát nyitja meg a fejlesztők számára. Európában a 3G 2003-ban történt bevezetésre. Becslések szerint 60-féle 3G-s hálózat működik világszerte.

II.2 Mobilfejlődés itthon

Magyarországon a mobiltávközlés a 90-es évek sikerágazata volt. Elsőként 1994-ben Westel 900 és a Pannon GSM kezdte meg mobiltelefonok forgalmazását. Az első forgalmazott telefonok még csak telefonálásra voltak alkalmasak, de ezek a készülékek is igen nagy sikert arattak nagy méretük ellenére. 1995-ben indult be az sms szolgáltatás elsőként márciusban a Pannon GSM-nél, majd októberben a Westel 900-nál. Eleinte csak saját hálózatban lehetett egymás között sms-t küldeni, majd nem sokkal később megvalósítják a hálózatok közötti sms-

kommunikációt. Közben sorban nyílnak mindkét távközlési társaság vidéki képviseletei és persze egyre nagyobb területi lefedettséggel rendelkeznek a szolgáltatók. Majd 1997-ben a Westel sms szolgáltatását kibővítette azzal, hogy e-mailcímre is lehetségessé tette az üzenetküldést. Ugyanebben az évben jelentek meg az úgynevezett feltöltőkártyák, a kártyás előfizetés. 1998-ban kezdtek előbukkanni a Symbian operációs rendszerrel rendelkező mobiltelefonok, melyek grafikus kezelői felületet biztosítanak a felhasználók számára. Ezek az operációs rendszerrel rendelkező telefonok igények szerint alakíthatóak és szoftverekkel fejleszthetőek az operációs rendszer nélküli telefonokkal szemben. Dinamikus memóriakezelésre alkalmasak, melynek lényege, hogy a telefon memóriája bármilyen célra felhasználható, nincs felosztva, hogy milyen alkalmazási területen kell adatokat tárolni benne. 1999-ben jelentek meg az első játékokat is tartalmazó mobilok. Ekkortájt tört be a mobilszolgáltató-iparba a Vodafone Magyarország Rt. (W.R.A.M néven), felvéve a versenyt a már piacon levő két másik szolgáltatóval szemben. E szolgáltató villámgyors elterjedését a Nokia céggel való szoros együttműködése tette lehetővé, melynek során a Nokia cég teljes hálózatot épített ki a Vodafone számára. Ennek következtében 2001-re a cég felzárkózott a másik két konkurenciával szemben. Ekkor már mindhárom szolgáltató 1800 MHz-es frekvenciatartományban működött, sms szolgáltatással és kitűnő lefedettséggel rendelkezett. Folyamatosan rukkoltak elő újabb és újabb akciókkal. Az előfizetők mobilon keresztüli internet- és chathozzáféréssel rendelkeznek. Majd a Westel előrukkol egy új szolgáltatással, az MMS-sel, mely már a multimédia üzenet küldését is lehetővé teszi. Telefonról akár már számlaegyenlegünket is lekérdezhetjük vagy például mozijegyet rendelhetünk. Továbbá mivel készülékeink már alkalmasak letöltésre, a különböző mobilszolgáltatók különböző letöltési felületeket hoztak létre (pl.:Vodafone Live!), ahonnan zenéket, képeket, játékokat, híreket lehetett letölteni. 2002-ben megjelennek a kamerás mobiltelefonok. Ez már várható is volt, mert a multimédia-üzenet akkor tölti ki teljesen célját, ha a mobiltelefon tulajdonos a saját képét/hangját, vagy esetleg videóját is el tudja küldeni más telefonokra, vagy e-mailcímekre. Kezdetben csak kis felbontású kamerák voltak a telefonokban. Majd látván, hogy ez jó, egyre nagyobb felbontású és egyre több beállítási lehetőséggel rendelkező kamerákat raktak telefonjainkba. Megjelent az infra elnevezésű mobilok közti kommunikációs eszköz is, amely két telefon között közvetlen kapcsolatot teremt, valamint olyan készülékek is piacra kerültek, melyekbe már rádióadást vevő antennák és memóriakártyák is belekerültek. A mobiltelefon

annyira elterjedt lett, hogy manapság már az emberek nagy részének csak ilyen telefonja van. A statisztikák szerint sokan kötik ki régi, vezetékes telefonjaikat és használják a kommunikációnak ezt a fajtáját. Talán azért is, mert így bárhol és bármikor elérhetőek mások számára. Talán ennek köszönhető a mobiltelefon sikere és a rohamos fejlődés is.

II.3 Hol tartunk most

Magyarországon három távközlési társaság, a T-Mobile Magyarország Rt. a Vodafone Magyarország Rt. és a Pannon GSM Távközlési Zrt. uralja a piacot. Számos mobiltelefon rengeteg és egyre bonyolultabb funkciókkal. Már senki nem lepődik meg, ha például kedvenceink zenéi szólalnak meg a telefonunkon, vagy ha ezen az egyre kisebb készüléken nézzük meg e-mailjeinket. És lassan a kedvenc játékaink is felkerülnek mobiljainkra. Az USB-kábeleken vagy Bluetooth-on keresztül számítógéppel kommunikálhatnak mobiljaink. Ugyanakkor a rohamos fejlődésnek köszönhető, hogy a mobiltelefonoknak viszonylag rövid idő a lefutási idejük. Tehát megveszünk egy korszerűnek számító telefont és már fél év múlva azt vesszük észre, hogy a mieinknél újabb és jobb mobilok vannak a piacon. Pedig ha belegondolunk, akkor a mobiltelefonnak az eredeti funkciója a telefonálás lenne, így nagyon jól teljesítené a neki kiszabott feladatot egy régebbi telefon is. És már az sem meglepő, hogy az egyre kisebb korosztályt fertőzi meg a mobilláz. Ugyanakkor már annyira hozzátartozik az emberi élethez a mobiltelefon, hogy stresszt és pánikot is okozhat egy elvesztett mobil. A Nemzeti Hírközlési Hatóság statisztikái szerint ma Magyarországon 100 emberre 110 mobil előfizetés jut. De ez még mindig az Európai Uniós átlag alattinak számít. És mindez 13 év alatt? De vajon mi lesz ezután?

II.4 A mobiltelefon jövője

A mobiltelefonok rohamos fejlődését tekintve láthatjuk, hogy még közel nincs vége. Tekintsük például a jövő okostelefonjainak operációs rendszerét. Erre példa a Google Android-ja, melyre a telefonszolgáltatók nagy figyelemmel tekintenek. Az Andorid egy olyan

szoftverkollekció, amely operációs rendszert, fejlesztői környezetet és kulcsalkalmazásokat foglal magában. Kétdimenziós és háromdimenziós grafikus megjelenítőt kapott, SQLite adatbázis kezelővel és optimalizált Java vezérlővel rendelkezik. Támogatja a legelterjedtebb médiafájlokat is (MP3, MP4, JPG, AMR, PNG, GIF). Hardverfüggő érintőképernyővel rendelkezik, támogatja a Bluetooth, EDGE, 3G, WIFI vezeték nélküli hálózatokat is. Fényképezőgépet, GPS rendszert, iránytűt és sebességmérőt is támogat a megfelelő szoftver megléte mellett. Lehetővé teszi a rendszer közelibb alkalmazások fejlesztését, ami magában foglalja például az sms-ek, hívások érkezésének lekezelését is. Az Android valóban teljes körű szolgáltatást ígér és mindezt nyílt forráskód támogatása mellett.

III. Mobilvilág és Java

III.1 A Java programozás kialakulása és tulajdonságai

A mobiltelefonok programozása Java nyelven a legelterjedtebb. Mielőtt azonban rátérnénk a a MicroEdition-ra ismerjük meg a Java kialakulását, elterjedését.

Az OO paradigma alapjai egy diplomamunka által lett megfogalmazva a 1960-as években. Az első olyan nyelv amely tényleg ezt a paradigmát próbálja megvalósítani a Smalltalk volt. De a szakma még sokáig nem értette meg ezt az új paradigmát. Majd az 1980-as években egyre elterjed az objektumorientáltság. Minden nyelvnek megjelenik az OO változata. Ezekben az években lettek egységesek az OO rendszerfejlesztési módszertanok.

Majd 1995-ben megszületik a 90-es évek egyik sikernyelve a Java. A Java nyelvet a Sun Microsystems mérnökei hozták létre James Gosling vezetésével. A fejlesztést 1991-ben kezdték meg, majd '95 májusában jelentették be hivatalosan a létezését. Az első verziót pedig novemberben adták ki Java Developement Kit 1.0 néven. Ez tartalmazta a futtatási környezetet és a fejlesztői eszközöket. Majd ezután következtek a Java többi verziói. Az 1.4-es verzióban új, Java platformhoz kapcsolódó kiterjesztések, architektúrák jelentek meg. Ilyen például aJ2EE vagy a J2ME is. Jelenleg az 1.6-os verziónál tartunk mely, már tartalmaz

például generikus programozást, autoboxing/unboxing-ot, eseménykezelést, annotációkat, újfajta for ciklust.

A Java egyik tulajdonsága, hogy platformfüggetlen, tehát a Javaban íródott programok hasonlóan futnak különböző hardvereken. Ezt úgy tudták megoldani, hogy a fordítóprogram nem fordítja le teljesen gépi kódra a magas szintű forrásprogramot, hanem egy közbenső bájtkódot állít elő. A bájtkód az ember számára olvashatatlan utasításkódokat és hivatkozásokat tartalmazó gépközegi kód. Ezt a bájtkódot az adott számítógépen a Java virtuális gép (Java Virtual Machine) futtatja úgy, hogy az utasításokat értelmezi és natív kóddá (gép kód) alakítja.

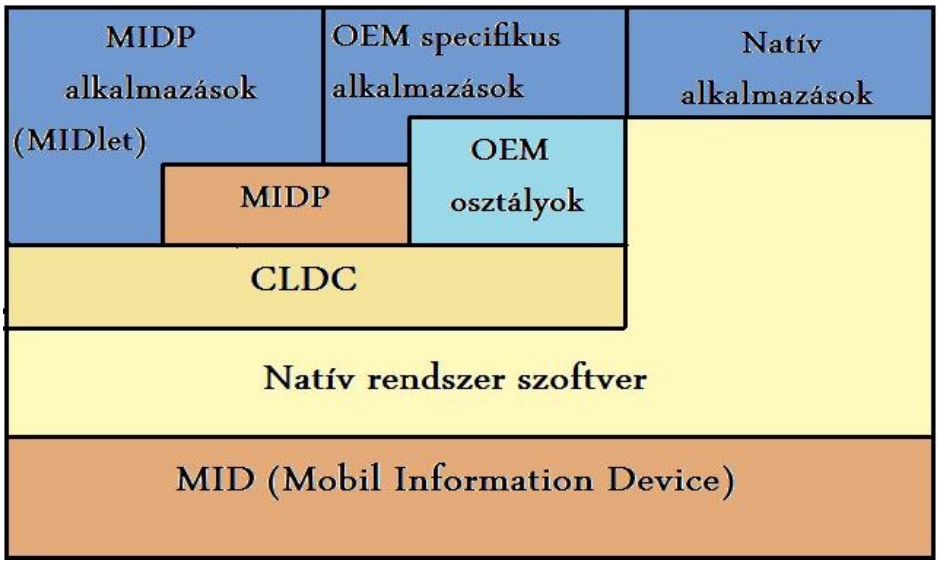
A Java tisztán objektumorientált nyelv, mely azt jelenti, hogy egy osztályhierarchia van felépítve, illetve egy programozási eszköz van, az objektum. A valós világban előforduló dolgokat nevezzük objektumnak. Az objektumnak vannak adatai(tulajdonságai), illetve valamilyen algoritmussal leírható viselkedésmódja. Tehát a Javaban az objektumok számítanak. Létrehozzuk őket (new operátorral), értéket adunk nekik, módosíthatjuk értékeit. Objektumainknak élettartamuk is van. Ez az élettartam addig tart, amíg valamilyen referencia azonosítja az adott objektumot. Amennyiben már nincs, ezt a rendszer figyeli, majd összegyűjti és megszünteti. Ez az automatikus szemétgyűjtögetés. A Javaban nincs szükség és nincs is lehetőség az objektumok egyenkénti megszüntetésére.

III.2 Java ME, a mobiltelefonok programozásának világa

A Java ME tehát a Java programozási nyelv egyik későbbi verziójának kiterjesztéseként jött létre. A J2ME virtuális gépe a KVM (Kilobyte Virtual Machine). Nevéből adódóan kilobájtos méretű (40-80KB!), kifejezetten a mobil környezethez íródott, és igényeknek megfelelően bővíthető. A J2ME profilban az alkalmazások kezeléséért és telepítéséért a JAM (Java Application Manager) felelős. Az egyes alkalmazásokat jar fájlok képében telepíthetjük (ez lényegében egy ZIP fájl, amely tartalmazza az alkalmazásban szereplő osztályok bájtkódját, illetve információkat a JAM számára).

III.3 CLDC konfiguráció

A J2ME platformon belül bevezettek különböző profilokat, hogy azok még inkább illeszkedjenek egy-egy hardver típusra.



1. ábra: J2ME architektúra

Minden ilyen profil alapja az úgynevezett CLDC (Connected Limited Device Configuration – Csatlakoztatott Limitált Eszköz Konfiguráció). A CLDC konfiguráció tartalmazza a J2ME alap osztálykönyvtárát, így az alapvető Java nyelvhez tartozó elemeket(java.lang), az IO kommunikációhoz szükséges osztályokat(java.io), illetve az util osztály egy szűkített készletét.

A CLDC közvetlen a hardver natív rendszer szoftvere felett áll. A CLDC-hez tartozik egy tipikus minimális hardverigény is (ezeket minden eszköznek érdemes teljesíteni):

- Legalább 16-bites CPU.
- Legalább 192KB memória a Java platform számára:
    - minimum 160KB nem felejtő memória: KVM, CLDC library-k.

    - minimum 32KB felejtő memória: KVM.
- Valamilyen limitált hálózati kapcsolódási lehetőség.
- Alacsony energiafogyasztás(akkumulátor).

Egy CLDC-t támogató Java virtuális gép a következőkben nem kompatibilis a Java virtuális gép specifikációval :

- Nem támogatja a lebegőpontos adattípusokat.
- Nem támogatja a JNI-t (Java Native Interface).
- Nem létezik finalize() metódus.
- Nem támogatja a szálcsoportokat és a démon szálakat.
- Nincs felhasználó-definiált Java-szintű osztálybetöltő.
- Korlátozott a hibakezelés. A java.lang.Error legtöbb leszármazottja nem támogatott.
- A Swing és AWT helyett maga alakítja ki a képernyőt, vagy előregyártott képekkel dolgozik.

A CLDC igényli azt, hogy a virtuális gép felismerje a hibás class fájlokat. Egy hibás class állomány interpretálása ugyanis összedöntheti a virtuális gépet. Mivel az J2SE kiadásban használt algoritmus memóriaigényes, ezért a CLDC egy másik módszert használ.A hálózatkezelésre a CLDC az úgynevezett GenericConnection keretrendszert alkalmazza. Kapcsolat létrehozása a javax.microedition.Connector osztály segítségével történhet.
A CLDC konfigurációra épülő profilok:
- **MIDP** (Mobile Information Device Profile): Ezt a profilt kifejezetten mobiltelefonokhoz fejlesztették ki.
- **IMP** (Information Module Profile): ezt a profilt olyan eszközökhöz fejlesztették ki, amelyek csak minimálisan vagy egyáltalán nem rendelkeznek kijelzővel, viszont szükségük van limitált kétirányú kommunikációra. (pl. hálózati kártyák, router-ek, telefonközpontok)

III.4 MIDP profil

Tehát J2ME mobiltelefonokra jutó profilját a MIDP képezi. Mint minden mást, a MIDP-t is folyamatosan fejlesztik. Jelenleg a MIDP 2.0 az elterjedt, de már a MIDP 3.0 fejlesztése is folyamatban van, így mobil fronton újabb követelményeknek megfelelő eszközök tűnhetnek fel hamarosan.

A MIDP 1.0 az első MIDP változat, amelynek eredeti 1.0-ás változatát 2000. szeptember 1.-én adták ki, majd három hónappal a kiadás után elkészült a 1.0a változata is. Ez lett a végleges változat. Mint látható, ez a szabvány az IT terület fejlődési tendenciájához viszonyítva elég régi, ám a későbbi profilok visszafelé kompatibilisek. A MIDP specifikációkat az ún. MIDPEG (Mobile Information Device Profile Expert Group) csoport készíti el, amelynek az 1.0-s változatnál 22 tagja volt (köztük a Nokia, az Ericsson, Siemens, a Motorola és a Samsung) A MIDP 1.0a a mai telefonok képességeihez viszonyítva minimális követelményeket támaszt, de ha belegondolunk, hogy ekkor még csak nagyrészt monokróm kijelzős készülékek uralták a piacot, és ezek között is kevés volt Java képességű, akkor érthetőek ezek a követelmények.

A hardver követelmények a következők voltak:

- Képernyő méret: 96x54 pixel, 1 bites színmélység, megközelítőleg 1:1 arányú kijelzővel.
- Bemeneti eszköz: egykezes-, vagy kétkezes billentyűzet, vagy érintőképernyő.
- Memória: 128KB ROM MIDP komponenseknek, 8KB ROM a MIDP alkalmazások számára szükséges hosszú távon tárolandó adatoknak (pl. beállítások), 32KB RAM a KVM halomterületnek.
- Hálózati elérés: Kétirányú, nem feltétlen folyamatos, korlátozott sávszélességű kapcsolat.

De a hardverkövetelmények mellett szoftverkövetelményeknek is eleget kellett tenniük a készülékeknek:

- minimális kernel, amely kezeli a hardvert (megszakítások, kivételek, minimális ütemezés.). A kernel képes legyen futtatni legalább egy virtuális gépet.
- biztosítsa a nem felejtő memóriából a való olvasás, illetve az oda történő írás lehetőségét
- olvasás és írási hozzáférés az eszköz rádiós hálózati kapcsolatán keresztül
- időkezelés
- minimális bitmap megjelenítése a grafikus kijelzőn
- legalább 1 input kezelése az előző fejezetben említettek közül
- az alkalmazás életciklusának kezelése

A MIDP 1.0-ás változatban kerültek definiálásra a MIDlet alapját képező osztályok és interfészek (javax.microedition.midlet), az lcdui API (javax.microedition.lcdui csomag), a hosszú távú adatok tárolását elősegítő osztályok (javax.microedition.rms csomag), illetve egy elég kezdetleges IO csomag, amely ekkor még csak http kapcsolatokat tudott létesíteni (javax.microedition.io csomag).
A MIDP 1.0-nak igen sok hiányossága volt. Többek között, hogy nem támogatja a közvetlen grafikus műveletek elvégzését, így például a képen megjelenő képpontok színét sem lehet meghatározni. Nem programozhatunk Audio-t, nem kérdezhetjük le a billentyűk aktuális állapotát (csak listener-ek útján értesülhetünk róluk). Később ezeket egyes gyártók próbálták orvosolni kiegészítő csomagokkal. Ilyen például a Nokia UI, amellyel már előállíthatóak egyszerű hangeffektusok és a vibra motor is programozható (természetesen akkoriban még csak Nokia eszközökön).

III.5 MIDP 2.0 újdonságai

A MIDP 2.0 végleges változata 2002. november 5.-kén került kiadásra. A hardver követelmények megegyeznek a MIDP 1.0-éval, kiegészítve azzal, hogy az eszköznek hardveres vagy szoftveres úton képesnek kell lennie hangot kibocsájtani (megjelenik a MIDI és a mintavételezett audio támogatás is). Megjelent a Media támogatáshoz szükséges osztályokat tartalmazó csomag is (javax.microedition.media).

A MIDP 2.0 másik nagy újítása, hogy már tartalmaz olyan osztályokat, amellyel képesek lehetünk biztonságos kapcsolatokat is kezelni (javax.microedition.pki). Minden MIDP 2.0 eszköz legalább az X.509-es titkosítást köteles ismerni.

Az X.509 olyan kommunikációs szabvány, mely az elektronikus tanúsítványok szerkezetére, felépítésére, tartalmára ad előírásokat. Tartalmazza a tanúsítvány verziószámát, egyedi sorozatszámát, a Hitelesítő Hatóság által az aláíráshoz használt algoritmus azonosítóját, a kibocsátó Hitelesítő Hatóság azonosítóját, a tanúsítvány érvényességi idejét, a tulajdonos egyedi azonosítóját, a tanúsítványhoz tartozó nyilvános kulcsot és annak algoritmusát, valamint más, a szabványt kiegészítő ún. toldalékokat.

Kommunikációnál maradva az 1.0-ás változathoz képest az IO csomag is kibővült, olyan osztályokkal, amelyekkel például Socket vagy Stream kapcsolatokat is létesíthetünk a külvilággal.

III.6 MIDP 3.0

Mivel a MIDP 2.0 sem teljesen felel meg a mai elvárásoknak, ezért folyamatosan bővítik újabb és újabb csomagokkal. Ezek már nem tartoznak a MIDP 2.0 alapkövetelményeihez, hanem a MIDP 3.0-át fogják képezni. Mert magáról, MIDP 3.0-ról még nem beszélhetünk. De folyamatosan alakulóban van. Az egyes bővítéseket úgynevezett JSR-ek (Java Specification Request) formájában nyújtják be a résztvevő tagok a MIDPEG csoportnak.

Bővítések, melyek már benne lesznek a MIDP 3.0-ban :

- Wireless Messaging API 1.0 és Wireless Messaging API 2.0 : SMS és MMS küldések és fogadása MIDlet-ekből
- Mobile Media API: hang és videó fájlok lejátszása és vezérlése
- Mobile 3D API: 3D-s grafikus alkalmazások készítése. Többnyire szoftveres renderelő eszközt használva.
- Bluetooth API: Bluetooth szerver és kliens kapcsolatok létrehozása.

– PDA Optional Packages: naptár, névjegyzék és az eszköz fájlrendszerének (belső memória, memóriakártya) elérése.

III.7 MIDP csomagok

A J2ME mivel a Java nyelv egyszerűsített változata, így a Java egyes csomagjai mellett új csomagokat is használ.

Tehát ami a Java nyelvből megmaradt:

- java.lang : A Java alaposztályait tartalmazza (Integer, Math, Object stb.) Ezt a csomagot nem kell importálni, bárhonnan elérhetőek a csomagba tartozó osztályok.
- java.util : Ez az osztály tartalmazza a kisegítő osztályokat. Itt találhatóak a konténerek (Vector, Stack stb.), a naptárral nemzetköziséggel kapcsolatos osztályok (Timezone, Date stb), valamint hasznos interfészek.
    - A J2ME szemszögéből szemlélve az util csomagot, érdemes megemlíteni a Timer osztályt, mely az időzítő feladatait egy háttérben futó szálként hajtja végre, ütemezi őket a későbbi futtatáshoz, ami lehet egyszeri vagy szabályos időközönkénti újraindítás.
- java.io : Az adatok perifériákról történő beolvasására illetve perifériákra történő kiíratására alkalmas osztályokat tartalmazza (InputStream, OutputStream stb).
    - A legjelentősebb szerepe a Connector osztálynak van, mely segítségével távoli kapcsolatot létesíthetünk a tagfüggvények első paraméterében URL címen keresztül (pl. :public static Connection open(String URLcim**,** intkapcsmod, boolean kellekivetel);). Az opcionális második paraméterrel a hozzáférés módját szabhatjuk meg (írás, olvasás, mindkettő), a harmadik, szintén opcionális argumentum pedig arra vonatkozik, hogy akarunk-e kivételt generálni időtúllépés esetén. Mivel I/O-ról van szó, minden függvénye generálja a java.io.IOException kivételt I/O hiba esetén, valamint további kivételeket.

Ezeken az osztályokon kívül a következő csomagok szükségesek még egy mobil alkalmazás fejlesztéséhez:

1.javax.microedition.lcdui :

Osztályhierarchiája a következő:

- **class java.lang.Object**
  - **class** javax.microedition.lcdui.AlertType
  - **class** javax.microedition.lcdui.Command
  - **class** javax.microedition.lcdui.Display
  - **class** javax.microedition.lcdui.Displayable
    - **class** javax.microedition.lcdui.Canvas
    - **class** javax.microedition.lcdui.Screen
      - **class** javax.microedition.lcdui.Alert
      - **class** javax.microedition.lcdui.Form
      - **class** javax.microedition.lcdui.List
      - **class** javax.microedition.lcdui.TextBox
    - **class** javax.microedition.lcdui.Font
    - **class** javax.microedition.lcdui.Graphics
    - **class** javax.microedition.lcdui.Image
    - **class** javax.microedition.lcdui.Item
    - **class** javax.microedition.lcdui.ChoiceGroup
    - **class** javax.microedition.lcdui.CustomItem
    - **class** javax.microedition.lcdui.DateField
    - **class** javax.microedition.lcdui.Gauge
    - **class** javax.microedition.lcdui.ImageItem
    - **class** javax.microedition.lcdui.Spacer
    - **class** javax.microedition.lcdui.StringItem
    - **class** javax.microedition.lcdui.TextField
  - **class** javax.microedition.lcdui.Ticker

Legfontosabb osztályai:

- **Alerttype** : Az Alert osztály objektumainak típusát határozhatjuk meg vele, hanggal észrevehetőbbé tehetjük a figyelmeztetéseket. Például:
  - WARNING : veszélyre figyelmeztető üzenetek átadására szolgáló típus.
  - ERROR : hibaüzenetek átadására szolgáló típus
- **Command** : Szemantikus információkat tartalmaz egy parancs kiváltotta hatásról. A hatást a CommandListener interfész definiálja a CommandAction() módszerével.

- **Display** : Ezzel az osztállyal állíthatjuk be a képernyőt, kezelhetjük a mobilunk egyébb funkcióit, tehát a megjelenítést valósítja meg.
- **Displayable** : Szoros kapcsolatban áll a Display osztállyal, mivel minden megjelenített objektumot ennek az osztálynak az objektumai tartalmazzák.
- **Canvas** : Displayable osztály egyik alosztálya, mely alacsony szintű események kezelésére, illetve bizonyos elemek képernyőn való megjelenítésére szolgál.
- **Screen** : Displayable osztály másik alosztálya, mely az összes magas szintű felhasználói interfész-osztályt jelképezi.
- **Alert** : Valamilyen üzenetről értesíti a felhasználót, majd vár egy bizonyos ideig mielőtt képernyőt váltana. Tartalmazhat szöveget vagy képet. Háromféle időintervallum lehetséges: megszabott ideig, örökre vagy észrevételi parancsig. (pl.: billentyűnyomásig).
- **Form**(Űrlap osztály): Programozási szempontból az űrlap képekből, szövegekből, grafikonokból stb. állhat. Ha az alkalmazás olyan elemet (Itemet) akar beilleszteni egy űrlapba, ami már benne van valamelyik Form objektumban, akkor IllegalStateException keletkezik, azaz egy elem egyszerre csak egy objektumban lehet benne. Az űrlap méretlét a benne lévő elemek száma határozza meg. Az űrlap sorokra osztott, minden sor magassága a benne levő elemek magasságától függ. Az űrlap üres sorral kezdődik, ebbe kerülnek az elemek megfelelő sorrendben. Az elemeket lehet balra, jobbra igazítani, illetve tördelési szabályokat meghatározni.
- **List** : Listát tartalmaz melynek elemei között lépegethetünk, elemei közül választhatunk. A lépegetés és a választás a telefon megfelelő billentyűje segítségével valósítható meg. Többféle listát hozhatunk létre:
  - Kizárólagos : mindig az aktuális elem van kiválasztva
  - Többszörös : egyszerre több elemet kiválaszthatunk az elemek megjelölésével.
  - Implicit : egy megadott parancsot hívhatunk a listaelem kiválasztásával (pl.:menü).
  - Előugró : a kiválasztott elem mindaddig látható, amíg végre nem hajtunk bizonyos lépéseket.
- **TextBox** : Szövegdobozszerű képernyő-objektumokat lehet létrehozni. Aktuális mérete a benne szereplő karakterek számával egyenlő. A szövegdobozban lévő szöveg görgethető, így bármely része szerkeszthető is.

- **Font** : Betűtípus kezelésére szolgál. Itt állítható be a betű stílusa (style), mérete (size), megjelenése (face).
- **Graphics** : Kétdimenziós grafikai műveleteket valósíthatunk meg a kijelzőn. Rajzolhatunk síkidomokat, vonalakat és ha a készülék támogatja, akkor ki is színezhetjük őket 24 bites színmélységben. A kijelző alapértelmezett koordináta rendszere a bal felső saroktól kezdődik (Tehát ott van a (0,0) koordináta). Az x tengely balról jobbra halad, az y pedig felülről lefelé. A rajzolt szöveg vagy kép úgynevezett horgonypontokon (anchor points) illeszkedik a helyére, így a legkisebb számításokkal megjeleníthetőek. A horgonypontokhoz viszonyítva beállítható elhelyezkedésük (középre, balra, felülre stb.). Úgy képzelhetjük el, hogy a szöveg vagy kép egy hajó és horgonyt vet egy pontba. Pl. ha felülre helyezzük a szöveget/képet, akkor a horgonypont a szöveg/kép fölött van. A vízszintes horgonyvetéseket lehet kombinálni a függőlegesekkel egy | jellel, ekkor értékeik összeadódnak.
- **Image** : Ennek az osztálynak a segítségével mozgó- illetve állóképeket készíthetünk alkalmazásunkhoz. Az állóképeken létrehozásuk után már nem változtathatunk. A mozgóképek fehér pixeleket tartalmazó képként jönnek létre, a megfelelő metódus segítségével változtathatjuk őket. Egy mozgóképből készíthető állókép a createImage() metódushívással. A képobjektum PNG formátumú lehet. Ahhoz hogy ezt a képet megjeleníthessük a készüléknek(mobiltelefonoknak) követelményeknek kell eleget tenniük. Ezek közé tartozik például:
    - A PNG kép és a képobjektum hosszának illetve szélességének meg kell egyeznie. A túl nagy értékek
    - memóriakapacitás hiányában nem biztos, hogy megjeleníthetőek.
    - Minden színt (akár alfa tartalommal együtt is) fel lehessen használni, bár a megjelenítés a készüléktől
    - függ.
    - Az alfa tartalommal rendelkező színtípusokat (4 és 6) az alfa értéknek megfelelően tudnia kell
    - dekódolni.
    - Bármely bitmélységet támogatnia kell
    - A készüléknek támogatnia kell a paletta alapú képeket.

- **Item** : Általában a Form objektumokhoz használjuk az elemeket. Egy elem elhelyezése a tördelési beállításoktól függ, amit az adattagok közül választunk.
  pl.**:** LAYOUT_LEFT: balra tördelt elhelyezési stratégia.
  LAYOUT_NEWLINE_BEFORE: az adott elemet egy sortörés után helyezi el.
- **ChoiceGroup** (opciók) : Az űrlapobjektumban elhelyezett választható elemeket tartalmazza ez az osztály. Ezek megvalósíthatók jelölőnégyzetekkel, rádiógombokkal vagy választógombokkal.
- **CustomItem** : Interaktív elemek készítésére szolgál. Az Item osztály tulajdonságaival rendelkeznek, ezenfelül szerkeszthetőek is. Billentyűnyomási vagy mutató kiváltotta eseményeket rendelhetünk hozzájuk, megjelenésüket megváltoztathatjuk.
- **DateField** : Az osztály tagfüggvényeivel dátumokat, időpontokat helyezhetünk el háromféleképpen:
  - időt mutató mező
  - dátumot mutató mező
  - időt és dátumot mutató mező

  Továbbá az időzónát is beállíthatjuk.
- **Gauge** : Grafikonok készítésére alkalmas. Egy grafikon lehet interaktív is, azaz a felhasználó állíthatja be az értékeit. Nem interaktív grafikonok lehetnek határozatlanok. Ez négy grafikon típusnál fordulhat elő: folyamatosan tétlen, növekvő-tétlen, folyamatosan futó, növekvő-frissülő. A növekvő-frissülő esetén nincs ismert végpont, de a grafikon értékeiről tájékoztatást kap a felhasználó. A folyamatosan futó típus esetén nincs végpont, és nem kap tájékoztatást a grafikon értékeiről a felhasználó, csak a futásról értesül. A folytonos-tétlen és növekvő-tétlen esetekben nincs folyamatban lévő munka, nincs aktivitás. Ha határozatlan grafikon esetén a kezdő- és aktuális érték nem az adattagokból ismert állapotok közül való, vagy a maximális érték nem pozitív egész, akkor IllegalArgumentException kivétel keletkezik.
- **ImageItem** : Egy ilyen objektum hivatkozást tartalmaz egy Image objektumra. Az Item osztályból örökölt elhelyezkedési direktívák némelyikét átdefiniálja.

- **Spacer** : Ez az osztály reprezentálja a térközöket és az üres helyeket az elemek sorában. Egy Spacer objektum cimkéje mindig null.Nem rendelhetőek hozzájuk parancsok.
- **TextField** : Az osztály objektumai az űrlapba illeszthető szövegelemek. A maximális méretet mi magunk szabhatjuk, illetve a telefon is korlátozhatja. Ilyen megszorítás például a szöveg formája, hossza stb.
- **Ticker** : Egy a képernyőn folyamatosan futó szöveget hozhatunk létre. Nem lehet leállítani, csak ideiglenesen szüneteltetni (ha a felhasználó nem foglalkozik a telefonnal). Az iránya és sebessége implementációfüggő.

2. javax.microedition.lcdui.game :

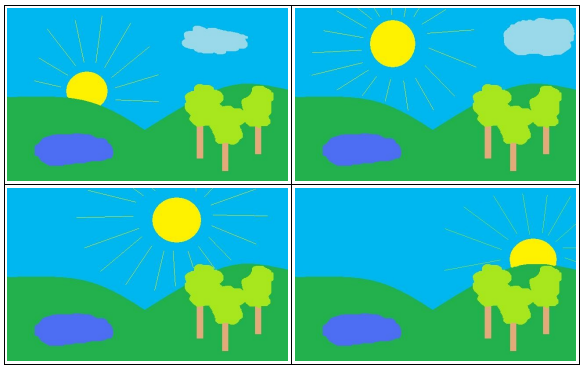
Osztályhierarchiája a következő:

• **class** java.lang.Object

- • **class** javax.microedition.lcdui.Displayable
  - • **class** javax.microedition.lcdui.Canvas
    - • **class** javax.microedition.lcdui.game.GameCanvas
- • **class** javax.microedition.lcdui.game.Layer
  - • **class** javax.microedition.lcdui.game.Sprite
  - • **class** javax.microedition.lcdui.game.TiledLayer
- • **class** javax.microedition.lcdui.game.LayerManager

- **GameCanvas** : A Canvas osztály leszármazottja, feladata a játékok felhasználói felületének megvalósítása. A parancsok hozzárendelését és a bemeneti eseményeket átveszi a Canvastól.
- **Layer** : A játék többrétegű képernyőjét ez az osztály képezi. A rétegobjektum koordinátarendszerét a ráfestett grafikai objektum adja.
- **Sprite** : Segítségével „mozgóképeket” készíthetünk. Ezt úgy oldjuk meg, hogy a mozgóképeket képkockákra bontjuk. Ezek a képkockák váltakoznak a képernyő frissítései során. Az azonos mozzanatokat ismétlésekkel lehet megoldani. Tehát képkockákat a Sprite objektum képelemének feldarabolásával kapjuk meg.

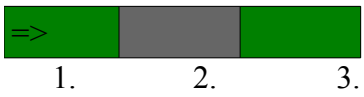
  Ezt a darabolást többféleképpen is megtehetjük. Például így is :



2. ábra: Sprite-ok

De lehet vízszintesen vagy függőlegesen is sorba rakni. Ezekhez a képekhez általában számokat rendelünk, így tudunk hivatkozni rájuk. De ha például megadunk egy hivatkozási pixelt, akkor könnyen tudunk a képen transzformációkat is végrehajtani. Például forgathatunk, tükrözhetünk képeket.

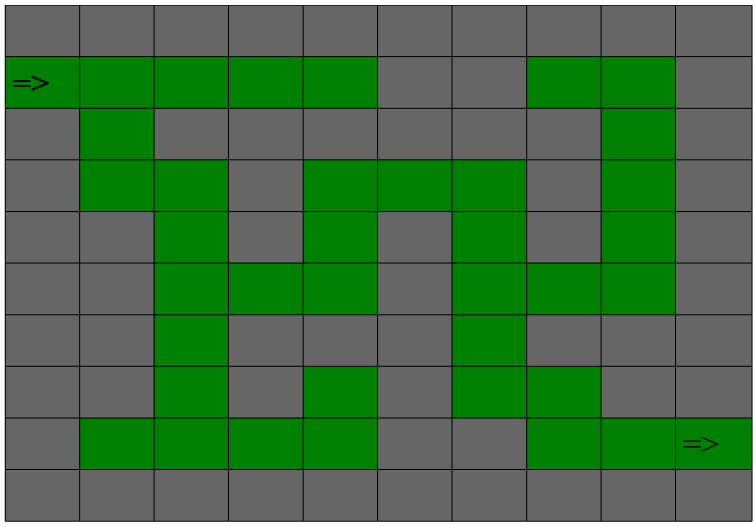
- **TiledLayer** : Kis képkockákból rakhatunk össze képet. Általában olyan képeknél használjuk, ahol sok az ismétlődés. Kétdimenziós játékokban szemléltethetjük a karakterek haladását. A csempék lehetnek statikusak, illetve animáltak. A képet cellákra osztjuk, melyek indexe alapesetben nulla. Ezekhez a cellákhoz rendeljük hozzá a képkockákat. A képek celláinak kitöltését a paint() függvénnyel, valamint a LayerManager osztállyal valósíthatjuk meg.
  Például egy labirintus hátterét könnyen elkészíthetjük az alábbi módon.



3. ábra: Egy TiledLayer képkockái

2	2	2	2	2	2	2	2	2	2
1	3	3	3	3	2	2	3	3	2
2	3	2	2	2	2	2	2	3	2
2	3	3	2	3	3	3	2	3	2
2	2	3	2	3	2	3	2	3	2
2	2	3	3	3	2	3	3	3	2
2	2	3	2	2	2	3	2	2	2
2	2	3	2	3	2	3	3	2	2
2	3	3	3	3	2	2	3	3	1
2	2	2	2	2	2	2	2	2	2

4. ábra: A labirintust reprezentáló táblázat



5. ábra: A táblázat alapján felépülő kép

Lehetőségünk van animált csempék készítésére is. Ezt a createAnimatedTile() metódussal valósíthatjuk meg. Ezeket az animált csempéket statikus csempékből tudjuk létrehozni. Forgathatjuk, tükrözhetjük, olyan benyomást téve, mintha az adott képkocka folyamatosan mozogna. A createAnimatedTile() metódus egy negatív számmal tér vissza és az animált csempe helyére ez az érték fog kerülni.

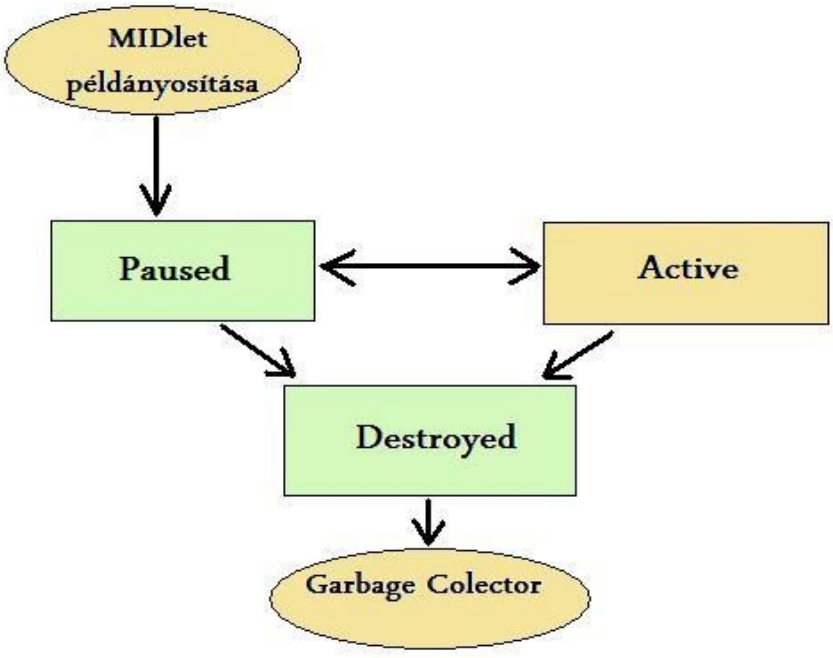
- **LayerManager** : Ezzel az osztállyal kezelhetjük a különböző rétegeket. Egy nézőablakot hoz létre, ami az adott réteg éppen látható állapotát jeleníti meg az alap koordináta rendszerhez képest. Így fel tudjuk tüntetni a képernyőn például a pontszámot, eredményt. A rétegek egymás fölött helyezkednek el. A legalsó réteg az egyes indexű, a legfelső réteg pedig a legnagyobb indexű lesz.

3. javax.microedition.MIDlet

Ez maga a főprogram. MIDletet minden mobil alkalmazásnak kell tartalmaznia.

III.8 MIDletek

A MIDP környezetben futó Java alkalmazásokat MIDleteknek nevezzük.
Ha alkalmazás töltünk le a webről, akkor nem a MIDletet töltjük le és indítjuk, hanem egy ún. MIDlet suit-ot, ami egy vagy több MIDlet-et tartalmaz összecsomagolva.
A MIDlet életciklus modellje:



6. ábra: MIDlet életciklus modellje

Tehát minden MIDletnek tartalmaznia kell egy startApp(), egy pauseApp() és egy destroyApp() metódust, melyek az alkalmazás életciklusát vezérlik.

IV. Saját mobiltelefonos játék fejlesztése

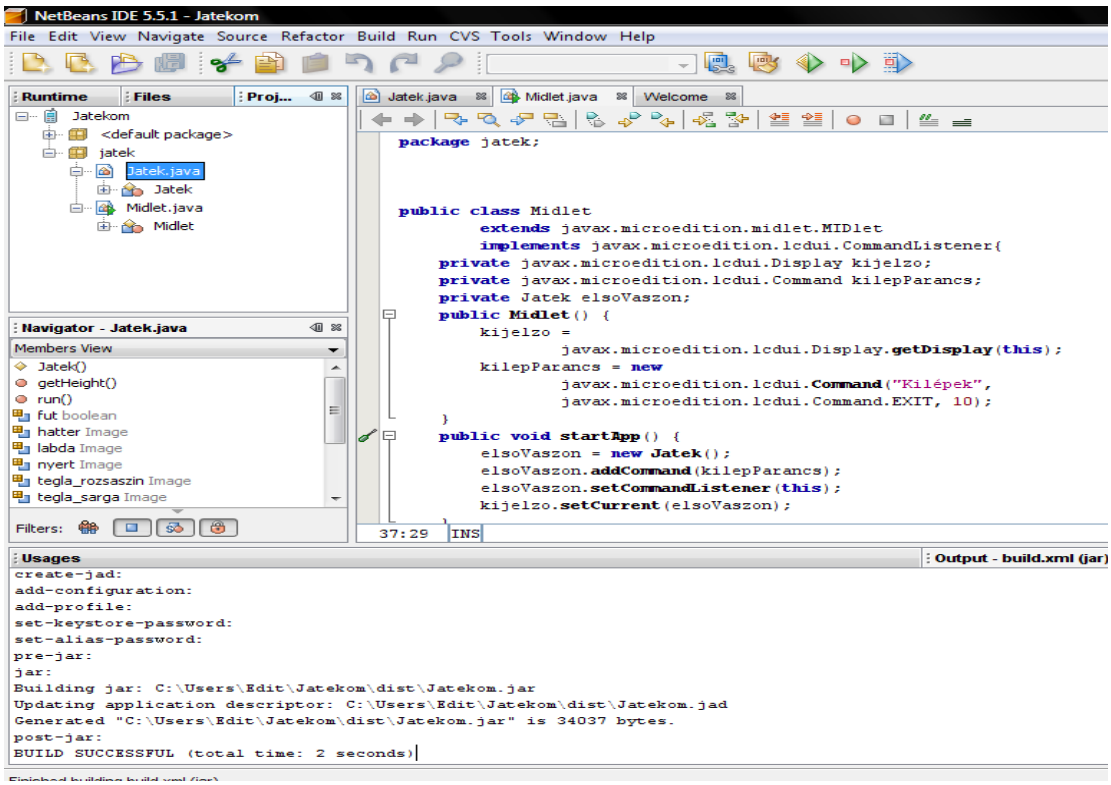
IV.1 Fejlesztői környezet

Játékomat Windows operációs rendszerben, Java nyelven írom Netbeans 5.5.1-es verziójának segítségével. Azért választottam ezt a fejlesztői környezetet, mert nagyon sok olyan kiegészítő elemet tartalmaz ami a fejlesztést megkönnyíti. Ilyenek például:

- Kód írása közben a hibákat felismeri, és egyes hibák esetén a javítási lehetőségeket is felajánlja.
- Forráskódot kiegészíti, osztályelemeket helyez el(gombnyomásra). Példa a get() és set() metódusok automatikus megírása.
- Könnyen és gyorsan át lehet alakítani a projektünket. Erre példa az osztályok nevének megváltoztatása, osztály áthelyezése... stb.

  És nem utolsósorban ingyenes fejlesztői környezet, amit egyaránt kezdő és haladó programozó is használhat bármiféle megszorítás nélkül.

  A Netbeans letölthető a http://www.netbeans.org/community/releases/55/1/ oldalról.



7. ábra: Netbeans IDE 5.5.1

A Netbeans-hez letölthető egy Mobility Pack, ami a mobiltelefonos alkalmazások fejlesztésének segítségére szolgál. Én a Mobility Pack 5.5-ös verzióját használtam a fejlesztések során. Egy emulátor segítségével tesztelhetem játékomat, ami egy valódi mobiltelefont szimulál, így nem kell minden verziót külön telefonra tölteni, ha tesztelni szeretném a játékomat. A telefon gombjaival lehet irányítani a játékot.

8. ábra: Mobility Pack emulátora

Ezt a Mobility Pack-ot még külön kellett letöltenem a http://www.netbeans.org/kb/55/mobility.html oldalról, de ma már lehetőség van olyan Netbeans letöltésére is, amivel együtt le lehet tölteni ezt a csomagot.

IV.2 Játék rövid leírása

9. ábra: Játékom

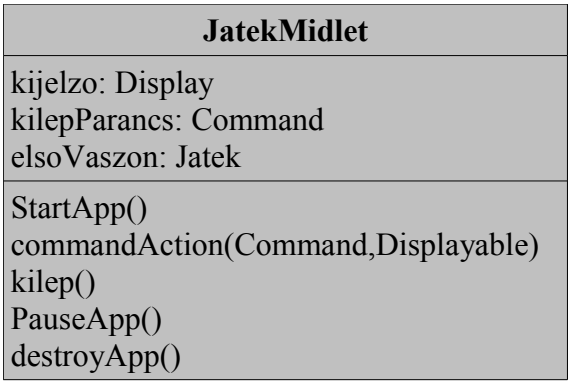
A játékom egy egyszerű faltörő játék. Egy golyó mozog és ütő segítségével lehetőség van irányítani. Ha a golyó hozzáér egy téglához, akkor a tégla eltűnik, „összetörik”. Akkor nyer a játékos, ha az összes téglát sikerül eltüntetnie. Veszít a játékos, ha nem sikerül visszaütnie a golyót, tehát a golyó leesik az ütő mellett. A játék teljesen az emulátorra lett optimalizálva, a mobiltelefonon a méretek eltérőek lehetnek.

IV.3 Játék forráskódjának elemzése

A Játék egy MIDletből (JatekMidlet) és egy osztályból (Jatek) osztályból áll. A Jatek osztályban van magának a játékomnak a működése, míg a MIDlet lényegében a java programok Main osztályát helyettesíti. Minden mobiltelefonra készített alkalmazásnak tartalmaznia kell egy MIDletet.

A játékhoz készítettem egy osztálydiagrammot is, mely lehetővé teszi a játékomban használt két osztály átláthatóságát. Jól látszanak a használt adattagok illetve metódusok.

A játék UML diagrammja a következő:



11. ábra: JatekMidlet

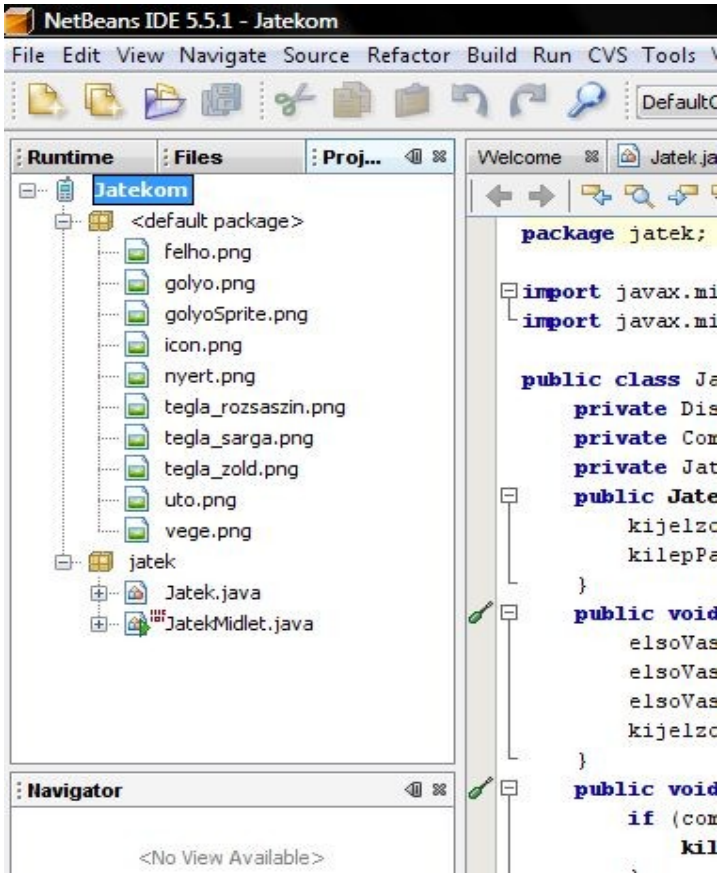
10. ábra: Jatek.java

IV.4 JatekMIDlet

Először a JatekMidlet elemzése következik. Soronként vagy pár sort egyszerre elemezve haladok a jobb érthetőség kedvéért.

Az JatekMidlet első sora egy csomagjelölés. A javaban az osztályokat, midleteket csomagokba rakjuk. Ez elhagyható,de az áttekinthetőség kedvéért én beleraktam egy jatek nevű csomagba. Ha nem rakjuk osztályainkat csomagba akkor egy névtelen, úgynevezett default package-be kerülnek. Ezt láthatjuk az alábbi ábrán is. A játékban felhasznált képek névtelen csomagba kerültek, míg a megírt osztályom és MIDletem a jatek csomagba.

```
package jatek;
```



12. ábra: Játékprogram csomagjai

A következő két sor csomagimportálás. Ez arra szolgál, hogy a midletemben bizonyos csomagokba tartozó elemeket fel tudjak használni. Midletemben szükség lesz az lcdui és a midlet csomagra, így ezeket importálom be, ami a következőképpen történik:

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

Következő sorok már magát a midletet tartalmazzák, az osztály adattagjait, a konstruktort és az osztály metódusait.
Az osztály neve után feltüntettem azt az osztályt, aminek leszármazottja a JatekMidletem, továbbá azt az interfészt amit implementál.

```
public class JatekMidlet extends MIDlet implements
CommandListener{
```

Tehát a JatekMidlet-nek tartalmazza a MIDlet összes adattagját és metódusát. A CommandListener implementálása miatt pedig írnom kell egy ActionListener metódust.
A JatekMidlet három adattagot fog tartalmazni, ezeket deklarálom a következőképpen:

```
    private Display kijelzo;
    private Command kilepParancs;
    private Jatek elsoVaszon;
```

Először egy Display típusú adattagot deklarálok. Ezzel az osztállyal állíthatjuk be a képernyőt, kezelhetjük bemeneti eszközt és kapcsolhatjuk be a készülék
egyéb funkcióit. A Command típusú kilepParancs szolgál a kijelzőn megjelenő események kezelésére. Ebben az esetben a kilépésre.
Az Jatek osztály fogja magát a játékot megvalósítani.

Majd ezután következik a midlet paraméter nélküli konstruktora:

```
public JatekMidlet() {
    kijelzo =Display.getDisplay(this);
    kilepParancs = new Command("Kilépek",Command.EXIT, 10);
    }
```

A konstruktorban először a kijelzőnek adom át a képernyőt megvalósító objektumot, ebben az esetben magát a jatekMidletet. Erre szolgál a „this” szócska. A kijelzőnek való átadást a Display.getDisplay(this) metódushívással valósítom meg.
Majd utána a kilepParancs-ot adom meg. Itt egy Command típusú objektumot példányosítok, melynek négy paramétere van. Az első paraméter tudatja a felhasználóval, hogy az adott parancs mit vált ki. Ez egy minimális hosszúságú szöveget jelent, ez jelenik meg a kijelzőn. A második paramétert nem szükséges megadni. Ez egy hosszú címkét jelent, ha bővebben szeretnénk kifejteni hogy mit vált ki az adott parancs. A harmadik paraméter a hozzárendelt parancs típusát jelöli, azt hogy melyik finombillentyűhöz rendeljük az adott parancsot. A negyedik paraméter pedig a prioritást jelöli. Minél kisebb a prioritást jelölő egész szám, annál fontosabb a parancs.
A következő dolgom hogy a MIDlet-ből örökölt startApp() metódust újraimplementálom úgy, hogy számomra megfelelően működjön.
Ebben a metódusban a new-val példányosítom a saját Jatek osztályomkat, majd az addCommand()-dal hozzárendelem a már előbb inicializált kilépParancsot, illetve a setCommandListener()-rel beállítom a parancsfigyelőt. Végül a midlet képernyőjét az elsoVaszonra állítom. Lényegében ebben a metódusban lép a MIDlet aktív állapotba, lefoglalja a memóriaterületeket:

```
public void startApp() {
        elsoVaszon = new Jatek();
        elsoVaszon.addCommand(kilepParancs);
        elsoVaszon.setCommandListener(this);
        kijelzo.setCurrent(elsoVaszon);
    }
```

Most következő metódust a CommandListener interfész implementálása miatt kötelező megírnom. A CommandListener interfész tartalmazza a commandAction() törzs nélküli metódust. Ha implementálunk egy interfészt, akkor az összes tartalmazó metódusfejhez törzset kell írni. A CommandAction arra szolgál, ha a Command típusú paraméterben a kilepParancs esemény van, akkor lefusson a kilep() metódus.

```
public void commandAction(Command command,Displayable
displayable) {
        if (command == kilepParancs) {
            kilep();
        }
    }
```

A kilep() metódust a következőkben implementálom. Ha a futás erre a metódusra kerül, akkor a Jatek boolean típusú fut adattagja true-ról false-ra változik, illetve a kijelzőt nullra változtatom. A destroyApp() metódusnak adok egy true paramétert, mely futása során a MIDlet futását befejezi, felszabadít minden lefoglalt memóriaterületet. Egy notifyDestroyed() metódust is futtatok, ami jelzést küld arról, hogy az alkalmazás befejezett állapotba került.
Tehát ez a metódus a kijelzőt eltünteti majd lezárja a MIDlet futását.

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

A MIDletemben szerepeltetnem kell még két metódust, mivel a MIDlet-ből öröklődtek. Ezek most nagy szerepet nem játszanak programomban, így nem szükséges törzset írnom hozzá. Csak üres törzzsel tüntetem fel.

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

Tehát ez volt a JatekMidletem.

IV.5 Jatek.java elemzése

Most lássuk a Jatek osztályt.
Az első sor itt is a csomagot jelöli. Tehát a Jatek osztály a jatek csomagba tartozik.

```
package jatek;
```

A második és harmadik sor csomagimportálást jelöl, tehát ebben az osztályban szükségem lesz a game illetve az lcdui csomagra.
E két csomag osztályait – metódusait, adattagjait - fogom felhasználni az osztályomban.

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

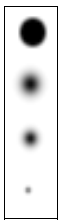
Majd kezdődik maga a Jatek osztály, az osztály nevével, a szülőosztályával és az implementált interfésszel.

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

Tehát a Jatek osztály a GameCanvas osztály leszármazottja. A GameCanvas szerepe a játék felhasználói felületének megvalósítása. A Runnable interfész implementálása miatt pedig majd szükségem lesz a run() metódus törzsének megírására. Így a játék futása könnyen megvalósítható lesz szálkezeléssel.
A következő sorokban deklarálom az osztályom adattagjait.
Az Image típusú adattagok képeket jelölnek. Mivel ezek állóképek, ezért ahogy már fentebb említettem, a képeken létrehozásuk után nem változtathatok.

```
   Sprite golyoSprite,utoSprite;
   Image hatter, golyo, uto, vege, tegla_sarga, tegla_zold,
tegla_rozsaszin, nyert;
    Sprite[] teglak=new Sprite[8];
    Sprite[] teglak1=new Sprite[8];
    Sprite[] teglak2=new Sprite[8];
    boolean fut = true;
```

Tehát nekem nyolc képobjektumom van, amit majd az alkalmazás futása során látni lehet.
Szükségem van még továbbá három tömbre is aminek elemei Sprite típusúak. Ezekben a tömbökben fogom majd tárolni a tégláimat. Azért érdemes a téglákból, golyóból, illetve az ütőből Sprite-okat készíteni mert így könnyebben tudom majd vizsgálni azt, hogy a golyó ütközik-e a téglával vagy az ütővel, vagy nem.
Az ütő és a téglák egy indexből álló képsorozatok lesznek, a golyó pedig egy 5 indexű képsorozat. A 0. indexű a teljes golyó, majd amikor nem sikerül visszaütni a golyót, akkor a golyó eltűnését szimuláljuk a képsorozat lejátszásával.



13. ábra: GolyoSprite

Van még egy boolean típusú adattagom, ami azt fogja jelölni, hogy az alkalmazásom futó állapotban van-e.
A következő metódusom egy get() metódus, amely visszaad egy int típusú adatot, a kijelző magasságát. Mivel emulátorra optimalizáltam a játékomat, ezért ez a metódus 310-et ad vissza.

```
public int getHeight() {
       return 310;
   }
```

Majd a Jatek konstruktora következik.

```
 public Jatek() {
  super(true);
  fut=true;
  try {
   hatter = Image.createImage("/felho.png");
   labda = Image.createImage("/golyo.png");
   uto = Image.createImage("/uto.png");
   vege = Image.createImage("/vege.png");
   tegla_sarga = Image.createImage("/tegla_sarga.png");
   tegla_zold = Image.createImage("/tegla_zold.png");
tegla_rozsaszin = Image.createImage("/tegla_rozsaszin.png");
   nyert = Image.createImage("/nyert.png");
       } catch(java.io.IOException e) {e.printStackTrace();
}

    new Thread(this).start();
   }
```

Az első dolgom, hogy a Jatek őskonstruktorának paraméterül adunk egy true értéket. A GameCanvas konstruktora ugyanis egy boolean típusú értéket vár, amellyel engedélyezhetjük, vagy letilthatjuk hogy az alkalmazásunkat csak játékbillentyűkkel lehessen irányítani. Ebben az esetben engedélyeztem a csak játékbillentyűk használatát. Utána a boolean típusú fut adattagomkat true-ra állítom, mivel az alkalmazásom futó állapotba lépett.
A következő sorokban a képeket hozok létre a createImage() metódussal a paraméterben megadott forrásokból. Itt relatív útvonallal adom meg a képeim helyét. A futás során megjelenő összes képet itt hozom létre. Tehát a játékomban használt összes képobjektum egy előre megrajzolt képből jön létre. Ezeket a létrehozásokat try catch blokkba kell írnom, mivel itt kiváltódhat kivétel abban az esetben,ha a képek valamilyen oknál fogva nem tölthetőek be.
Majd példányosítom a Thread() osztályt, amelynek paraméteréül adom magát az osztályomat. Ezzel létrehoztam egy szálat, mely szálon a játékom majd futni fog. A start() metódussal indítom el a programszálat.
Egy szál csak olyan objektumot futtathat mely implementálja a Runnable interfészt. Ezért kell implementálnom ezt az interfészt. Amikor a Jatek szál elkezd futni, akkor lényegében a Runnable interfész run() metódusának implementációját futtatja. Tehát maga a programszál addig fut amíg a run() metódus.
A következőkben ezt a metódust követhetjük végig.

```
public void run() {
```

Egy getGraphics() metódus hívásával a képernyő közepére pozicionált képet hozhatok létre.

```
Graphics g = getGraphics();
```

A dx és dy adatokat azért kell létrehoznom, hogy majd a golyó mozgását később tudjam befolyásolni.
Az y és x adattagok a golyó kezdőkoordinátáit adják meg.
A sleepTime- mal kicsit várakozó állásba rakjuk a programunkat.

És deklarálok egy boolean típusú win változót amit először false-ra állítok, mivel nem vagyok nyerő állapotban.

```
int dx = 2, dy = -2;
int y = getHeight() - 100;
int x = getWidth() / 2;
int utoX = getWidth() / 2 - 25;
int utoY = getHeight() - 20;
int sleepTime = 20;
boolean win = false;
```

A következő sorban beállítom, hogy a teljes kijelzőmet kitöltse az alkalmazom. Ezt a metódust a Jatek osztályom a GameCanvas osztályból örökölte.

```
this.setFullScreenMode(true);
```

Majd egy while ciklus kezdődik, ami addig fut, amíg a fut boolean típusú adattagunk true. A fut akkor fog false-ra változni, ha minden téglát „összetörtem”, vagy a golyót nem sikerül visszaütnöm az ütővel.

```
while(fut) {
```

Következő sorban deklarálok egy int típusú változót, mely azt fogja tárolni, hogy a billentyű épp lenyomott állapotban van-e. Ezt adja vissza a GameCanvas osztály getKeyStates() nevű metódusa. Ha a függvény 0-val tér vissza, akkor a billentyű felengedett állapotban van, ha 1-gyel, akkor pedig lenyomottban.

```
int billentyu = getKeyStates();
```

Majd elkészítem az ütőt mozgató utasításokat.

Az ütő csak bizonyos feltételek megléte mellett mozog. Ilyen feltételek, hogy az előbb deklarált int típusú pillentyű értéke 1 legyen. Továbbá azt is figyelembe kell vennem, hogy melyik billentyűt tartom lenyomva. Erre szolgálnak a GameCanvas osztály RIGHT_PRESSED, illetve LEFT_PRESSED adattagjai. Továbbá azt is szem előtt kell tartanom, hogy az ütőmet csak a kijelző széléig tudjam mozgatni. Tehát itt adom meg azt, ha a telefonom bal billentyűjét tartom lenyomva, akkor az ütőm valóban balra mozogjon.

```
if((billentyu & GameCanvas.RIGHT_PRESSED) != 0) {
                if (utoX < this.getWidth()-50)utoX = utoX+3;
              }else if((billentyu&GameCanvas.LEFT_PRESSED)!=0)
{
                if (utoX > 0) utoX = utoX-3;
            }
```

A következőkben pedig a golyó mozgását fogom megadni.
Azokat a feltételeket adom meg, amelyek a golyót a játékteremben, tehát a kijelzőn tartják. Mert ugyebár senki sem örülne neki, ha a golyó egyszer csak eltűnne a kijelzőről.
Tehát a golyóm egy getWidth() szélességű és getHeight() magasságú területen mozoghat. Amennyiben elérné a kijelző szélét, abban az esetben a dx, vagy dy ellenkező előjelűre vált, attól függően, hogy melyik szélét érte el.

```
            x = x + dx;
            y = y + dy;
            if (x >= this.getWidth() - 15) dx = -dx;
            if (x <= -5) dx = -dx;
            if (y >= this.getHeight() - 15) dy = -dy;
            if (y <= -5) dy = -dy;
```

Majd következik a képek kirajzolása a kijelzőn.
Először a hátteret helyezem el a Graphics osztály drawImage() metódusával. Ez egy négy paraméteres metódus. Az első paraméter egy Image típusú objektum, a második, illetve

harmadik,tehát az x és az y pont a horgonypont helyzetét jelöli,amihez képest a horgony értéke alapján a képet megrajzolom. Egy kép úgynevezett horgonypontokon (anchor points) keresztül illeszkedik be a megfelelő helyre.

Ezek segítségével a lehető legkisebb számításokkal megjeleníthetőek. A horgonypontokhoz viszonyítva beállítható elhelyezkedésük (középre, balra, felülre stb.). Majd a Sprite-okat helyezem el a képbe. A golyót, amely-nek megadom a setFrameSequence() metódus segítségével, hogy a képsorozat 0. indexű képe legyen látható először majd a játékomban. Majd kijelölöm a horgonypontokat és belerajzolom a képbe. Ugyanezt teszem az ütővel is.

```
            g.drawImage(hatter, 0, 0, 0);
            golyoSprite.setFrameSequence(new int[]{0});
            golyoSprite.setRefPixelPosition(0,0);
            golyoSprite.setPosition(x,y);
            golyoSprite.paint(g);
            utoSprite.setRefPixelPosition(10,0);
            utoSprite.setPosition(utoX,utoY);
            utoSprite.paint(g);
```

Majd elhelyezem a téglákat is.

Egy for ciklusban végigmegyek a három tégla tömbön és sorban rakosgatom bele a képekből készített Sprite-okat. Itt is ugyanúgy járok el minta golyónál és az ütőnél.

```
    for(int n=0;n<8;n++){
       for (int k=0 ; k <= 213; k=k+30 ){
        teglak[n]= new Sprite(tegla_zold,20,10);
        teglak[n].setRefPixelPosition(10,0);
        teglak[n].setPosition(k,50);
        teglak[n].paint(g);
        teglak1[n]= new Sprite(tegla_sarga,20,10);
        teglak1[n].setRefPixelPosition(10,0);
        teglak1[n].setPosition(k,90);
```

```
            teglak1[n].paint(g);
            teglak2[n]= new Sprite(tegla_rozsaszin,20,10);
            teglak2[n].setRefPixelPosition(10,0);
            teglak2[n].setPosition(k,130);
            teglak2[n].paint(g);
                 }
               }
```

Következő sorokban pedig azok a feltételek láthatóak, melyek eldöntik a játék folyamán, hogy a Nyert vagy a Vége Image jelenjen meg a kijelzőn. Ez ugyebár attól függ, hogy a golyót sikerül-e mindig visszaütnöm az ütővel, vagy sikerül-e minden téglát összetörnöm.
A nyerő állapotot a „win” elem true állapota jelzi. A vesztő állapot ellenőrzéséhez a golyó helyzetét kell figyelembe venni. Vesztő állapot esetén a golyoSprite 1. , 2. , és 3. indexű képkockája fog látszódni, tehát eltűnik a golyó.
De akármelyik állapot is áll fenn, a program futását le kell állítani.
A sleepTime-mal itt is egy időre várakozó állába teszem a programomat, mielőtt a fut adattagomat true-ról false-ra állítva jelzem a program futásának végét.

```
if ((y >= getHeight() - 20) || win) {
 if(win)g.drawImage(nyert,this.getWidth()/2-60,
      this.getHeight()/2-30,0);
      else  {
         golyoSprite.setFrameSequence(new int[]{1,2,3});
         for(int i=1;i<3;i++){
              golyoSprite.setFrame(i);}
        g.drawImage(vege,this.getWidth()/2-60,
         this.getHeight()/2-30, 0);}
                        sleepTime = 1000;
                        fut = false;
                    }
```

Majd azt vizsgálom,hogy a golyó mikor ér az ütőhöz. Nagyon könnyű dolgom van, hiszen csak azt kell megvizsgálnom hogy ütközik-e az utoSprite-tal a golyoSprite-om.

```
if (golyoSprite.collidesWith(utoSprite,false)) {
                dx=-dx;}
```

Majd a következő sorokban jönnek azok a feltételek, melyek alapján a téglákat a golyó „összetöri".
Továbbá amikor hozzáér a golyóm a téglához, akkor azt is be kell állítani, hogy az visszaütődjön a tégláról. Ezt a dx előjelének ellentétesre változtatásával teszem meg. A téglák eltűnését pedig az adott tégla Sprite setVisible() metódusának false-ra állításával oldom meg.

```
 for(int i=0;i<8;i++){
                if(golyoSprite.collidesWith(teglak[i],false)){
                teglak[i].setVisible(false);
                dx=-dx;
                }
                if(golyoSprite.collidesWith(teglak1[i],false))
{
                teglak1[i].setVisible(false);
                dx=-dx;
                }
                if(golyoSprite.collidesWith(teglak2[i],false))
{
                teglak2[i].setVisible(false);
                dx=-dx;
                }
            }
```

Majd a következőkben a nyerő állapotot vizsgálom.

Ehhez beállítom a win-t true-ra, majd végigmenve a három „teglak” tömbön, megvizsgálom, hogy van-e még olyan tégla, mely nincs „összetörve”, tehát a hozzátartozó isVisible() metódus visszatérési értéke true.
Ha van ilyen, akkor a win értéket false-ra állítom, ellenkező esetben, ha minden boolean érték false, akkor a win érték marad true, tehát nyerő állapot áll fenn.

```
            win = true;
             for (int k=0 ; k <8; k++ ){
        if((teglak[k].isVisible())||(teglak1[k].isVisible())||
(teglak2[k].isVisible()))
                      win=false;
```

Majd a GameCanvas osztály flushGraphics() metódusát meghívom, mely a bufferem tartalmát üríti. Minden egyes használathoz egy (az objektum maximális méretével megegyező méretű) dedikált puffer rendelődik, ami minimalizálja a heap használatát. (Heap: egy Java kupac, mely egy alkalmazás futtatási területe)

```
flushGraphics();
```

Majd az osztály lezárásaként a szálat „sleepTime” várakozó állapotba rakom.
A sleepTime várakozási idő random várakozási időt jelent.
Azért kell try- catch blokkba rakni, mert előfordul, hogy egy másik szál is megszakítást kér, és ekkor InterruptedException lép fel.

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

Lényegében ezekből a lépésekből áll a Jatek.java.

V. Összegzés

Tehát, ahogy az elemzésből is kiderült a MIDP csomagok jelentősen megkönnyítették a fejlesztést. Ha például a Sprite-okat, vagy a Layereket tekintjük is látjuk, hogy ez így van. Ami személy szerint nekem a játékom megvalósításánál tetszett, hogy a játékfelületen két tárgy ütközését sokkal könnyebben meg lehet vizsgálni, mint más esetekben. Nagyon sok számolgatástól kíméli meg a programozót. A korlátozott lehetőségek ellenére széles skála nyílik egy fejlesztő számára. A játékomat azért is fejlesztettem ki, hogy bemutassak pár lehetőséget ebből a széles körből. Persze lehetne még továbbfejleszteni. Például több pályát csinálni, pontozást számolni, esetleg egy két nehezítést még belerakni (leeső tárgyak, egy téglát többször kell eltalálni hogy eltörjön stb.). De ebben a kis játékban is benne vannak a mobiltelefonos játék fejlesztésének apró fortélyai, trükkjei. Persze a telefonoknak nagyon sok funkciója van a játékon kívül, amihez már más ismeretek is szükségesek. Ilyenek, a kommunikációs eszközök (Bluetooth), és még számtalan lehetőség, amit a mai okostelefonok már képesek megvalósítani. De számomra mindig is az volt a fontos, hogy amit fejlesztek az érdekeljen. És mivel a játékok állnak a legközelebb hozzám, így számomra egyértelmű, hogy egy olyan dologgal kezdjem a mobilok működésének elsajátítását, ami érdekel is.

VI. Néhány kép a játékról

14. ábra: Játékkezdés

15. ábra: Nyerő képernyő

16. ábra: Vesztő képernyő

VII. Irodalomjegyzék

- Angster Erzsébet : Objektumorientált tervezés és programozás, Akadémiai Kiadó 2003.
- Juhász István : Programozás 2, MobiDiák
- Vartan Piromuian : Wireless J2ME Platform Programming, Prentice Hall 2002.
- http://java.sun.com/javame/technology/index.jsp (Java ME Technology)
- http://hu.wikipedia.org/wiki/Java (Java)
- http://logout.hu/iras/j2me_hobbi_programozas_2_resz_a_java_es_ami_mogotte_van.html (J2ME hobbiprogramozás)
- http://t-mobile.hu/egyeni/rolunk/ceginformaciok/cegtortenet/index.shtml (T-Mobile Cégtörténet)
- http://www.pannon.hu/pannon/sajtoszoba/ceginformaciok/cegtortenet/ (Pannon Cégtörténet)
- http://www.vodafone.hu/egyeni/vodafonerol/magyarorszagon/tortenet_hu.html (A Vodafone Magyarország története)
- http://www.mobilport.hu/?r=7814 (A mobiltelefon eredete)
- http://www.jox.hu/cikkek/836/3 (Google Android–itt a sokat ígérő mobil operációs rendszer)
- http://www.sg.hu/cikkek/57788/10_millio_magyar_11_millio_mobiltelefont_hasznal (10 millió magyar 11 millió mobiltelefont használ)